Data-driven Integration Models for Commercial Cloud Storage

Application and Evaluation in the ATLAS Experiment

Tobias Wegner

A thesis presented for the degree of
Doktor der Naturwissenschaften

Faculty of Mathematics and Natural Sciences
University of Wuppertal
October 8, 2022

Abstract

The issue of storing large quantities of data is already challenging for research institutions, and will become increasingly difficult over the next decade as more data are generated in addition to the data already requiring storage. CERN, as the largest particle physics laboratory globally, faces a particularly critical problem in this regard and, with specific reference to the ATLAS experiment, this is complicated by the upcoming implementation of the High-Luminosity Large Hadron Collider upgrade in 2027. In order to address this issue, novel models for the optimal utilisation of already existing storage systems as well as the integration of further storage solutions are required. However, this is hindered by the absence of sufficient test beds, considering the scale of the problem, as well as an overall lack of research related to alternative storage models at the exabyte scale.
The primary aim of this thesis was to extend the research in this area towards new models and commercial cloud storage. This thesis therefore explored the combination of the Data Carousel model, which is currently being evaluated at CERN, and the Hot/Cold Storage model, which is planned to moderate certain disadvantages of the Data Carousel model. The GACS simulation tool was developed throughout this thesis and then used for the evaluation of the new model combination. The validation of GACS showed a difference between real world data and simulated data of at most 3.3%. Together, the new model and GACS provides a foundation for further investigation of cost-effective and efficient data storage methods at exabyte scale by current R&D programmes at CERN.

Acknowledgements

The work presented in this thesis was continuously supported by the distributed computing and data management groups of the ATLAS Collaboration, and we thank the collaboration for its support and cooperation. Furthermore, the department of physics at the University of Wuppertal and Prof. Dr. Peer Ueberholz from the Hochschule Niederrhein significantly supported this work with frequent discussions and regular reviews. I am also very grateful for the immense support of my CERN supervisor Mario Lassnig, who was always available to offer help and support for numerous topics. Also, special thanks to Alice Willison for a thorough language review and to Christian Albrecht for valuable comments concerning the content of this thesis. I also want to thank my parents for their continuous mental support and patience. This work has been sponsored by the Wolfgang Gentner Programme of the German Federal Ministry of Education and Research (grant no. 13E18CHA).

Declaration

I declare that this thesis has been written by myself and has not been submitted, in whole or in part, towards any previous degree application or professional qualification. Except for where indicated by reference or acknowledgement, I confirm that the work presented in this thesis is my own.
I agree to the presence of audience members who are not members of the examination board.

Wuppertal,

T. Wegner

Contents

1 Introduction **1**

1.1 Context . . . 1

1.2 Motivation . . . 2

1.3 Outline . . . 5

2 Background **7**

2.1 ATLAS Computing Fundamentals . . . 7

2.1.1 ATLAS Data . . . 7

2.1.2 Worldwide LHC Computing Grid . . . 11

2.1.3 ATLAS Distributed Computing . . . 14

2.2 Commercial Clouds . . . 17

2.2.1 Overview . . . 18

2.2.2 Google Cloud Platform . . . 19

2.2.3 Related Cloud Projects . . . 22

2.3 Related Models . . . 23

2.3.1 Data Carousel model . . . 24

2.3.2 Hot/Cold Storage model . . . 25

2.4 Related Simulation Tools . . . 26

2.4.1 Requirements . . . 27

2.4.2 Considered Simulation Tools . . . 28

2.4.3 Discussion . . . 31

3 Simulation Tool Architecture **35**

3.1 Overview . . . 35

3.1.1 Architecture Requirements . . . 36

3.1.2 Simulation Types . . . 37

3.2 Architecture Description . . . 38

3.2.1 Modules . . . 39

3.2.2 Interfaces . . . 46

3.2.3 Common Concepts . . . 48

3.3 Discussion of the Architecture 51
3.3.1 Event Size 51
3.3.2 Simplifications 52
3.3.3 Output Module 53

4 Simulation Tool Implementation 55
4.1 Overview 55
4.1.1 Software Dependencies 56
4.1.2 Design Patterns 57
4.1.3 Simulation Configuration System 58
4.1.4 Simulation Run Time 63
4.2 Module Implementations 65
4.2.1 Infrastructure Module 66
4.2.2 Cloud Module 69
4.2.3 Simulation Module 72
4.2.4 Output Module 78
4.3 Simulation Tool Validation 80
4.3.1 Simulation Setup 81
4.3.2 Parameters calculation and configuration 81
4.3.3 Evaluation 97
4.4 General Scalability Considerations 100
4.5 Run time scalability of the validation 105

5 HCDC model 109
5.1 Overview 109
5.1.1 Motivation 109
5.1.2 Model Variations 110
5.2 Simulation Implementation 114
5.2.1 Infrastructure Configuration 115
5.2.2 Transfer Generator Implemenntation 117
5.2.3 Used Parameters 120
5.2.4 Performance 123
5.3 Evaluation of the Results 123
5.3.1 Methodology 124
5.3.2 Evaluation Implementation 125
5.3.3 Results 127

6 Conclusion 135

List of Figures

1.1 Estimated disk and tape storage requirements for the next decade . . 3

2.1 Persistent ATLAS data formats and their transformations 9
2.2 Data derivation and volume reduction 10
2.3 WLCG hierarchy structure 12
2.4 ADC resource organisation 14
2.5 Schematic of the Google Cloud Architecture 20
2.6 Hot/Cold storage model 25

3.1 GACS modules 39
3.2 GACS infrastructure module composition 41
3.3 Simulation event loop transitioning 45
3.4 Simulation interface architecture 47

4.1 Simulation run time overview 64
4.2 UML class diagram of the infrastructure module 66
4.3 Sequence diagram showing the creation of file and replica objects . . 69
4.4 UML class diagram of the cloud module 70
4.5 Flow chart illustrating the cloud storage cost tracking 71
4.6 UML class diagram of the simulation module 73
4.7 UML class diagram of the event rescheduling 74
4.8 Flow chart of the default transfer manager 76
4.9 UML class diagram of the output module 77
4.10 Flow chart outlining the process of writing output values 78
4.11 Infrastructure setup of the simulation validation 83
4.12 File size distribution comparison 85
4.13 Number of transfers distribution from the monitoring data 90
4.14 Throughput distribution from the monitoring data 95
4.15 Summed daily simulated traffic 99
4.16 Traffic comparison of real world and simulated data 100
4.17 Mean simulated transfer duration per day 101
4.18 Transfer duration comparison of real world and simulated data 102

4.19 First run time scalability test 106
4.20 Second run time scalability test 107

5.1 Schematic of a HCDC model variation 113
5.2 Storage and network configuration of the implemented HCDC model 115
5.3 State transitioning of jobs during production phase. 118
5.4 HCDC simulation output data model 126
5.5 HCDC simulation results: storage filling 128
5.6 HCDC simulation results: waiting time distribution 130
5.7 HCDC simulation results: cold storage filling 131

List of Tables

2.1 Tape and disk volume by data format 17
2.2 Simulation feature comparison 32

4.1 Simulation validation parameters 82
4.2 RSS scores of different distribution functions 87
4.3 Simulation validation results 98

5.1 Storage mappings resulting in different HCDC variations 112
5.2 HCDC simulation parameters 120
5.3 HCDC simulation network configuration 122
5.4 Different storage limits per configuration. 124
5.5 HCDC simulation results: number of jobs 127
5.6 HCDC simulation results: transferred volume 132
5.7 HCDC simulation results: cloud storage costs 133

Acronyms

ADC ATLAS Distributed Computing. 14, 36, 81

AOD Analysis Object Data. 10–12, 17, 24, 110

DAOD Derived Analysis Object Data. 11, 17

DDM Distributed Data Management. 14, 15, 24

DID Data Identifier. 16

DQ2 Don Quijote 2. 16

ESD Event Summary Data. 10

FTS File Transfer Service. 16, 22, 48

GACS Grid And Cloud Simulation. 5, 21, 22, 26, 28, 32–40, 42–45, 48, 49, 52, 54–57, 69, 75, 80, 81, 100, 101, 103, 104, 109, 114, 135, 136, 138, 139

GCE Google Compute Engine. 20, 112

GCP Google Cloud Platform. 19, 22, 23, 43, 69–72, 123, 133

GCS Google Cloud Storage. 19–23, 43, 112, 115–120, 122, 124, 127–129, 131–134, 136–139

HCDC Hot/Cold Data Carousel. 4–7, 23, 27, 35, 38, 109–111, 114–121, 123, 124, 130, 134–139

HL-LHC High Luminosity Large Hadron Collider. 1–3

HLT High Level Trigger. 8, 12

HS06 HEP-SPEC06. 15

IaaS Infrastructure as a Service. 18, 19

JSON JavaScript Object Notation. 47, 56–58, 63–65, 70, 73, 75, 89

LHC Large Hadron Collider. 1, 3, 8, 11, 38

MC Monte Carlo. 8, 9

MIPS Million Instructions Per Second. 27, 28, 31

MTU Maximum Transmission Unit. 28, 32

PaaS Platform as a Service. 18

PanDA Production and Distributed Analysis. 15

ProdSys Production System. 15, 24

QoS Quality of Service. 13, 25, 26, 112

RAW RAW data / detector output format. 8–12, 17, 24

RDO RAW Data Object. 9, 10, 17

RSE Rucio Storage Element. 16

SaaS Software as a Service. 18

SSD solid state drive. 13

STL Standard Template Library. 56, 101, 117

VM Virtual Machine. 18, 19, 29, 30, 32, 33

WFMS Workflow Management System. 14

WLCG Worldwide LHC Computing Grid. 11, 13, 14, 16, 17, 22, 40, 41, 43, 48, 52, 106, 112, 134, 137–139

XML Extensible Markup Language. 57

1 Introduction

1.1 Context

The Large Hadron Collider (LHC) [BC+04; EB08] at CERN is a particle accelerator with a circumference of 26.7km, designed to collide proton-proton or heavy ion beams. The LHC employs two beam pipes, in which bunches of particles are accelerated nearly to the speed of light. Particle bunches in each pipe are accelerated counter-directionally to allow for collisions at specific interaction points built into the LHC. There are multiple detectors for various experiments located at interaction points along the LHC.
The operation of the LHC follows a long-term schedule. The major items in the schedule are *runs* and *long shutdowns*. During runs, the LHC is accelerating and colliding particles. During long shutdowns, the LHC and the detectors are maintained and upgraded. LHC operation started with run 1 in late 2009, followed by long shutdown 1 in early 2013. Run 2 started in 2015 and finished with the start of long shutdown 2 in 2018. The LHC resumed operation with run 3 in 2021. Typically, the hardware as well as the software environments and computing models experience significant modifications during long shutdowns.
An extensive upgrade of the LHC and its detectors is planned to start being used with run 4 in 2027. The upgrade is referred to as High Luminosity Large Hadron Collider (HL-LHC) [AB+17]. The HL-LHC upgrade will significantly increase the number of collisions, and thus increase the expected amount of data generated by the detectors.

One of the largest experiments at CERN is the ATLAS experiment [ATL08]. The centre of the experiment is the ATLAS detector, which is one of the detectors attached to the LHC. The ATLAS detector is a common-purpose detector used for a diversity of research topics. In general, the objective is to detect interactions of subatomic particles with different modules of the detector. The measurement of these interactions is stored as the output data of the detector. The largest amount of data delivered by the detector is produced during particle collisions at the LHC interaction point in the centre of the detector. However, the detector is also able to

measure interactions with particles from other sources, e.g., data can be taken from measurements of cosmic ray interactions.
Furthermore, there is a full software stack to create simulated detector data. The data are created in multiple steps. First, collision data are generated using statistical methods. Second, the collision data are processed by the detector simulation. Third, the output of the simulation is transformed into a format similar to the real detector output.
From the creation to the analysis, ATLAS physics data pass through various transformations and formats. The transformations are used for two main reasons. First, to obtain pre-processed data that are already prepared for analysis. This preprocessing comprises common operations that are required prior to all types of analysis. Second, to split the data in smaller, derived data parts, while keeping the amount of redundant information low. However, the amount of data generated in the scope of the ATLAS experiment is growing quickly. With the start of HL-LHC, the data rate of the detector is expected to increase by at least an order of magnitude. The rate at which collisions are stored is expected to increase from approximately 1.4 kHZ to 10 kHZ. The average data volume of a single collision is estimated to increase from approximately 1.6 MB to 4.4 MB [ATL20].

1.2 Motivation

The expected rapid increase of the data rate challenges the existing computing and data model. In order to continue properly storing, processing, and distributing the data, a large increase of storage, compute, and network resources would be required. ATLAS Collaboration [ATL20] evaluates three types of research and development approaches. (i) The *baseline* scenario covers the minimal set of improvements. It represents the case of continuing largely the same way as in run 2. It only assumes the adoption of current, concrete improvements planned for run 3. (ii) The *conservative R&D* scenario assumes the successful adoption of numerous improvements that are still being researched, such as the *Data Carousel* model, lossy compression, and using fast simulation for most of the detector simulation. (iii) The *aggressive R&D* scenario is the most ambitious approach. It assumes the adoption of the most recent research approaches, including new experimental data formats and numerous ideas of porting parts of the ATLAS software to GPUs.

Figure 1.1 illustrates an estimation of the tape and disk storage requirements for the next decade based on the three different research approaches. The left graph

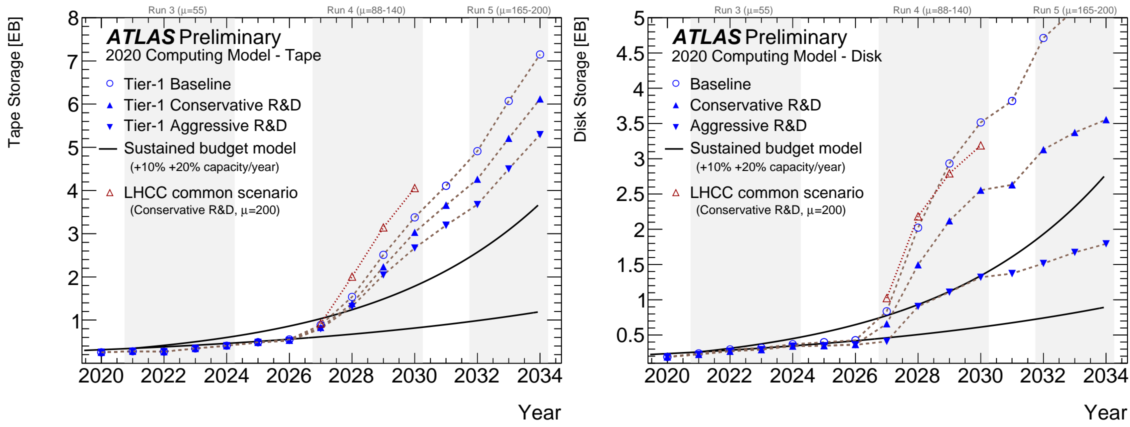

Figure 1.1: Estimated tape storage (left) and disk storage (right) requirements in exabyte for the next decade, assuming the three different research approaches. The solid black lines illustrate a predicted storage capacity increase of 10% and 20%, respectively. [ATL20]

shows the required Tier 1 tape storage, i.e., the required tape storage of the largest computing centres used by ATLAS. The right graph shows the overall required disk storage. Disk storage as high-performance storage and tape storage as archival storage provide the vast majority of storage available to ATLAS. The solid black lines show a yearly storage capacity increase of 10% and 20% assuming a sustained budget. The red triangles show the conservative approach under the assumption that the LHC will allow an average of 200 interactions per proton-proton bunch crossing. The computing conceptual design report [ATL20] concludes that "storage remains the most difficult of the HL-LHC challenges". Figure 1.1 shows that there will be insufficient tape storage starting with the HL-LHC even with the aggressive usage approach. Availability of free disk storage will be limited and most likely negatively impact the performance of analyses. These reasons suggest increasing the research on data management and storage strategies beyond the current models.
There are various ongoing researches related to the resource challenges induced by the increasing data rate. One category of research focuses on topics about improving the information density of the different data formats. For example, this includes new data compression techniques and further reducing redundant information. Another research category investigates alternative data management strategies and storage models. For example, models that allow migrating more data to low-cost storage without unduly reducing the computing performance. Other storage models try moving the problem to the network by storing all data centrally. When data from the central storage are required, it would be cropped to the minimum required amount and transferred to the destination.

Another storage model that is being investigated integrates commercial cloud resources in ATLAS. Most researches related to commercial clouds focus on utilising computing resources and consider cloud storage and network resources only as utility for the computing resources. However, commercial cloud storage provides various advantages. An evaluation of different approaches integrating those resources in ATLAS is required to assemble advantages and disadvantages and to highlight possible technical challenges.
Commercial cloud resources can be adapted in various ways. One of the primary advantages of commercial cloud resources is the flexible allocation and deallocation of resources to a seemingly unlimited extent. Another advantage is that maintaining and upgrading the hardware resources is the responsibility of the cloud provider. Furthermore, the cost models are typically much more explicit and purely usage based.

As mentioned above, one approach to face the upcoming storage challenge is the migration of data to low-cost storage, such as tape storage. The Data Carousel model implements this approach. The Data Carousel model assumes that certain data are solely stored on tape storage. When the data are required for processing, the data are transferred to a more performant storage, such as disk storage, and processed. After processing, the data are deleted from the disk storage.
There are two main concerns about the Data Carousel model. First, tape storage systems typically come with much higher access latency compared to disk storage systems. Especially, random and concurrent data access significantly reduce the tape storage performance. This makes the Data Carousel model preferably suitable for workflows with predictable and infrequent data accesses. Second, assuming the workflows are organised into predictable bulk processing campaigns, there must be sufficient disk storage to hold the bulked data when processing them.
An approach to overcome these concerns is the *Hot/Cold Storage* model. The Hot/-Cold Storage model introduces a cold storage layer between the tape storage and the disk storage. The cold storage layer serves as cache for data that might be required again. Depending on the workflow, the cold storage layer can alternatively be used as buffer storage for predicted bulk data. The combination of the Data Carousel model and the Hot/Cold Storage model is referred to as Hot/Cold Data Carousel (HCDC) model. As part of this thesis, the implementation of the cold storage layer with commercial cloud storage is investigated. This provides a flexible sized cache layer, which enables various options to optimise for cost or performance. The investigated workflow is the production of derived data. In ATLAS this workflow exists in both modes, the infrequent, bulked processing campaigns and the frequent,

unpredictable continuous production.

This thesis discusses the following research questions:

- Using a tape storage system to implement a Data Carousel model for frequently and unpredictably accessed data reduces the processing efficiency by at least 10%.

- Adding a cache-aware model to the tape storage based Data Carousel model allows reducing the on-premises disk storage requirements to less than 5% of the unique input volume without reducing the number of finished jobs by more than 5%.

In this work, these research statements are explored by evaluating an approach to implement the HCDC model into the currently effective ATLAS data flow. Another focus of this thesis is the evaluation of the potential benefit of integrating commercial cloud storage into ATLAS. For this reason, the HCDC model was considered to include commercial cloud storage.
For the evaluation, the HCDC model was implemented in a simulation software. A study of popular existing simulation tools was performed. This showed that most of the existing tools are not up-to-date and not further maintained. The other simulations did not fully conform to the requirements of the required simulation. For this reason, the Grid And Cloud Simulation (GACS) toolkit was developed. Compared to existing simulation software, GACS aims at allowing the implementation of models from a data management perspective.

Results of this thesis have been peer-reviewed and were published in [Weg+22]. In particular, this comprises the investigation and evaluation of the HCDC model described in Chapter 5.

1.3 Outline

Chapter 2 starts with describing the fundamentals and related work required for this thesis. The first part of the chapter explains the ATLAS computing topics including the various data processing chains, data formats, and the organisation of the distributed computing. The second part summarises information about commercial clouds and related projects. The last part gives an overview of related simulation frameworks.
Afterwards, Chapter 3 elaborates on the requirements for simulation software and develops an architecture based on the requirements. The chapter ends with a discussion of the presented architecture.

Chapter 4 specifies the implementation of the simulation architecture. The chapter describes the primary parts, such as the used library dependencies, design patterns, the configuration system, and the runtime. Furthermore, it describes the validation of the correctness and defines the performance boundaries.
Subsequently, Chapter 5 describes the HCDC model, the implementation of the model into the simulation and the evaluation of the results.
Chapter 6 concludes the results in respect to the research questions and derives possible topics for future work from the conclusion.

2 Background

This chapter is organised in four sections. The first section describes the environment and the basic elements for which the HCDC model was developed. This includes the primary types of ATLAS data and their characteristics, the structure of the distributed computing and storage resources, and the projects and systems used to manage these resources.
The second section explains the basics of commercial cloud resources, with particular reference to Google, as this cloud was used throughout this thesis. In addition, the section provide an overview of relevant research related to commercial cloud resources.
The third section describes the Data Carousel model and the Hot/Cold Storage model. These models are especially relevant since their combination - the HCDC model - is simulated and evaluated throughout this thesis.
The last section of this chapter states the requirements to a simulation tool, lists various existing simulations toolkits, and finally discusses the architecture and features of the existing simulation toolkits under consideration of the defined requirements.

2.1 ATLAS Computing Fundamentals

The first part of this section describes which types of data ATLAS uses, how the data are structured, and how the data are processed. The second part describes the structure of the resources used to store and process the data. The last part describes the software that is used to organise and work with the data.

2.1.1 ATLAS Data

The vast majority of the storage space available to ATLAS is required to store physics data originating from the detector or its simulation. The data generated by the ATLAS detector are subsequently grouped in different layers. An *event* is the basic unit of data taking, which contains the data of a single collision of particle bunches. Events are collected in *luminosity blocks*. A luminosity block contains events of approximately one minute of data taking. The events of a luminosity

block are considered to have the same detector conditions, such as the luminosity [Aab+16]. The luminosity blocks generated by a single fill of the LHC are composed to a *run*. A run typically contains up to 8 hours of data taking. Runs made under similar conditions are grouped together in *sub-periods*. Sub-periods are further grouped into *periods*. The last level of data grouping is the *LHC run*. Data grouped in a LHC run contain the periods between two long shutdowns.
Particle bunches accelerated in the LHC are colliding inside the detector with a rate of $\approx$ 40 MHz. The storage space required for an event is $\approx$ 1.6 MB [ATL10]. Storing all events would be an extremely challenging task and result in an unnecessary amount of additional data. For this reason, a two level trigger system quickly estimates for each event: whether to reject or keep it for further investigation. The first level is implemented in custom hardware inside the detector and reduces the event rate to $\approx$ 100 KHz. The second level, the High Level Trigger (HLT), is implemented in software running on a computing farm close to the detector. The HLT further reduces the event rate to $\approx$ 1 KHz. Events passing the HLT are written to persistent storage and are further processed offline.
For accurate data processing, the conditions of the detector are required, e.g., the alignment refers to the exact detector and beamspot position and the calibration describes the background noise of the detector. These conditions are determined and stored in the *condition database*.
As mentioned in Section 1.1, there is a full software stack generating simulated detector data in three main steps. Being able to simulate measurements of the ATLAS detector serves different purposes, of which the most important is - probably - to produce data based on the theoretical understanding of the physical processes in the detector. If this data leads to results similar to the data measured by the detector, it reinforces the correctness of the theoretical understanding of these processes. Furthermore, a simulation eases the process of analysing the exact same event under different conditions or with different properties.
In ATLAS, data or data transformations often contain either the term data or Monte Carlo (MC) in their name. The term data indicates that the origin of the data is the detector, while MC indicates that the data were generated computationally. For example, the derivation of detector data is called data derivation and for simulated data MC derivation. The same is done for the various data formats.

Figure 2.1 illustrates the various transformations and formats of ATLAS physics data. The two origins of the data are given at the top of the graphic. Data from the detector are directly stored in RAW data / detector output format (RAW) files after passing the HLT.

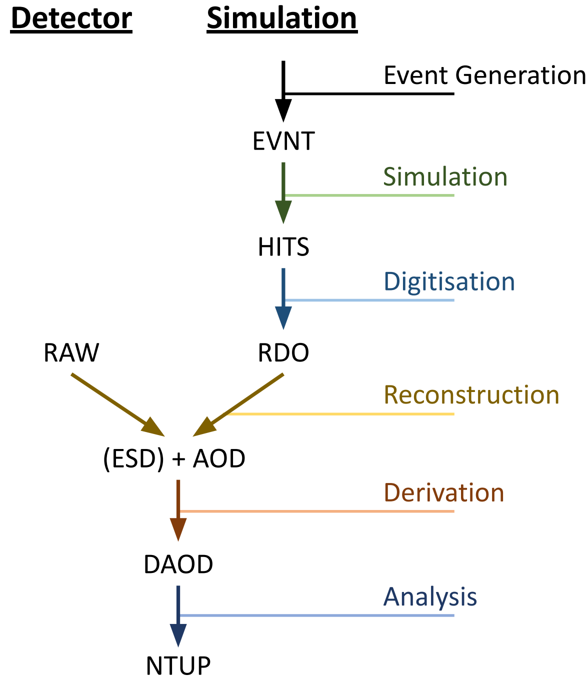

Figure 2.1: Schematic of the persistent data formats of ATLAS. Event generation creates EVNT data. The simulation uses EVNT data as input, generates detector hits, and outputs HITS data. Digitisation is the process of generating detector readout from HITS data. The readout is stored in RDO data. AOD data are created during reconstruction from RDO data of simulated events or RAW data from the detector. AOD data are derived in different DAOD formats for the various physic analysis groups.

The generation of simulated data is divided into three main steps, with the last step creating data similar to RAW data. The first step is the *event generation*. The event generation is the process of computationally generating data that describe collision events. Typically, the approaches to generate events are based on MC algorithms. The generated events are stored in the EVNT format.

The second step is the ATLAS detector *simulation*. The simulation uses the EVNT data as input. The objective is to simulate the occurrence of collision events from EVNT data in the detector. The simulation can be configured with different configurations and detector conditions. The output of the simulation describes the detection of subatomic particles hitting and interacting with various detector modules. The output is stored as data in the HITS format.

The last step is called *digitisation*. The digitisation uses data in HITS format to generate data in the form of the detector readout, i.e., electrical signals of the detector components. The digitisation output is stored in the RAW Data Object (RDO)

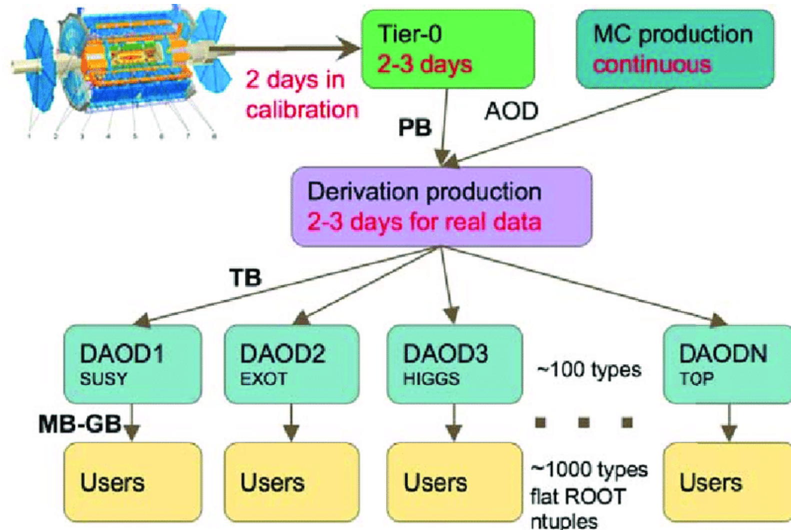

Figure 2.2: Run-2 derivation production workflow. The Figure shows how data data derivation allows reducing the data volume by adjusting the data to different analysis topics. [ATL17a]

format. During the digitisation, trigger- and pileup effects are simulated and added to the RDO data. Other than RAW data, RDO data are intermediate and not archived because they can be reproduced.
The content of RAW data is referred to as *byte-stream* data. Byte-stream data refers to the data representing the detector readout, e.g., the energies measured in the different detector modules. Byte-stream data do not contain any higher level information of the event such as, particle type, trajectories, or similar objects. Since these high level information are required for all physics analyses, the RAW and RDO data are processed and transformed by the *reconstruction*. As RDO data are intermediate, they are typically created and reconstructed in one process [BB+14].
The reconstruction creates the required high level information and stores them in the intermediate Event Summary Data (ESD). The ESD contain all information for each event, including overhead and redundant information. Moreover, ESD are not directly readable by the ROOT software framework [AB+09], which is a widely used analysis tool. For these reasons, the data are further transformed and persistently stored as Analysis Object Data (AOD). Within ATLAS, the actual data files are referred to as AODs [BE+15]. The format of the content of an AOD is called xAOD. The xAOD format is directly readable by ROOT.

Figure 2.2 shows that the data volume used for AOD production is in the order of petabytes. AODs contain a great deal of information. However, not every analysis requires all the information but only a selected set of information, e.g., only specific events or only specific variables from the events. For this reason, AODs are further

derived into the Derived Analysis Object Data (DAOD) format.
Typically, three operations are applied to derive an AODs. *Skimming* removes whole events based on the requirements of the physics topic. *Thinning* removes objects within events. *Slimming* removes variables within specified objects of all events. [ATL19]
A DAOD requires less storage, but different analyses require different DAOD formats. The derivation of AODs allows reducing the volume to the terabyte scale for each DAOD format. The DAOD formats are typically named after their analysis topic. For example, SUSY for Supersymmetry searches, HIGGS for Higgs physics, or TOP for Top-Quark physics.
Since the different DAOD formats are not disjoint, the various DAOD formats lead to numerous duplicated events. Since Run-1 there were 179 different DAOD formats. During Run-2 84 different DAOD formats were still in regular use. For Run-2, events had averagely 10 duplicates in different DAOD formats [ATL19].
Analysis jobs analyse the event data and usually produce NTUP files, which store the results in the form of ROOT ntuples.

Two to three times a year, reprocessing campaigns are performed. This means, the majority of the AODs and DAODs are recreated from the RAW data. A reprocessing campaign might be required when conditions of the condition database were improved or when a major update of the reconstruction and derivation software was released.

2.1.2 Worldwide LHC Computing Grid

ATLAS and the other LHC experiments use resources of the Worldwide LHC Computing Grid (WLCG) [BB+14] to store and process their data. The WLCG is the largest computing grid in the world. A typical attribute of the grid-computing paradigm is the heterogeneity of the combined resources. Another typical attribute is that various institutions, groups, or individuals, provide resources to the grid while keeping the authority and responsibility of the resources. The considered resources provided by the WLCG are computing, storage, and network resources.

The resources of the WLCG are organised hierarchically in four layers (Tier 0 - Tier 3). Figure 2.3 illustrates the four layers from the perspective of the ATLAS experiment. In addition, it shows the connections between the layers and the typical objectives of each layer. The objectives for each tier were partly determined by using monitoring data of 2018, except for archival. The objectives include job types with either the most number of completed jobs or most consumed CPU time. For example, analysis jobs consumed only a very small amount of CPU time at Tier 0,

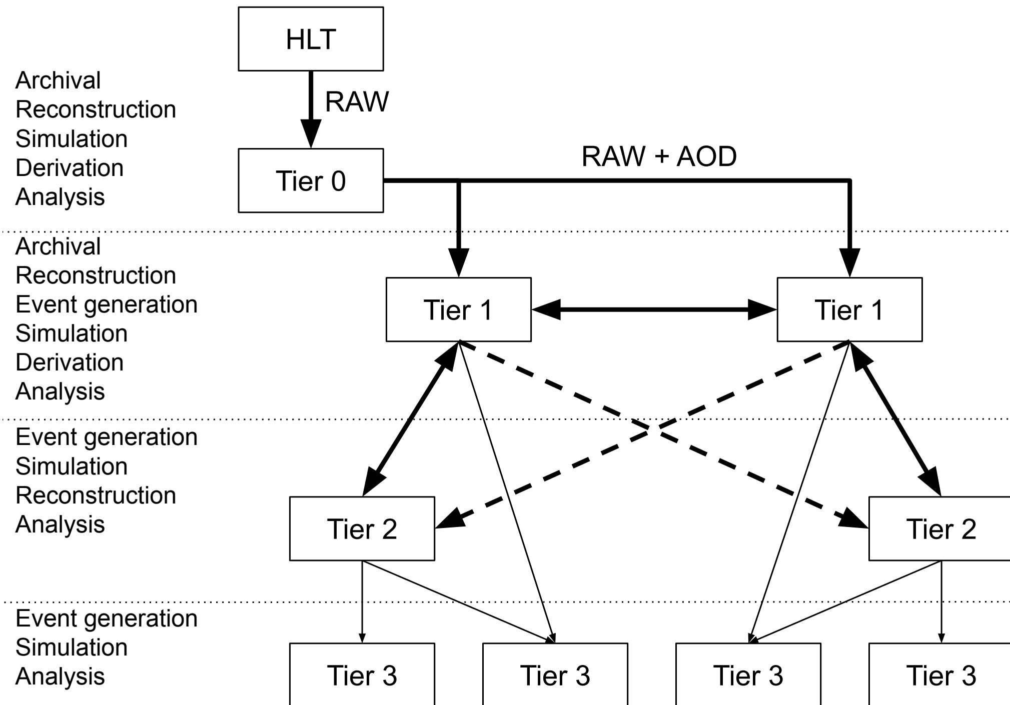

Figure 2.3: The tiered hierarchy structure of the WLCG. Written on the left side are the typical computing task types for each layer. The bold arrows indicate the high-throughput network connections.

but they provided the second most number of completed jobs.

The CERN data centre is the Tier 0 and represents the top level of the hierarchy. All ATLAS data from the HLT pass through the Tier 0. Resources from the Tier 0 are primary used for archiving and for reconstruction of the RAW data. The RAW data and their AODs from the reconstruction are distributed from the Tier 0 to a number (≈ 16) of large Tier 1 data centres.

Tier 1 centres have to guarantee data durability and serve as data archives. Computing resources of Tier 1 centres are also used for reprocessing campaigns of the RAW data. The smaller Tier 2 computing centres access the data at the Tier 1 centres.

Tier 2 centres are usually local universities and institutes in close geographical proximity to a Tier 1 centre. Initially, Tier 2 centres were supposed to operate with a single Tier 1 centre. Because of evolving network capabilities, ATLAS shifts away from this approach and allows workflows among Tier 2 and multiple Tier 1 centres based on network connectivity metrics [BB+14]. For full reprocessing campaigns, Tier 2 centres are employed as well.

Typically, physicists run their analyses on Tier 3 resources. The Tier 3 layer comprises resources that are experiment-affiliated but do not have a computing pledge,

e.g., small clusters of a local institute or personal computers.

The WLCG provides the vast majority of storage space to ATLAS. There are three main types of storage media to consider, which are disk storage arrays, magnetic tape drives, and solid state drives (SSDs). Each storage media provides different Quality of Service (QoS) attributes. The three most relevant QoS metrics for storage are cost, performance, and reliability. In terms of the cost per volume ratio and the performance for various workflows, disk storage is generally placed between tape storage and SSD storage. However, tape and SSD storage are considered to be the more reliable storage types. Tape storage provides the best cost per volume potential. On the other hand, tape storage comes with certain performance drawbacks, e.g., larger access latencies. SSDs operate highly performant but also have a high cost per volume ratio compared to the other storage types. For this reason, they are typically only used as smaller cache or buffer storage.
Most of the persistent storage space available to ATLAS is implemented by disk and tape storage. The difference between the work mode of these two storage technologies results in significant performance differences for certain operations. For example, disks are typically permanently mounted in the corresponding storage system, which avoids having a latency for mounting the medium. Especially in combination with concurrent and parallel processes, these two properties of disk storage allow a significantly reduced access latency compared to tape storage.
On the other hand, tape storage is based on magnetic tapes. Generally, a tape is spooled around a reel build into a cartridge. The cartridges are held in a tape storage library disconnected from the online storage system. This results in a chain of operations to read or write data to a tape. Typically, a tape access operation is firstly queued for some time to allow optimising the access pattern. Afterwards, the required cartridge must be brought online by mounting it into the storage system. This is often performed by robots that are build into the tape storage library. After the cartridge was mounted, the tape must be wound to the required position. Subsequently, the data can be read or written. Then, the tape is rewound to a certain position. Finally, the cartridge gets unmounted and placed back in the tape storage library. These operations can require time in the order of seconds, and thus introduce a significant access latency. Since also the winding of the tape requires a notable amount of time, tape storage typically performs worse for random accesses or concurrent processes.

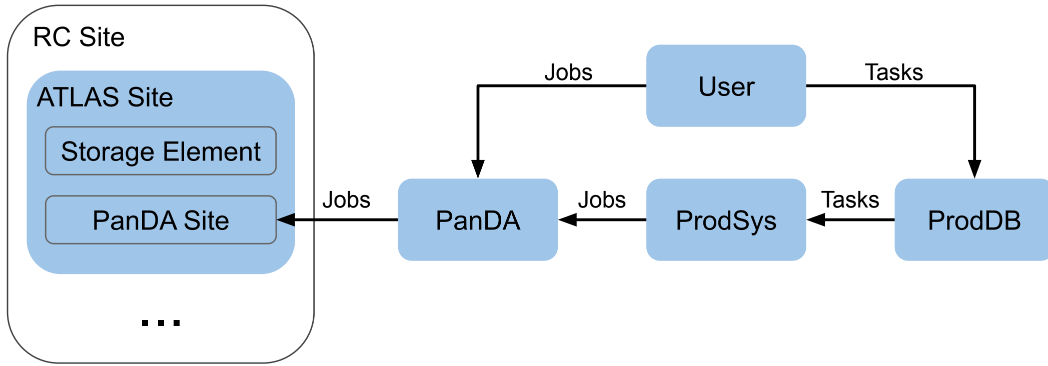

Figure 2.4: Illustration of the different elements and systems in the resource organisation. User or groups generate jobs and tasks, which are resolved by various systems and distributed to an ATLAS site.

2.1.3 ATLAS Distributed Computing

As outlined in Section 2.1.2, ATLAS uses highly distributed systems to store and process data. The group of scientists responsible for developing, maintaining, and operating the software that implements the distributed workflows is called ATLAS Distributed Computing (ADC) [ATL17a]. ADC works on two major activities: the Workflow Management System (WFMS) and the Distributed Data Management (DDM) [ATL17b]. WFMS comprises projects related to components that allow managing the workloads in the WLCG. DDM comprises projects that are related to accessing and distributing data in the WLCG.

Various terms are used to logically structure the computing and storage resources available to ATLAS. The different terms and their relation are illustrated in Figure 2.4. A *resource centre (RC) site* typically pools resources of a single computing facility. An *ATLAS site* is associated to an RC site. An ATLAS site describes the resources available to ATLAS and provides the domain for *storage elements* and *PanDA sites*. A storage element allows the description of a storage area of an ATLAS site, including the storage endpoint addresses. One storage element is associated with exactly one ATLAS site, but an ATLAS site can have multiple storage elements. In such a way, it is possible to create storage elements addressing storage areas with different properties or for different purposes, e.g., each Tier 1 ATLAS site has at least one storage element for disk storage and one storage element for tape storage. A PanDA site addresses a pool of computing resources.

In the context of ADC, a *job* refers to the basic unit for a description of computational work that can be executed by a computing node in the WLCG. Typically, these descriptions comprise information, such as the program to execute, the pa-

rameters for the program execution including required input data, and potentially special hardware requirements. A composition of jobs is called a *task*. Tasks are usually organised in task campaigns, e.g., reprocessing of a particular dataset with new physics algorithms.
RC sites provide a number of job slots depending on the number of available computing nodes. Similar to a storage element that allows describing a certain storage area at a site, *job queues* are used to describe the computing resources of a site. Jobs from a job queue get matched to job slots based on the properties of the job and the job queue.
The Production and Distributed Analysis (PanDA) system is responsible for the matching of jobs to job slots. However, the heterogeneity of resources in grid computing makes it challenging to store and maintain detailed information of every computing node. For this reason, the computing resources are not managed centrally. A key concept of PanDA is implemented in the *Pilot* system, which bypasses the requirement of a centralised management system.
All job queues providing computing resources have to register themselves at the *Harvester* system. The Harvester distributes the Pilot software to the computing nodes. The Pilot is executed on each computing node and investigates its hardware and software capabilities. As part of this investigation, the HEP-SPEC06 (HS06) benchmark is executed. The HS06 benchmark is based on a subset of the industrial standard benchmark SPEC CPU 2006 [GS20]. The benchmark is executed to receive the HS06-score for each node. The Pilot reports the collected information including the score to the PanDA server and requests a suitable job. The PanDA server uses the HS06-score as a metric for the computing power of the nodes.
As the Pilot receives a job, it reads the job options and prepares the execution of the payload. The Pilot imports required dependencies and downloads the job input data. After validating the input data, Pilot executes the payload and uploads the output data of successfully finished jobs. The whole process is monitored by Pilot and collected metrics, such as downloaded volume, download duration, memory consumption, and execution time are sent to the corresponding monitoring system. Furthermore, detailed logs of the Pilot and the used software components are uploaded.
The Production System (ProdSys) is an abstraction layer for the PanDA system. A production team in ATLAS defines tasks in ProdSys, which are saved to ProdDB. ProdSys is responsible for resolving the tasks into jobs and submitting the jobs to PanDA.

A central aspect of the DDM project is the development and operation of the

Rucio [BB+19] software. Rucio is the successor of Don Quijote 2 (DQ2) [BC+08], the previously used data management tool for ATLAS. Since its development, Rucio has evolved as a modern scientific data management software, which is also being evaluated, adapted, and extended by a broad spectrum of experiments in the scientific community [BB+19].
All WLCG storage available to ATLAS is managed by Rucio. Thus, Rucio must allow cataloguing all ATLAS data and data structures. For this reason, Rucio employs three levels of data grouping:

- A *file* as the description of a single data object that comprises all the metadata, such as, file size, checksum, creation time.
- A *dataset* as a collection of files.
- A *container* as a collection of datasets or containers.

Each of those objects can be addressed by a Data Identifier (DID), which is unique across all three data groups.
A file is a purely logical unit. The data described by a file can exist multiple times as *replica*. To address an actual replica of a file, the information of the storage element is required. Rucio uses an extra layer of abstraction for storage elements, which are called Rucio Storage Element (RSE).
Another objective of Rucio is the management of data transfers. Rucio does not implement transfers itself, but employs the *transfer tool* interface. For ATLAS and the WLCG, this interface is implemented using the CERN File Transfer Service (FTS) [AS+14]. FTS is a service provided by the CERN IT department that allows scheduling and monitoring of data transfers in the WLCG.
The way Rucio creates transfers is by evaluating *replication rules*. Individuals or systems using Rucio do not request transfers directly. Instead, they add rules to Rucio that describe how data must be available. For example, a rule states that a specific dataset must be available on RSEs with certain attributes, such as, a specific country or all Tier 1 sites. Rucio resolves those rules, checks if the rules are met, and submits transfers to FTS when necessary.
Furthermore, replication rules indicate the requirement of replicas. Replication rules have an expiration time. Replica are not deleted as long as there is a rule that states that a specific replica is required. If all rules on a replica were expired, the replica can be deleted.
Beside transfers, there are *downloads* and *uploads*. The difference to transfers is that the source and destination data of a transfer are both registered in Rucio. A

Metric	RAW	AOD	DAOD
Volume tape	113 PB	63 PB	6 PB
Volume disk	2 PB	64 PB	103 PB
No. Events	$3.66 \cdot 10^{11}$	$3.63 \cdot 10^{11}$	$9.82 \cdot 10^{11}$

Table 2.1: Share of used tape and disk storage by data format for all ATLAS data as of July 2020.

download pulls data from the WLCG and the downloaded data are not managed by Rucio. An upload works the other way around; unmanaged data are placed in the WLCG and must be registered in Rucio. Moreover, downloads and uploads are not processed by a transfer tool. Typically, downloads and uploads are executed directly by a client machine.

RAW, AOD, and DAOD are the three ATLAS data formats that require the vast majority of storage space. Table 2.1 shows the volume distribution at tape- and disk storage and the number of stored collision events for the three formats. The first two rows give an impression of the usage of tape- and disk storage for the data. Typically, RAW data do not have persistent copy at disk storage but are solely stored on tape because they are required infrequently. Furthermore, RAW data cannot be reproduced, and thus they are archived by storing one copy of each RAW file at the Tier 0 and another copy at an arbitrary Tier 1.
AOD have different requirements than RAW data. AOD are requested more frequently to produce or reproduce DAOD. Moreover, AOD can be reproduced from the corresponding RAW data for detector data or from the RDO for simulated data. Since the reconstruction of AODs requires significant computational effort, AODs are typically saved with an additional copy at tape storage. Because AODs are used more frequently, there is an additional copy at disk storage of most AODs.
On the other hand, DAODs are the most frequently used data among the three formats and require the highest availability. DAODs can be reproduced from the corresponding AOD data. In addition, DAODs are accessed in random and concurrent access mode. For these reasons, DAODs are primarily stored on disk storage.

2.2 Commercial Clouds

This section starts by describing the basics of commercial cloud resources, including common terminology, the fundamental concept of cloud resources, and certain

characteristics of commercial clouds. Afterwards, details of Google as a commercial cloud provider are given. Finally, certain projects related to commercial cloud resources are summarised.

2.2.1 Overview

Cloud computing has remained a constant topic of research for ATLAS in the past decade [PM+14; TB+15]. A key aspect of cloud computing is the virtualisation of resources. Multiple Virtual Machines (VMs) are running on a single physical machine. The specification of a VM can be chosen based on the requirements. The VM starts from an image, which includes the operating system and the software stack. Beside computing resources, clouds typically provide storage resources. Large cloud providers also maintain their own network infrastructure among their data centres.

With the ability of acquiring resources on-demand to a seemingly unlimited extent, clouds provide a convenient method of resource provisioning. Two base types of clouds are distinguished based on the resource domain. *Private clouds* comprise resources only used by a single organisation, while *public clouds* can be used by an arbitrary number of organisations. In addition, there are hybrid models that combine these two approaches, e.g., by hosting storage in a private cloud and using CPU resources from a public cloud.

In general, three types of service model are differentiated in a layered model.

1. Software as a Service (SaaS) is the highest layer service type. This layer provides fully fledged software services, typically in the form of web applications.

2. Platform as a Service (PaaS) is the middle layer service type. This layer provides a platform to allow executing given applications or program code. The platform provides all required dependencies for the application.

3. Infrastructure as a Service (IaaS) is the lowest layer service type. This layer provides bare computing and storage resources.

Different cloud providers implement different cost models. Typically, computing resources are charged depending on the VM specifications and the time the VM was used. Storage is usually charged based on the volume per time, e.g., GiB/month. Larger providers offer various storage classes for data with different access characteristics. A common approach of charging network cost is to only consider the network egress. Ingress traffic is typically free.

The cloud providers that provide their own intercontinental network connections

typically charge the egress based on the destination location. In general, egress includes all traffic leaving a data centre of the cloud provider, i.e., traffic from the cloud provider to another network, such as the internet, and traffic from one data centre to another data centre within the same cloud. For larger providers, the cost of a VM or storage resource can vary, depending on the geographical location of the associated data centre.

Most research for cloud resources is related to VM provisioning and scheduling techniques. Other prominent research topics cover data security in public clouds or energy-aware algorithms for cloud providers. Storage and data management concerns are often neglected. From a data management perspective, three use-cases of commercial cloud resources can be considered for an ATLAS associated institution.

1. Cloud storage as permanent storage extension to ATLAS, e.g., acquiring cloud storage instead of increasing the on-premises resources.

2. Cloud storage bursting, i.e., in the case of peak storage requirements, cloud storage could be quickly allocated and used to cover this peak. As the demand for storage falls off, the cloud storage would be deallocated accordingly.

3. Network bursting, i.e., in the case of high network load or if additional bandwidth is required, data can be sent through the cloud network.

2.2.2 Google Cloud Platform

Google offers a large variety of online services based on Googles computing infrastructure. These services build the Google Cloud Platform (GCP) [Goo21]. For this thesis, Google was chosen as resource provider because several collaborative projects between ATLAS and Google were initiated in parallel. This provided the opportunity to exchange information with experts from Google and acquire real world data from dedicated tests.

The GCP contains different products related to data storage, each of which meet different requirements. For example, the Persistent Disk product is implemented by a block storage and is designed to be used together with computing services to store disk images for virtual machines. Alternatively, the Cloud Storage product is implemented by an object storage and can be used as an IaaS. The Cloud Storage product appeared to deliver the best compatibility considering the requirements for ATLAS in terms of scalability, data formats, and independence of other GCP products.

In this thesis, Google Cloud Storage (GCS) denotes storage related to the Cloud

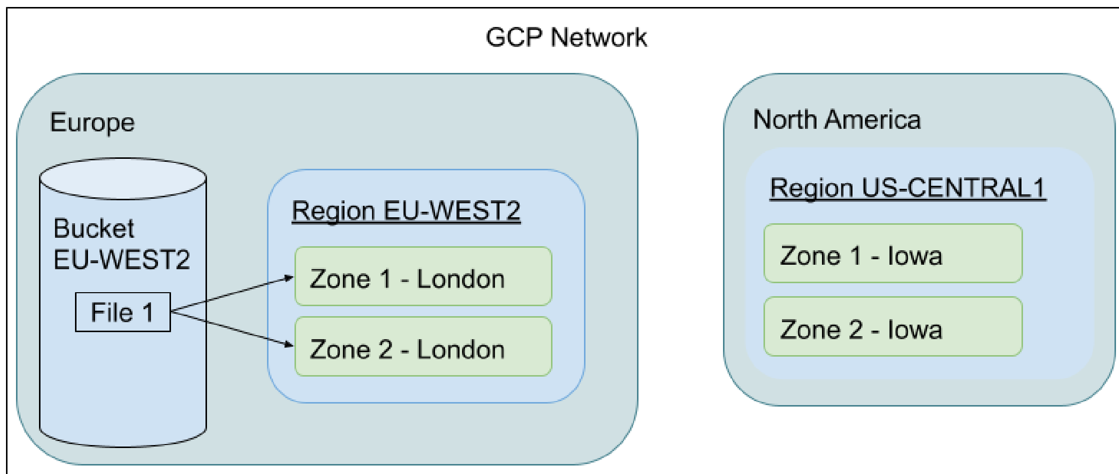

Figure 2.5: Schematic of the GCS architecture. Illustrated are two continental locations. Each continent can provide multiple regions. The example shows the region EU-WEST2 in Europe and US-CENTRAL1 in North America. Typically, each region has multiple zones, which provide a certain degree of redundancy. The user can create arbitrary buckets associated to a region. Files in a bucket are transparently stored in at least two zones.

Storage product. Furthermore, Google Compute Engine (GCE) denotes any services that provide computing capabilities, such as services that provide functionality to operate virtual machines.

As object storage, GCS allows storing immutable objects containing data in arbitrary formats. The objects are stored and organised into logical containers called *buckets*. Primarily, buckets are described by a globally unique identifier, a specification of the geographic location, and a storage class. The identifier can be chosen arbitrarily as long as it is globally unique. The geographic location of a bucket within GCS is implicitly specified by associating the bucket to a *region* as illustrated in Figure 2.5. A region is a logical abstraction for a specific geographic location and consists of multiple *zones*. Which regions are useable for GCS and which zones compose them is defined by Google. Typically, a zone corresponds to a specific data centre, which provides the hardware resources.
All data uploaded to a GCS region is stored redundantly in at least two zones. The geographic location of each region can be found in the GCS documentation. For example, the region `EU-WEST2` with zones located in London or `EU-WEST9` with zones located in Paris [Goo21].
However, buckets in GCS are not limited to one region. Buckets have to be assigned to at least one region, but they can also be set to *dual-region* or *multi-region*

mode. In dual-region mode, two regions within the same continent can be specified for a bucket. Using dual-regions, data insertion or removal from the bucket will transparently be replicated to both regions. In multi-region mode, the data will be transparently replicated across multiple regions of a given continent.

In dual-region and multi-region mode, special region names can be used that indicate which geographic locations should be considered for the corresponding bucket. For example, the dual-region `EUR4` for Finland and Netherlands or the multi-region `EU` for potentially all GCS data centres in the European Union [Goo21]. Using such a special region is typically more cost-efficient than individually selecting arbitrary regions.

The primary use case for the dual-region and multi-region mode is for applications that require the best possible availability and latency to access data. A typical example are video streaming platforms, which often require the requested data to be available close to the receiver to provide a low latency access with a proper data transfer rate. For ATLAS, the availability and low latency access are not the main priority. Whether the data can be accessed in less than a second or in several seconds does not significantly impact the ATLAS workflows, typically.

The last part required for creating a bucket is to decide which storage class to use. Google provides four different storage classes: standard storage, nearline storage, coldline storage, and archival storage. The main differences between the storage classes are the minimum storage duration and the costs.

Standard storage is the most flexible storage class. The standard storage class has no constraints on the storage duration of data, has the best monthly availability, and is the most cost-efficient storage class for frequent data reads and writes. However, standard storage has the most expensive cost per volume ratio.

Nearline storage is a storage class for less frequently accessed data. The nearline storage class expects a minimum storage duration of 30 days. Removing data earlier than 30 days would introduce additional cost. In addition, accessing data on nearline storage is approximately 2.5 times as expensive as using standard storage. On the other hand, the cost per volume ratio is approximately half as expensive as for standard storage.

Coldline storage and archival storage have even further increased minium storage durations. Coldline storage is designed for data with a minimum storage duration of 90 days, while archival storage expects data to be stored for at least 365 days. Consequently, the cost for data reads and writes increases and the cost per volume ratio decreases for coldline and archival storage.

The implementation of GCS in GACS focused on standard, regional storage. Multi-

regional storage was not considered beneficial for the use cases of ATLAS and is infeasible to implement since the replication across multiple regions is done completely transparently to the user. Storage classes other than the standard storage have not been used in this thesis, but are already implemented in GACS.

As a commercial cloud provider, most of the services offered in GCP come at a certain cost. Related to the adoption of GCS, the costs can be described in three main categories. First, the storage cost that depend on the volume of data stored in the cloud. Second, the network cost depending on the volume of data transferred out of the cloud or within the cloud. Last, the operation cost that depend on how many storage operations such as download, upload, delete, or the change of metadata, were executed.

The order of the costs depends on the usage. However, typically network cost can be considered the most expensive, while the operation cost are the least expensive. For example, storing 1 GiB in a single region, standard storage bucket in Frankfurt would cost 0.023 USD per month, as of writing this thesis. Downloading 1 GiB to a worldwide location, excluding Asia and Australia, would cost 0.12 USD. The operational cost for requesting the data download would be 0.004 USD per 10k requests.

GCP provides a so-called SKU ID for every type of operation within the GCP. For example, a transfer from North America to Europe is identified by a certain SKU ID. All SKU IDs and information about the operation they describe can be downloaded via the GCP API. The information also include the pricing details.

2.2.3 Related Cloud Projects

Various projects in the past already investigated the potential benefit of commercial cloud storage. In the scope of the Data Ocean project [Col19], ATLAS and Google defined three use cases for which GCP could be advantageous. The considered workflow was the physics analysis. The general goals were to allow ATLAS to explore and evaluate new models, including commercial cloud resources. Moreover, the project allowed ATLAS to identify technical challenges when including GCP resources into ATLAS. Google profited from the project by having the opportunity to observe the usage of their resources for scientific use cases. This allows Google to collect experience and improve their offers in the scientific market.

A large part of the Data Ocean project was the implementation of GCP features into the ATLAS components. One problem was the authentication at GCP for data transfers. This led to not being able to test the full bandwidth between the WLCG and GCP. For this reason, an extension to the FTS was developed, but because of

a late deployment time, it was not possible to test the transfers at full bandwidth for the Data Ocean project. Another problem was introduced when executing the analysis using direct-IO, i.e., the analysis software reads the input data directly from the GCS without storing it locally. This lead to data corruption, and therefore it could not be used.
In summary, the project allowed identifying the technical challenges of integrating GCP into ATLAS. Moreover, it was possible to extend the existing ATLAS components to be prepared for future projects using GCP.

Another project evaluated options of using GCP resources to extend the on-premises resources of the Tokyo regional analysis centre, which is an ATLAS Tier-2 site [Kan20]. This project considered three approaches to integrate GCP into the Tokyo site, full on-premises, full cloud, and a hybrid approach. The project considered the cost of acquiring on-premises resources in comparison to using GCP resources.
For the full on-premises approach, the acquisition costs were estimated to a total of 200 thousand USD/month for three years. Excluded are variable costs, such as, power cost, maintenance cost, and other infrastructure cost. On the other hand, the full cloud approach was estimated to 210 thousand USD/month for the computing resources only. In addition, the cloud storage and network cost estimation is 270 thousand USD/month. This results in a total estimate of 480 thousand USD/month. The third approach uses a hybrid system. The hybrid system consists of GCP computing resources and on-premises storage resources. This approach avoids the large cloud storage cost. However, the data extraction from the GCP computing resource introduces additional network cost. The estimation of the total cost of the hybrid approach is 270 thousand USD/month.

The VR Observatory went through a similar evaluation process and decided to start using cloud resources from Google as interim data facility in 2023 [K20].

2.3 Related Models

This section elaborates on the Data Carousel model and Hot/Cold Storage model, which are later combined to create the HCDC model and evaluated throughout this thesis. The models are considered for usage with the data types, workflow types, and resource types described in the previous sections.

2.3.1 Data Carousel model

The Data Carousel model [BB+20] aims at reducing the usage of performance-oriented storage like disk storage and prefers the usage of low-cost storage like tape storage. This is particularly applicable for data that are accessed rather infrequently or for data with an easily-predicted access pattern. As mentioned in Section 2.1.3, these conditions apply to RAW- and AOD data. Ideally, these data are only requested once per reprocessing campaign.
A data derivation campaign starts with the definition of the workload. In ATLAS this is done by a production team creating tasks in ProdSys. ProdSys coordinates with the DDM system [ATL17b] and the workflow management system [ATL11] the transfer of the required data from tape storage to disk storage and the start of the processing of those data.
In the best case scenario, all input data of derivation production workflows would be stored solely on the low-cost storage. As mentioned in Section 1.2, the two main concerns about the Data Carousel model are ensuring sufficient free disk storage for processing campaigns and the tape performance.
Currently, the Data Carousel model uses a *sliding window* to address the concern about having sufficient free disk storage. The input data are solely stored on tape. When the derivation campaign is defined and the data to process are determined, a sliding window is created. The sliding window has a specific size, e.g., the size of a given percentage of the total input data. Data that are required for processing must allocate space in the sliding window. After a successful allocation, the data can be transferred from tape to disk storage. When sufficient data were transferred, the workload can start to process the data. The data are downloaded from the disk storage to the worker nodes, where they are processed by the derivation software. When the processing of the data is complete, the corresponding data are deleted from the disk storage and deallocated in the sliding window. Using this approach, only disk storage equal to the sliding window size is required at any one time. [BB+20]
The possible size of the sliding window is limited by the following parameters:

- available storage for the sliding window
- volume of input data to process
- throughput of the tape storage
- time between start of transfers and start of workloads
- available computing resources

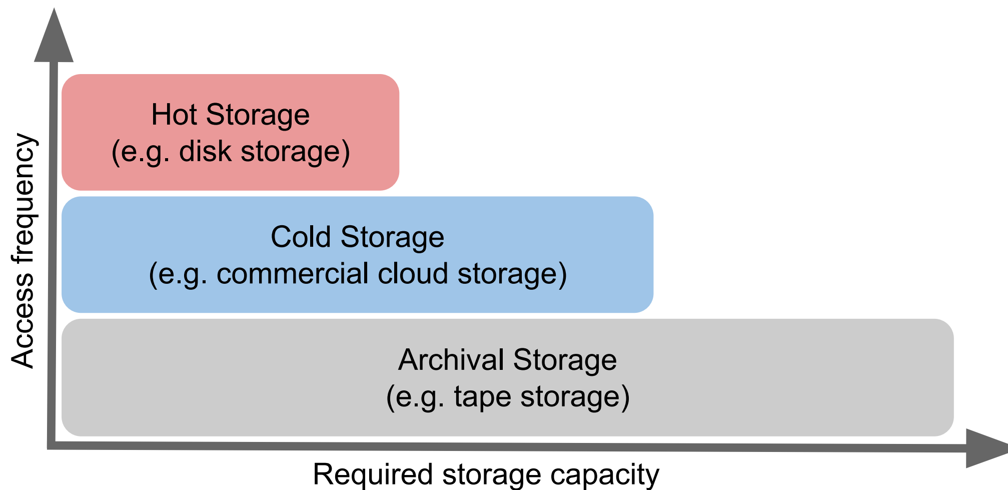

Figure 2.6: Hot/Cold storage model. One replica of each file is stored on archival storage. Files are migrated between cold and hot storage based on a popularity metric.

The minimal and maximal size of the sliding window depends on the available storage and the volume of the required input data. Typically, the volume of input data is larger than the storage available for the sliding window. Thus, the temporary storage available for processing is the limit rather than the volume of input data. The window size must be sufficiently large to hold all the input data for all currently running jobs.
Another potential limitation of the size of the sliding window is given by the performance from the tape storage to the disk storage and the time it takes to process the data. For example, if the performance from the tape- to the disk storage is the bottleneck, a very small sliding window size would be sufficient. The reason is that a large window could not be filled up.

2.3.2 Hot/Cold Storage model

As shown in Figure 2.6 the Hot/Cold Storage model divides the storage into hot storage, cold storage, and archival storage. The main dimensions in which the requirements to the storage categories differ are the storage capacity and a popularity metric such as the access frequency of data estimated by the number of times a file was used.
Hot storage requires a small capacity to store only the most frequently accessed data. Optimally, hot storage should be located in close geographical distance to the computing resources to allow a high bandwidth and low access latency connection. Regarding the QoS properties, the medium implementing hot storage must provide

good performance in terms of throughput and access latency, especially in concurrent and random access mode.
Cold storage requires a larger storage capacity than hot storage. There are two use cases for cold storage. First, it can be used as a temporary buffer when the hot storage is full. In this case, cold storage accepts data from archival storage that are required or are likely to be required based on the popularity metric. Second, cold storage can be used as a cache between an archival and a hot storage. In this case, the cold storage caches the data from the hot storage that are no longer required at the hot storage but are likely to be required again in the short term.
Archival storage requires the largest capacity. The QoS properties of archival storage typically describe a higher access latency and significantly lower performance for concurrent and random access mode. The Hot/Cold Storage model assumes that at least one replica of each file is kept at the archival storage.
Different approaches are possible to use the storage categories together. Typically, data on hot storage are replaced very frequently. The data are preferably transferred from cold storage to hot storage. If the required data are not available on a cold storage, they are transferred from an archival storage.
In the implemented variation of the Hot/Cold Storage model, the required data that are not available on cold storage are directly transferred from the archival to the hot storage. Prior to the deletion of the data from the hot storage, the data are replicated to a cold storage. Alternatively, the required data could firstly be transferred from the archival to the cold storage and then be transferred to the hot storage. This would result in less delay for the deletion because the data do not have to be transferred to the cold storage first. However, it would increase the waiting time for the required data because of the initial transfer to the cold storage.
Another point is how the allocation and deallocation of cold storage are managed. Ensuring the existence of hot storage data on cold storage prior to their deletion, requires either a sufficient cold storage capacity or a deletion strategy of cold storage data. Another approach would be to set a threshold based on the popularity metric and only allow transferring data to the cold storage that have a certain popularity. This threshold could be used to improve the hit/miss ratio when using the cold storage as cache.

2.4 Related Simulation Tools

As mentioned earlier, the GACS tool was developed as part of this thesis to simulate storage resource acquisition and file transfers among grid and cloud resources. The

following section gives an overview of the most relevant related simulation toolkits developed over the past two decades. Most of these existing toolkits are discontinued, rarely updated, or developed for rather specific scenarios.

2.4.1 Requirements

The most relevant features of a simulation software that are required for the HCDC model are:

- The modelling of grid storage resources is required. The data centres are considered as black boxes. Thus, the storage systems are described by high-level properties, such as storage capacity or access latency.
- The possibility to model commercial cloud storage is required. The underlying cost model should be customisable.
- Data management functionality is required, such as representation of storage topologies, replica bookkeeping, as well as creation and deletion of files and replicas.
- Modelling network topologies must be possible based on high-level metrics, such as shared bandwidth or transfer duration.
- File transfer functionality is required. The transfer mechanic must use the configured network topologies and interact with the data management to create new replicas and keep track of incomplete replicas
- The simulated model is primarily defined by the logic that implements the data generation, the data replication, and the data deletion.

The models are created from a high-level data management perspective. Hence, features for a detailed simulation of CPU resources, e.g., using Million Instructions Per Second (MIPS) to estimated job run times, are not required. Neither are detailed network elements required, such as multiplexing and routing elements. The level of detail about the available information is given by metrics, such as number of total jobs finished, average job run time, average input- and output volume per job, hourly throughput between storage elements. There are no precise information of the hardware resources or the network topologies. The ATLAS sites, storage elements, and cloud resources are considered as black-boxes.
Network simulation models are typically differentiated between packet-level and flow-level [VS+13]. Packet-level network models priories accuracy over scalability

by simulating each packet and the full network protocol in every detail. Flow-level simulations simplify the network model by simulating the different data flows in the network using parameters, such as bandwidth and number of active transfers. The requirements for GACS suggest employing a flow-based network model implementation.

2.4.2 Considered Simulation Tools

Four simulation tools were found that were considered potentially relevant for this thesis, and thus were investigated in more detail.

GridSim

The GridSim toolkit, presented in 2002 by Buyya and Murshed [BM02], is a software specifically designed for simulating models related to grid computing structures. The development was motivated by the requirement to develop and evaluate grid algorithms without having access to real test beds. The latest release was in 2010 with version GridSim 5.2 in their subversion repository on SourceForge [Sou21].
The toolkit is based on a multi-layer interface architecture. The two lowest layers describe interfaces for the basic programming- and simulation functionality. They are implemented by the Java Virtual Machine and SimJava [HM98], a generic discrete-event simulation framework. The next layer contains interfaces representing the various grid resources and services and is the core of the GridSim toolkit. The next layer describes the algorithms that assign the user requests and applications to the resources provided by the lower levels. Finally, the top layer comprises the applications and scenario configurations of the user.
The main features of the GridSim comprise modelling of grid resources by creating various objects, such as resource descriptions, physical machine descriptions, application descriptions, or network links. Computing performance is specified in form of MIPS. Network topologies are defined by creating objects for endpoints, links, router, packets, and packet scheduler [SP+07]. These objects use properties such as, bandwidth, Maximum Transmission Unit (MTU), or latency to calculate travel time of packets in the simulation.
Originally, GridSim did not contain storage and data management features. Sulistio, Cibej, et al. [SC+08] proposed a data grid extension, which was adopted in GridSim. They describe and implement three components that are essential to simulate data management in a grid environment: storage resources, replica catalogues, and a resource manager. They implemented an interface for storage resources with default

implementations for disk- and tape-storage. Two approaches of replica catalogues are implemented, a hierarchical and a centralised catalogue. The resource manager represents the system that combines the other components. It allows creating the storage topology, receives data requests, resolves the requests, and manages the resulting data transfers.

SimGrid

The SimGrid toolkit, presented in 2001 by Casanova [Cas01], started as a simulation software for heterogeneous resources, such as in grid computing. It was motivated by the requirement of tools that allow developing and evaluating scheduling algorithms more rapidly and accurately compared to other simulation toolkits being researched at that time. As of writing this thesis, SimGrid is still being actively developed. Numerous researchers used SimGrid for a variety of topics. The code is managed with git and publicly available on Framagit [Fra21].
Other than GridSim, SimGrid is implemented in C++ and includes several language bindings. SimGrid provides multiple APIs for the user to set up a simulation scenario. The MSG API was introduced in SimGrid v2 [CLQ08]. It allows the user to control the simulation flow by programming concurrent simulation processes and letting them communicate with each other via messages. The SimGrid documentation states that the MSG API is in the process of being replaced by the new S4U API. The SMPI API allows the user to describe and run the simulation as MPI programs. The last user API is the SimDAG API, which allows the user to describe the simulation as an abstract task graph [CG+14].
The feature list of SimGrid covers numerous requirements for popular research topics. It allows simulating applications on various platforms, such as grid-, cloud-, or hpc-resources. It is possible to set up individual hosts or pre-defined cluster topologies. Objects representing VMs are implemented, as well as VM migration techniques. Barisits [Bar17] mentions that the network models are the most noticeable contribution of SimGrid. The used network model is TCP flow-based and frequently re-validated and improved [VS+13]. Using the S4U interface, the network topology is represented by hosts placed in net zones, which use routing configurations to send the data through the corresponding network links.
Similar to GridSim, SimGrid did not contain storage and data management capabilities initially. These functionalities were added by an extension proposed by Lebre, Legrand, et al. [LL+15]. The extension added components to simulate files, which represent physical data. Files can be stored on a disk storage and accessed through a file system. The file system assumes sequential access and simulates latencies and

I/O durations for file accesses. A disk resource can be attached to any type of host.

CloudSim

The CloudSim toolkit, presented in 2009 by R., Ranjan, and Calheiros [RRC09], is a software focused on simulating cloud resources. The presentation of CloudSim was motivated by the increasing demand for research related to resource allocation strategies and application scheduling algorithms in cloud computing. As of writing this thesis, the last full release was 2016, although there is a pre-release version from 2019. The code is managed using git and publicly available on Github [Git21].
CloudSim uses an architecture similar to the layered architecture of GridSim. Initially, the architecture was also based on SimJava. Since this led to suffering from the same scalability issues and in order to support advanced functionality, the SimJava component was removed from CloudSim [CR+11; OP+11].
The features of CloudSim address some of the most popular research topics related to clouds. CloudSim allows modelling cloud resources including computing, storage, and network. It is possible to specify individual CPUs of physical machines, which run VMs. Furthermore, VM images can be specified to allow simulating application containerisation. Various models are implemented to configure network topologies within and among data centres. User-customisable policies are used to define the resource provisioning. In addition, energy-aware computing models are implemented.
CloudSim did not provide data management functionality in the beginning. Long and Zhao [LZ12] proposed an extension to CloudSim based on the conepts of the GridSim extension [SC+08] adding storage- and data management functionality to the software. The extension adds replica capabilities by defining *master files* and allowing additional *replica files*. Master files are pinned, which prevents them from being deleted before their copies are deleted. Furthermore, the extension to CloudSim implements a *name node* that serves as a catalogue, resolving file identifiers into location information. The extension also implements simulation of a distributed file system using a block manager.

GroudSim

The GroudSim simulation framework, presented in 2011 by Ostermann, Plankensteiner, et al. [OP+11], is a software specially developed to simulate models including both grid and cloud resources. The development of GroudSim was motivated by the requirement for a software able to simulate grid and cloud resources and the low scalability of existing simulations. The last updates were committed in 2010 to their subversion repository hosted on assembla [Ass21].

GroudSim is based on events scheduled at discrete time points. The simulation engine stores an integer value representing the clock. Events are invoked based on the clock. The user creates events to control the simulation flow. For example, a job submit event that is configured with properties describing a simulated job, such as the destination that runs the job, required CPU performance in MIPS, and the state of the job. When this event is invoked, it acquires resources based on user defined policies, and creates a job queued event. The handling of the job queued event and the following events can be implemented by the user.
GroudSim provides basic file transfer features. In GroudSim a transfer consists of a static number of events. Typically, one event on transfer submission, one on transfer activation, and one event on transfer completion. There are a few more event types, e.g., transfer cancellation or failure. On invocation of an activation event, the transfer duration is calculated based on the file size and the bandwidth shared among other active transfers. The transfer completion event is scheduled based on the calculated transfer duration. When a transfer- activation or completion event changes the number of active transfers, all other transfer completion events are rescheduled. Because of the static number of events, the amount of consumed storage at the destination and the consumed traffic on the network link are updated once in the transfer completion event.

2.4.3 Discussion

In the following, the described simulation tools will be compared with consideration of the software requirements mentioned earlier. Furthermore, examples are given that show the relation between the scalability and the accuracy of the simulation tools.

Feature Comparison

Table 2.2 shows the available and unavailable required features for each investigated simulation toolkit. Grid storage means whether storage without an underlying cost model can be simulated, while cloud storage requires the possibility to add such a cost model. Furthermore, it should be possible to store the lifetime of data including automatic deletion functionality. Managed replication means the creation and transferring of a new replica and the automatic updating of the catalogue.
The storage and data management extension added disk- and tape storage resources to GridSim. The resources are unified by an interface definition, which could be used to implement additional cloud storage resources including a cost model. The net-

Feature	GridSim	SimGrid	CloudSim	GroudSim
Grid storage	✔		✔	
Cloud storage	✔		✔	
Bandwidth based network		✔		✔
Duration based network		✔		✔
Files and Replicas	✔		✔	
Data lifetime				
Data catalogue	✔		✔	
Managed replication	✔			

Table 2.2: Most relevant features required for GACS and the availability in the related simulation toolkits. A check indicates the available features.

work functionality on the other hand is based on packet transfers using the MTU metric. This can not be used to implement transfers based on shared bandwidth or average transfer duration. For this reason, network capabilities are considered unavailable.

As already mentioned, SimGrid provides functionality to simulate disk storage resources. Each disk storage is represented in a separate object. Multiple disks can be attached to a host or a VM. However, these resources are meant to represent local compute node storage and not the large scale distributed grid and cloud storage resources that are required. For this reason, storage features are considered unavailable.

SimGrid provides functionality to create files on disk storage and to copy these files. However, it does not logically associate the two copies with each other and does not ensure read only access to copies. For this reason, the files and replicas feature is considered unavailable.

As mentioned before, the storage- and data management extension to CloudSim is based on the extension to GridSim. For this reason, the availability of required features match between both simulations. A difference is the managed replication. Although, it is possible to create replicas, there is no transfer functionality that automatically transfers data from one storage element to another.

GroudSim does not allow modelling distributed data structures with files and replicas and does not provide functionality to model distributed storage resources. Shared

bandwidth transfers are implemented by default and described by multiple transfer volumes sharing the bandwidth of a network link. Duration based transfers can be used by customising the scheduling of the transfer completion event.

Despite SimGrid having the fewest required features of the compared tools, it would be the most appropriate option for an extension of missing features and use for this thesis. This is due to it being the most actively developed tool of the considered software. However, none of the tools allows creating models from the data management perspective as required.

Scalability and Accuracy

Casanova, Giersch, et al. [CG+14] mentions two key concerns of a simulation software, which are accuracy and scalability. Throughout their work, they describe several cases when these two key concerns develop counter directional. Furthermore, they state that only few simulation toolkits implement storage resource capabilities. One reason for this is the small demand for those features. The other reason is that it can be very challenging to develop accurate storage resource models.

A fine-grained discrete-event simulation would be an approach resulting in an accurate simulation [CG+14]. This simulation would use events, such as address resolutions of storage controllers in sectors and blocks, seek time of the disk drive, and caching and buffer effects, to simulate the storage system. This would result in an increased accuracy, but also in limiting the scalability.

The existing simulations allow or even force setting up the simulated resources in a high level of detail. This often delivers a more intuitively understandable simulation flow and a potentially much better accuracy. On the other hand, it negatively impacts the scalability of the simulation and requires an extremely precise description of the simulated resources.

For example, the aforementioned simulation tools have large object representations for each host, VM, and job. SimGrid also requires a detailed description of every single disk drive in a computing system. Some implemented network models create object representations for each individually transferred packet. The simulated storage systems simulate each read and write of a data block. Considering the objectives of GACS and this thesis of evaluating data management models for grid and cloud resources, these simulation tools provide an unnecessary level of detail in the wrong place.

The scalability limitation introduced by too detailed resource descriptions can be observed in the complex simulation scenario in the proposed data grid extension of GridSim [SC+08]. They had to limit the simulated computing resources because

the simulation required more than the available 2 GB of memory. They simulated 11 computing centres with a total of 279 compute nodes.
GACS is supposed to simulate models from a data management perspective, i.e., the extreme level of detail of the mentioned simulations is obstructive. There are two main reasons for this. First, it is infeasible to collect the highly detailed information of the modelled systems, especially for commercial cloud providers, which rarely publish details about their system internals. Second, the scalability is favoured over the accuracy. Hence, storage and network components are considered from a higher-level perspective with simple simulation attributes, such as fixed or randomly distributed access latency, used storage volume, available storage volume, or used traffic.

Furthermore, there are a few concerns about the implementation of the related simulation tools. Casanova, Giersch, et al. [CG+14] states that GridSim, CloudSim, and SimGrid implement simplistic models to simulate data access times based on fixed seek times and data transfer rates and that these models are neglecting important storage effects. Moreover, the scalability of GridSim is questioned because each simulated process is executed in its own thread, leading to an increasing number of context switches.
In addition, there are two potential issues with the implementation of transfers in GroudSim. First, the fact that GroudSim updates the amount of used storage and used traffic only at the end of a transfer, could lead to inaccuracies for streamed transfers. The state of the storage becomes the more inaccurate, the larger the data and the slower the transfer is. This is because the used storage and used traffic values are only correct at the beginning and at the end of a transfer. Depending on the billing intervals of cloud providers, this could accumulate to significant errors in the cost calculation.
Second, the rescheduling of all completion events of a network link can possibly affect the scalability of the simulation. The method that reschedules all completion events when the shared bandwidth changes, removes each event from the event schedule and adds a new, updated completion event. The event schedule is implemented using Java's `PriorityBlockingQueue`. A typical implementation of a priority queue is using a min-heap. A min-heap requires linear time complexity to find an arbitrary element and logarithmic time complexity to insert or remove a specific element. This matches the Java documentation, which states that removing an arbitrary object from `PriorityBlockingQueue`, has linear time complexity. For large network topolgies with a large number of transfers, the constant removal of events from the schedule could result in significant performance issues.

3 Simulation Tool Architecture

The development of the HCDC model, which will be described in more detail later, resulted in certain requirements for a simulation software. The requirements were already outlined in Section 2.4.1. This chapter starts by elaborating the requirements in the context of a software architecture. In addition, some typical simulation toolkit characteristics are described and which of these characteristics apply to GACS. Afterwards, it is explained how the architecture was developed to comply with the requirements. The chapter ends by discussing certain design decisions of the architecture.

3.1 Overview

This section explains the motivation of developing GACS. The first part of the section lists the detailed requirements of the architecture. This allows comprehending the design decisions of the architecture in the next section. The second part explains various simulation types.

As discussed in Section 2.4.3, there are no existing simulation tools satisfying the requirements for the research goals of this thesis. For this reason, the GACS toolkit was developed. In general, GACS can be used to analyse models and scenarios combining grid and commercial cloud resources from a data management perspective. Key features are the modelling of storage and network resources and their usage by simulating transfers.
A simulation software provides additional advantages. Especially, in the context of commercial cloud resources, a simulation allows the evaluation of models, without considerations about the cost of a real test bed. Furthermore, the implementation of a model in a simulation tool can yield new ideas, findings, or problems before use in a test or production system. In addition, a simulation typically allows the quick exchange of algorithms or parameters and observing the change of the model's behaviour.

3.1.1 Architecture Requirements

To create storage models based on ATLAS data and workflows, the architecture of GACS must allow creating models similar to the infrastructure used by ADC. In the following, the requirements defined in Section 2.4.1 are elaborated and the specific requirements for the architecture are described.
The first requirement is the data management functionality. This includes the possibility of creating storage resources. As in ADC, the architecture of GACS must allow creating an arbitrary number of storage elements. Each storage element is associated to exactly one site. Sites can own an arbitrary number of storage elements. Details about hardware and software solutions of the different data centres are not available. Thus, it must be possible to describe sites and storage elements by high level metrics, such as storage capacity or access latency.
Furthermore, the architecture must allow creating, accessing, and deleting data objects. Similar to the ADC environment, there must be a catalogue of all files. As in the context of ADC, in the context of GACS a simulated file is only a logical description of a physical file, while a replica is a physical file occupying storage space at a storage element. Each file can have one replica at each storage element. Replicas can be accessed per file or per storage element.
A key aspect of data management is the replication of data using transfers. To allow the simulation of transfers, it must be possible to model network resources. From the data management perspective, there are typically only high-level information of the network resources available. An architecture is required that allows modelling network resources based on directed point to point connections between pairs of two storage endpoints. Detailed information about network packet routing paths should not be required. Thus, the architecture accepts flow based network models. The most relevant parameters of a network connection are the bandwidth and the maximum number of active transfers. The primary metric of the network flow is the traffic, i.e., the transferred bytes.
Given a network model, the architecture must include functionality to create and manage transfers. It should be possible to simulate transfers based on transfer duration and shared bandwidth. Furthermore, the transfer management must interact with the data management to keep track from the creation of new replicas until their completion. In contrast to existing simulation software, the data structures representing the storage- and network connections must be able to keep track of the change over time of used storage and induced traffic, respectively. This is necessary to allow requesting the amount of used cloud resources at any point in time. Hence, it is not sufficient to consider only the start and the end of a transfer.

The logic defining the creation of transfers, including number of transfers, source data selection, destination storage selection, and transfer failure handling is the major part of a model. The architecture must describe proper interfaces to allow the user to define this logic.

The simulation architecture must enable the user to implement custom cloud provider resources. These resources must be usable in the same way as the existing grid resources. In addition to the grid resources, it must be possible to implement a custom cost model for storage and network resources of cloud resources.

3.1.2 Simulation Types

Numerous types of computer simulations can be found in the literature. A typical classification of simulation types is given by pairs of counter-directional characteristics. The most prominent are described in the following.

In a *discrete simulation* the state of the simulation progresses only at discrete points in time. The state stays exactly the same between two consecutive time points. On the other hand, a *continuous simulation* uses equations to describe the state of the simulation for a continuous time.

In a *deterministic simulation*, all inputs and algorithms must be well known and deterministic. No random variables or distributions are used. A *stochastic simulation* on the other hand, uses random values and distributions to approximate certain input parameters.

A *steady-state simulation* describes the values of a system at a specific fixed behaviour. This type of simulation does not account the changes of the behaviour. For example, a steady-state would be to consider only the maximum or average bandwidth of a network link without accounting any changes in the bandwidth. The opposite type is called *dynamic simulation*, which considers changes of the behaviour.

Given these attributes, GACS can be classified as discrete, stochastic, and dynamic simulation. A discrete simulation type was chosen because the reference data that will be used for parametrisation and validation are also based on discrete values. For example, the available real world data used for accuracy validation is based on discrete time points, e.g., the creation of files and transfers is stored as discrete time point, files expire at discrete time points, etc. In addition, continuous simulations are based on various equations, which can become very complex for large systems. This can result in very complex changes when the model has to be extended or modified.

The simulated and evaluated models described later in this thesis used a stochastic

approach to approximate certain input values, such as file size distribution. This was done because the real world data were significantly larger than the approximation. Furthermore, it was possible to well describe the inputs using random distributions. In general, a deterministic model can be implemented in GACS, if all input parameter are available.
GACS is required to be dynamic to accurately calculate the cloud cost. For example, the cost of allocated storage have to be calculated accounting every change over time in the amount allocated. Using only the maximum or average of allocated storage per month could result in significant inaccuracies.

3.2 Architecture Description

This section describes the details of the architecture of GACS. The section is structured in three parts. The first part explains the various modules of the architecture. The second part shows the organisation of the interfaces that are used for the communication among the modules. The last part describes common approaches and concepts for implementing models in GACS. These approaches were also used to evaluate the HCDC model. The section regularly refers to the detailed requirements stated in Section 3.1.1 and explains how the requirements are achieved by the given architecture.

GACS is a discrete-event simulation, i.e., the simulation is based on events that are scheduled to be invoked at discrete time points. A simulation event is not related to the collision events of the LHC that were explained earlier.
In the literature, a simulation event is often defined as a message that changes the simulation state [OP+11]. In GACS, it is possible that, based on the current state or random number generation, the state is not changed. For example, an event indicating the requirement of a replica at a specific data centre, but the replica already exists at this destination. Such an event would not necessarily change the simulation state.
In GACS, an event is the information that the simulation state must be evaluated and potentially be changed. The event contains the information about what parts have to be evaluated. The *simulation state* is provided by the data that represent all objects of the simulated model at a specific point in simulation time.
Ostermann, Plankensteiner, et al. [OP+11] mention two more architectural parts of a discrete-event simulation. The *time advance algorithm*, which increases the simulation time provided by the simulation clock and the *event scheduling algorithm*, which is responsible for the correct processing order of the events. In GACS, both

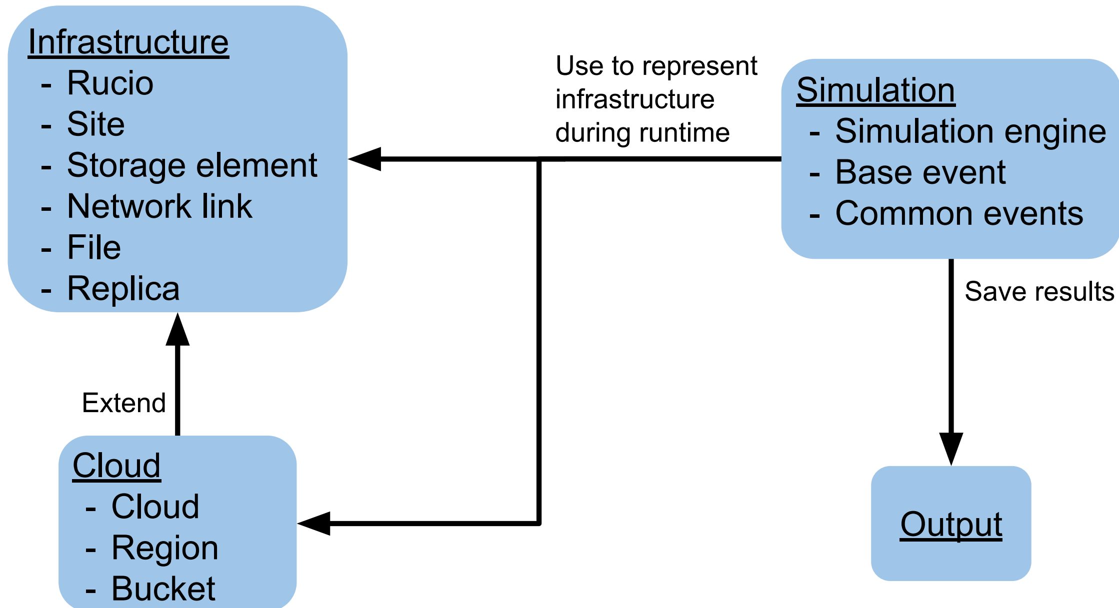

Figure 3.1: The four modules of the simulation. Simulation is the central module that manages the control flow and uses the other modules to create, manage, and save the simulation data. The bucket lists represent the most relevant components of the module.

algorithms operate in combination to build the so-called *event loop*.

3.2.1 Modules

As illustrated in Figure 3.1, the architecture of GACS comprises four modules, the infrastructure module, the cloud module, the simulation module, and the output module. Further, the Figure shows the most relevant components of each module and outlines the relation among the modules.
The modules can be categorised in active modules and passive modules. Active modules provide functionality that is executed directly by the simulation engine and makes decisions about changing the simulation state. For example, all events are active functionality. Active modules use the functionality of passive modules to change the simulation state and interact with the simulated model. Whereas, the primary purpose of passive modules is to provide data structures that can be used to represent a model. The functionality provided by passive modules is only used to modify the data and ensure that the data remains consistent. Functionality of passive modules is only executed by active modules.
The infrastructure module is a passive module that provides data types to represent the model that is simulated, e.g., storage endpoints, network links, files, and replicas. Furthermore, the module provides functionality to create, modify, or remove

instances of these data types. This way, it is possible to ensure that the data relations are consistent, e.g., ensure that each replica is associated with a file.

The cloud module is a passive module that extends the infrastructure module by functionality that is required to simulate commercial cloud resources. This approach of extending the infrastructure module allows implementing different cloud providers based on the same infrastructure components. Three main components are required to create a cloud implementation. A specific cloud implementation must provide the region and bucket components, which extend the site and storage element components of the infrastructure module, respectively. Further, each cloud implementation must implement its own cloud component. This component will be used by the simulation to create region and bucket instances of this specific cloud.

The simulation module is an active module that contains functionality to control and execute the simulation flow. The two main components of the simulation module are the simulation engine and the base event. The base event provides the basic data and functionality to represent an object that can be scheduled and contain a payload to be executed. The simulation engine provides the functionality for advancing and keeping track of the simulation time, executing the payload of scheduled events, and rescheduling events. Furthermore, the simulation module contains various common events that are already fully implemented. These events provide generically implemented payloads for common, recurring tasks.

The output module is a passive module that provides functionality to persistently store data generated by the simulation.

One aspect of the architecture is to ease the exchange of existing functionality and of extending GACS with new functionality. Therefore, the modules allow replacing most of their components or to be extended by custom implementations.

Infrastructure Module

The first requirement stated in Section 3.1.1 presumes data management functionality and the possibility to set up a distributed storage infrastructure. The infrastructure module comprises types and functionality to satisfy these requirements. The key types of the module and their quantity relations are illustrated in Figure 3.2. The functionality of the infrastructure module provides operations to create and access storage resources, network connections, and data. The structure of the infrastructure types is inspired by the ATLAS structure of WLCG resources.

The centre of the infrastructure module is the Rucio component. The name is adopted from the ATLAS data management software. There is only one Rucio component per simulation engine. The Rucio component is the access point for the

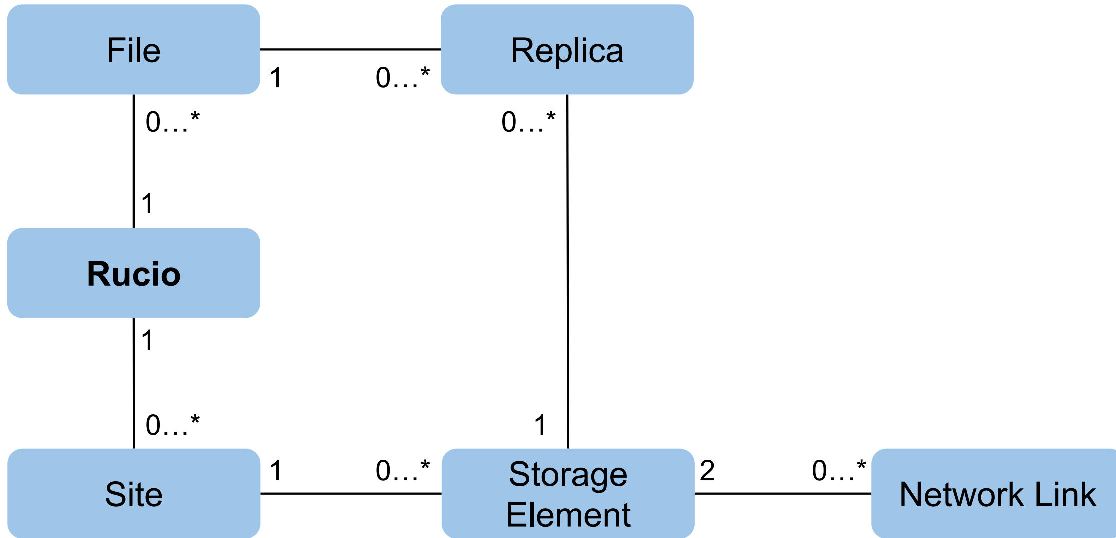

Figure 3.2: Illustrates the various components of the infrastructure module, how they reference each other, and the corresponding reference quantities. The centre of the module is the Rucio component, which typically exists only in a single instance per simulation. The Rucio component provides access to the sites and files, which themselves provide access to storage elements and replicas, respectively.

interaction with data and storage resources. Hence, it provides functionality to create, delete, and access infrastructure objects, such as sites or storage elements and data objects, such as files or replicas. In terms of the requirements from Section 3.1.1, the Rucio component fulfils the requirement of being able to represent the distributed data and cataloguing of distributed files.

As with WLCG resources, sites are components representing data centres. They are described by properties like name, location, and continental index. In addition, each site can be addressed using a unique ID number. Sites provide functionality to create and access their storage elements. As illustrated in Figure 3.2, each site is associated with exactly one Rucio instance, but the Rucio instance can contain numerous sites.

Storage elements are components that address a storage area of a site and describe its properties. Primary properties describing a storage element are a unique ID, the access latency, and the storage limit. The functionality provided by a storage element comprises creation of and access to network links, creation and deletion of replicas given a file, tracking of storage accesses, and storage space accounting. Different types of storage elements are possible, e.g., one storage element might allow multiple replicas of the same file, while other storage elements prevent creating multiple replicas of the same file.

Related to the storage space accounting, a storage element stores the values for used storage and allocated storage. When creating a replica, the size of the file associated to the replica must be allocated at the storage element. The amount of allocated storage can not exceed the storage limit. As the size of a new replica increases, the used storage value of the storage element increases as well.
Storage elements have the ownership of their replicas and their outgoing network link objects. Other simulation objects might have references to these objects. In other words, the source storage element of a network link possesses the ownership of this network link object. Additionally, Figure 3.2 shows the quantity limits for those references. A storage element is associated to exactly one site, while a site can contain multiple storage elements. Each replica is owned by exactly one storage element, but a storage element can contain numerous replicas.
A network link represents the connection between two storage elements and resolves the requirement of modelling network resources from Section 3.1.1. Since the network functionality of GACS requires a flow based model, the most important properties are the bandwidth and the maximum number of transfers allowed at the same time. A flag indicates whether the bandwidth has to be interpreted as shared bandwidth or as the unshared bandwidth for each transfer. Furthermore, statistics, such as the number of active, completed, failed transfers, and the amount of induced traffic are stored by a network link object. As shown in Figure 3.2, a network link is associated to exactly two storage elements, namely the source and destination storage element. However, a storage element can own numerous network links.
Files represent the description of data, which can be replicated or transferred. The most important properties are the unique file ID, the file size, the creation time, and the expiration time. In addition, a file object stores a reference to each of its replicas. The expiration time can be used for the lifetime based file deletion. A file object provides functionality to access its replicas and to extend the expiration time.
A replica represents the physically stored data of a file on a storage element. The most important properties are the unique replica ID, the currently used storage space of the replica, the creation time, and the expiration time. Furthermore, a replica object contains a reference to its file- and storage element object. Figure 3.2 illustrates the quantity relations of files and replicas. A file is associated to exactly one Rucio instance. A replica is associated to exactly one file and one storage element. However, a file can be linked to multiple replicas.
One of the most complex parts of the real Rucio is the processing of the replication rules. To ease the replication- and especially the deletion process, for GACS the

expiration time is directly applied to the files and replicas. When a file is expired and is going to be deleted, then all replicas will be deleted as well, irrespectively of their expiration time. On the other hand, if all replicas of a file are deleted, the file is also deleted. This mechanism can be deactivated to prevent automatic deletion and enable the user to maintain a custom deletion strategy.

Cloud Module

The cloud module is designed to be an extension of the infrastructure module. There are two relevant differences between the infrastructure- and the cloud module.

1. The cloud module can provide multiple, different implementations, i.e., provide different implementations to simulate different cloud providers.

2. The file creation, registration, and deletion is only handled by the infrastructure module. This means the cloud component does not provide additional data management functionality. Instead, the cloud component provides data types based on the infrastructure module components but extended by functionality required to model cloud resources.

Similar to the Rucio component of the infrastructure module, each cloud implementation must provide a cloud component that enables the user to create and access the cloud resources. Currently, GACS provides only a GCS implementation. In context of GCP, resources are pooled in *regions*, which refers to sites in terms of WLCG resources. Storage space is addressed with *buckets* instead of storage elements.
The GCS implementation in GACS extends the site- and storage element type of the infrastructure module by typical functionality of regions and buckets. Primarily, this includes storing the values of the cost model, tracking of operations inducing costs, and providing functionality to calculate the cost. The values of the cost model must be acquired by the Google billing API and can then directly be loaded by GACS.

Simulation Module

The simulation module provides the majority of active functionality of GACS. The module contains type definitions and functionality that are used to dictate the basic behaviour of the simulation and how the simulated resources are configured. The two primary subjects of the simulation module are events and the simulation engine. Each event requires data and functionality that enables the event to be scheduled during a simulation. To enable an event to be scheduled, the most important property is a scheduled time point. Additionally, GACS stores a name and a real time

counter for each event. The counter is used to accumulate the real time execution duration that the event required for processing. Storing the name and the execution duration, enables the simulation to print performance statistics. Another mandatory part of each event is the payload. The payload defines the functionality that is processed when an event is executed.
The simulation engine, as the second primary module subject, is responsible for three subsequent tasks:

1. Initialising and configuring data structures to represent the model to simulate. This is done using the functionality of the infrastructure and cloud modules. The primary steps are creating and configuring sites, storage elements, and network links.

2. Running the event loop and processing the schedule until a stop condition is met. This means executing all events of a certain time point and increasing the simulation clock accordingly.

3. Processing clean up routines and ensure that the results are stored persistently when the simulation stops.

The *base simulation engine* component describes a data and interface definition that are required to implement these tasks. Moreover, the base simulation engine implements a basic event loop and time advance algorithm. Furthermore, GACS provides the *default simulation engine*, which represents a default implementation of the defined interfaces. This eases the creation of a simulation model and allows loading configuration files. The functionality of both parts is generic and can be used for various simulation scenarios. Alternatively, a user can replace or extend the functionality by custom code.
The primary data attributes of every simulation engine are the simulation clock and the event schedule. Furthermore, the simulation engine provides the access to the Rucio component and all available clouds. GACS allows using exactly one grid component and any number of cloud components.
The behaviour of the default event loop and time advance algorithm in the base simulation engine are illustrated as an activity diagram in Figure 3.3. After the start of the event loop, there are three conditions that can stop a simulation again. First, when an event requests the simulation to stop. Second, when the clock reaches the configured maximum simulation time. Third, when the schedule contains no more events. If none of these conditions are given, the simulation engine extracts the next event from the schedule and sets the clock to the scheduled time of this event.

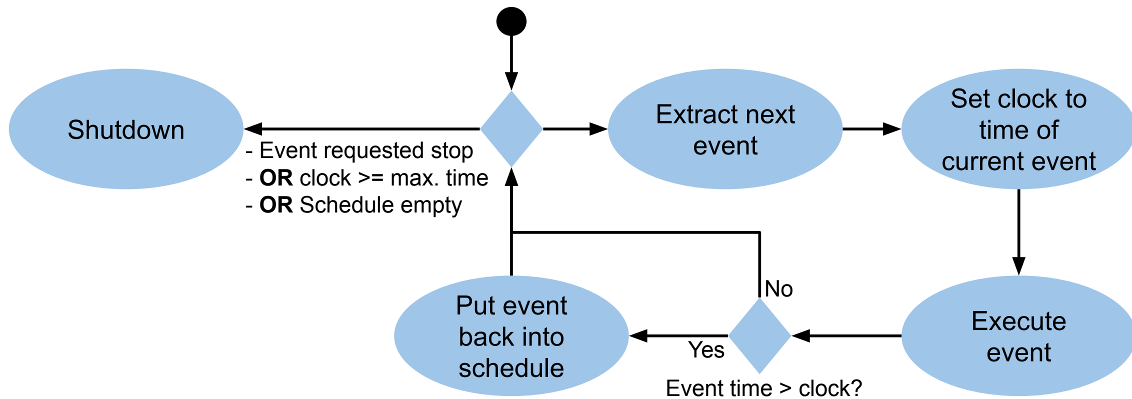

Figure 3.3: Illustrates the various steps of the event loop and the conditional transitioning.

Then, the payload of the event is executed. During the execution, the payload can increase the scheduled time of the event. After the execution, the simulation engine compares the scheduled time of the event with the clock. If the time was increased, the event is put into the schedule again.
The default implementation of the simulation engine allows setting up basic simulation scenarios using only configuration files. This is possible because of the common events and functionality implemented in the simulation module. These events will be explained in more detail in Section 3.2.3. The configuration files can be used to compose the implemented events and thus create a model. These common events also implement the requirements related to the transfer generation and management mentioned in Section 3.1.1. Moreover, the events allow the user to implement custom transfer generation routines.

Output Module

The output module provides functionality to persistently store the simulation results. The output module is used by the simulation module and can use one of several backends. GACS implements three backends by default.

- The *dummy backend* does not store any data. This is particularly useful for dry runs of a specific model. In this case, the user is only interested in checking the model implementation for any issues and not in the actual simulation results. As soon as the model runs flawlessly, the output backend can be changed.

- The *database SQLite backend* stores data persistently in a SQLite database [Hip21]. This requires the user to provide a configuration file describing the database model. It is also possible to run certain database queries at the

shutdown of a simulation, e.g., to create all table indexes at once.

- The *database psql backend* is similar to the database SQLite backend, but it is implemented using a PostgreSQL [Gro21] database and the psql API.

The output module is the only module that runs in a parallel thread by default. This supports the reduction of delays in a simulation due to I/O operations with the database. The output data are inserted into a buffer. The output module thread consumes the data in parallel and inserts them into the database.
The current implementation of the output module does not allow editing or deleting results that have been stored already. This has several reasons. First, it avoids inconsistencies when attempting to edit results that have not been stored by the I/O thread yet. Second, it improves performance in several cases, e.g., the results buffer can be handled unordered and the database indexes can be built once in a bunch at the end of a simulation. Editing stored results without existing indexes would most likely result in bad performance. The most important disadvantage of this approach is that the information of a result must be entirely known before the insertion, e.g., a replica can only be inserted after its deletion so that the deletion timestamp is known.

3.2.2 Interfaces

Figure 3.4 shows the various communication directions between the modules. To define a model, the architecture provides two types of interfaces. The first type is an interface based on configuration files, which is illustrated by the green box in the figure. Configuration files contain data only and can be used to define and configure compositions of events and simulated resources. The second type is an interface based on program code, which is illustrated by the yellow boxes. The program code interfaces must be used in order to implement custom functionality, e.g., in the form of events. Typically, both interfaces are used in combination, i.e., program code is used to define the behaviour of a model and parameters values are written in the configuration file.
In the architecture, configuration files serve as an additional layer between the user and the simulation engine. The configuration files allow changing the parameters without compiling code. In addition, they organise the parameters centralised and structured without functional code elements among the data. On the other hand, the configurations files are less flexible than the programmatic interface because the configuration files can use only existing functionality.
The configuration system allows combining and configuring existing functionality.

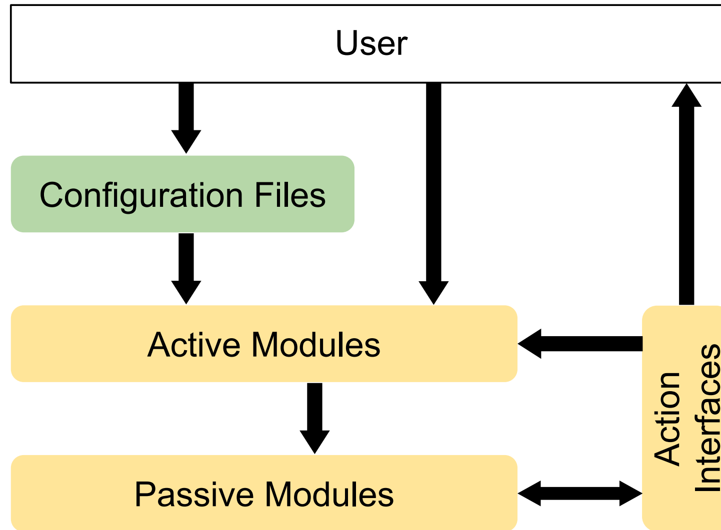

Figure 3.4: Interfaces that allow the user to set up models in the simulation and that allow the various parts of the simulation to communicate with each other. Interfacing with yellow boxes is done using program code and requires recompilation. Configuration files are formatted in JavaScript Object Notation (JSON) and can be modified without recompilation.

Typically, each simulation requires at least four configurable elements. First, a configuration for the infrastructure, i.e., what sites exist and what storage elements each site provides. Second, a configuration for the network links, i.e., in what directions the storage elements are connected and what are the corresponding bandwidth. Third, a configuration of the data generator events that specify either the initial files and replicas to create or a regular creation of new files. Last, the transfer configuration that describes how the transfers are generated and executed. Optionally, a configuration of the cloud module is possible allowing to define regions and buckets for a specific cloud.

The programmatic interfaces allow implementing custom events and customising the simulation engine. There are three programmatic parts interfacing with each other in the simulation. The user extends the active modules with code. The active modules implement the logic of the model, utilising the functionality provided by the passive modules. In addition to these two modules, there are action interfaces. Various operations of the passive modules invoke action interface operations. For example, pre- and post-file creation, file deletion, or replica creation invoke action interfaces. Passive modules implement certain action interface operations to keep their data consistent. For example, a file object receives replica creation and deletion actions to update its replica references accordingly. Active modules and the user can receive actions to react on operations in the passive module.

3.2.3 Common Concepts

The previously presented architecture builds the basis for the given requirements. However, to use certain features, such as lifetime based deletion or managed data replication, additional active components are required. For example, each file storing its expiration time defines the lifetime, but a file cannot process a deletion by itself because it is a passive component. Hence, an active component, i.e., an event, is required that executes the required activity.

An often used approach in GACS is the substitution of numerous small events by a single large event. For example, instead of creating an event for each expiration of a file, there is a single event that regularly removes all expired files. Another example are transfers. Instead of creating one event for each transfer, there is a single event that manages the creation, progress, and completion of numerous transfers.

Typically, these large events increase their schedule time during the execution of the payload. As explained in 3.2.1, this instructs the simulation engine to schedule the event again. However, the choice of the increment of the schedule time depends on the event. For some events, it might be possible to calculate the exact time point. Other events might increment the time by a random value, e.g., for irregular data generation. The most of the implemented events are configured with a *reschedule frequency*. This frequency represents a constant or randomly generated increment for the scheduling time.

Beside the default simulation engine, the simulation component contains several event implementations. These events provide active functionality that is commonly required to implement models and eases the use of some architecture features. The events are explained in the following.

Transfer Manager

The transfer manager event is used to initiate, update, and keep track of transfers. The transfer manager event can be compared to the FTS, which manages transfers for the WLCG resources. Both systems provide functionality to create transfers given a source resource and a destination specification. In addition, both systems move the transfers through different states. Typically, network links allow a limited number of active transfers. Hence, transfers start in a waiting or queuing state. Afterwards, they become active until the destination data is complete, and the transfer is finished. The transfer manager provides queues for additional replication requests, respecting the number of active transfers and its limit. The FTS uses central message brokering services to notify systems about transfer state

changes. The transfer manager uses the action interfaces to notify the various simulation components about transfer state changes.
Each execution of the transfer manager event updates all existing transfers. That means, based on the passed time, the destination replicas are increased, network traffic is consumed, and completed transfers are replaced by new transfers from the queue. Furthermore, the transfer manager uses action interfaces to fail transfers if the source replica of a transfer is deleted.
By default, there are two transfer manager events available in GACS. The two transfer manager events implement two different approaches of calculating the progress of transfers. The first approach calculates the transferred volume between two progress updates by using the passed time and the bandwidth. The second approach calculates the transferred volume based on a configurable transfer duration. These two approaches allow using a bandwidth-based or a duration-based transfer model.

Transfer Generator

As mentioned in Section 3.1.1, the transfer generation logic is typically the most relevant part to simulate a model. A straightforward approach of implementing this logic is to create a transfer generator event. This event defines all the details about the generation of transfers, including the number and frequency of the generation of transfers, the source replica selection, and the destination storage selection.
Typically, the logic implemented in transfer generators is very specific to a certain model. Thus, for more complex models, custom transfer generators must be implemented. For simple cases, built-in transfer generators are available. The built-in transfer generators are based on configurable random distributions, i.e., they generate transfers by selecting sources and destinations randomly.

Data Generator

A common task for a simulation is the generation of new data. The data generator event provides generic functionality that allows creating new files and replicas. The primary configuration options of a data generator event are a list of destination storage elements for generated replicas, the number of files and replicas to create, a file size distribution, and a file lifetime.
If more than one storage element is configured, a list of ratios can be provided. Using these ratios, it can be configured that a certain number of files should have multiple replicas. For example, creating 100 files and providing a ratio list with the values 0.7, 0.2, and 0.1, would result in the creation of 70 files with one replica, 20 files with two replicas, and 10 files with three replicas. A random or ordered selection of

the storage elements can be configured.
The number of files, file size, and file lifetime can be configured using the random number configuration functionality. Furthermore, the event can be configured with a rescheduling frequency. A rescheduling frequency of 0 indicates that the event will be executed only once. The single execution of the event is typically used to initialise storage elements with data at the start of a simulation.

Reaper

The Reaper event is the event that takes care of the lifetime based deletion. The name is adopted from the component that manages the deletion in the real ATLAS distributed data system. The functionality is also inspired by the real Reaper system. On each execution of the Reaper event, all expired files are extracted from the file catalogue. Then, all replicas of each file are deleted from their storage elements. Finally, the simulation internal file objects are deleted and the next execution time of the Reaper event is specified. Typically, the execution intervals of the Reaper event are specified by the rescheduling frequency.

Cloud Billing

The billing generator event is used to output and store the bills of cloud resources. The only configuration property is the reschedule frequency, which defaults to 30 days in simulation time. Each time this event is invoked, it uses the given cloud management objects to calculate the cost based on the passed time. The billing objects are persistently stored using the output module and optionally are printed to the standard output.

Heartbeat

The heartbeat event is an event that writes various values about the current simulation state to the standard output. The event can be configured with a reschedule frequency, an arbitrary number of event references, and an arbitrary number of transfer manager references.
Each time the Heartbeat event is executed, it iterates through the referenced events. For each event it writes the event name and the accumulated execution duration in real time of the event, as described in Section 3.2.1, to the standard output. It also prints the share of the total execution duration for each event. Furthermore, the heartbeat event prints the statistics of the configured transfer managers, including the average transfer duration and the number of active, completed, and failed

transfers.

Action Interface

As pointed out in Section 3.2.1, the output module does not allow editing already saved data. This means data can only be saved when all information of the data are available. For example, a replica can only be stored after it was deleted, so that the deletion timestamp is known. For this reason, an implementation of the action interface is provided in the simulation module. This implementation is used by the various transfer generators. It is notified when a file or replica is deleted and stores it using the output module.

3.3 Discussion of the Architecture

As mentioned earlier, the simulation is designed for models from a data management perspective. This leads to a focus on storage, network, and data resources. Data management systems typically use abstraction layers for the underlying resources, i.e., the storage and transfer systems are black boxes with well-defined interfaces. This perspective is fundamental for certain design decisions of the simulation architecture.

3.3.1 Event Size

One of the most noticeable differences of the presented architecture compared to the considered related simulation toolkits is the large comprehensiveness of the events. The described related simulations in Section 2.4.2 use significantly more fine-grained events. Two approaches can be considered, related to the scale of events.
The first, most intuitive approach for an event based simulation is to schedule an event for each required activity. For example, for the lifetime based file deletion a deletion event would be scheduled for each file expiration time, i.e., one event for every file. The fine-grained event approach was already mentioned in Section 2.4.3. The approach discussed in that section schedules a transfer completion event directly after creating the transfer.
A disadvantage of the approach using numerous fine-grained events is that it can reduce the scalability of a simulation. This is the case if the schedule times of the events are changed frequently, e.g., the expiration time of data can be increased or decreased and the transfer durations change continuously. Changing the scheduling time requires searching the position of the event in the schedule, removing it, and

reinserting it. A tree data structure could be used for the schedule, which allows inserting, searching, and deleting in logarithmic time complexity. However, with a single schedule, the scalability would still depend on the overall number of events and the number of times events have to be rescheduled, which are especially large in this approach.

The second approach that was considered uses fewer events with a larger payload. The approach is inspired by the deletion concept of the real ATLAS data management. The real system regularly evaluates the deletion conditions of data and eventually triggers a deletion at the storage endpoint. In GACS, this approach is implemented with a single event that updates all or a large part of the passive objects. For example, instead of having an expiration event for each file object, there would be the Reaper event that is executed regularly. Each time the Reaper event is executed, it collects all expired replicas and triggers their deletion at the storage endpoints. Only one Reaper event is scheduled at any time. Depending on the execution frequency of the event, this approach introduces an inaccuracy, e.g., if the event is executed every 10 simulated seconds, the maximum error of a deletion is 9 seconds. On the other hand, this approach potentially improves the scalability by reducing the number of events and removing the requirement to reschedule events.

3.3.2 Simplifications

Another point of the presented architecture that should be mentioned is the neglecting of certain latencies that are typically introduced by the data management system. For example, downloading a file from the WLCG requires various steps that each induce a delay. These delays include time for authentication at the data management, requesting the existing replicas of the file to download, selecting a replica, authenticating at the storage system, and finally downloading the replica. By default, these delays are not considered in detail by GACS. It is possible to configure a fixed or random start latency of data transfers, but there is no differentiation among the various delay sources.

These latencies were considered neglectable because they typically cover only a small fraction of the overall data transfer. For example, the process of resolving replicas with Rucio normally requires a few seconds. On the other hand, data transfers can require several minutes or hours. Presumably, this simplification significantly improves the scalability of the simulation and reduces the complexity of the simulated models. However, it should be reconsidered when creating simulation models that potentially stress the data management system in a way that would increase the neglected latencies.

Considering the storage systems as black boxes is another point that potentially introduces inaccuracies. For example, the simulation does not consider the IOPS limit of the underlying storage systems. The parameters for the planned simulation use cases were calculated using real world data. Thus, the limits of the storage systems were implicitly considered. However, using different scenarios, e.g., in the case that many small transfers are created, limits of metrics, such as IOPS, should be considered.

In the current architecture, the bandwidth of network links is provided as a fixed configuration value. This means there are no fluctuations of the bandwidth, which typically is not realistic. Related simulators, such as SimGrid, implement background traffic generators, which generate fluctuations of a network link bandwidth. For the planned simulation use cases, the bandwidth values were calculated using a few months of monitoring data. In this way, uncertainties in the bandwidth are already included in the bandwidth values. For this reason, the bandwidth fluctuations were not implemented into the architecture. However, adding this feature belated would not require significant changes.

3.3.3 Output Module

During development, the output module passed through various approaches. The first approach used plain CSV file dumping. This approach was motivated by the advantage of a straightforward implementation and high performance. However, the CSV approach became complex as a more flexible and dynamic data model was required. Furthermore, there was no form of constraint checking.
This led to the consideration of implementing a database system. The first implementation was based on SQLite. The advantage of SQLite is the small size and small overhead. It can be included to the code using a single source file and provides a rich set of SQL features. Two advantages of the database approach are the insertion of data into more explicit structures and the automatic checking of data constraints. Additionally, the evaluation of the data is more flexible. When using CSV data, the parsing for the evaluation is based in the order of the CSV data. Then, for each evaluation type, data have to be selected, filtered, and aggregated manually. A database approach eases these tasks by providing a query language interface.
With larger simulation scenarios, the amount of output data increased. The query interface of SQLite did not scale properly with tens of millions of replicas and transfers. For this reason, an output module backend for a different database system was implemented. PostgreSQL is a significantly larger project than SQLite. Typically,

PostgreSQL runs in a dedicated server process. Tests showed that PostgreSQL performed significantly better in both, inserting the data and running queries, than SQLite. However, when the amount of data becomes larger and the required operations for the evaluation becomes more complex, even PostgreSQL started to require an increasingly large response time.
For this reason, the same approach that is used for the real world monitoring data was adapted. That is, migrating the data from PostgreSQL to a Hadoop file system and running the filter and aggregation operations using map reduce algorithms. For this thesis, the goal was to achieve a data format similar to the real world monitoring data format to allow comparing the results more easily. In the case that the output volume becomes too large, it would be possible to implement runtime data aggregation into GACS.
A different approach that is commonly used in the related simulations is the use of traces. These traces store all simulation states for any processed time point. To reduce data, only the change between two subsequent simulation states is stored. In this way, it is possible to calculate the state of the simulation at any given time following the simulation state changes.

4 Simulation Tool Implementation

This chapter focuses on the implementation details of the previously explained architecture. First, an overview of the implementation is given, explaining the choice of the programming language, the required dependencies, and the used design patterns. Moreover, the details of the configuration system implementation are described and how it is integrated into the overall simulation runtime. Afterwards, the most relevant implementation information, for each of the modules describe in Chapter 3, will be described. Then, a simulation scenario will be discussed, which was used to validate the correctness of the basic functionality of the simulation. The discussion includes the details of the scenario setup, the preparation of the used parameters, and the evaluation of the results. Finally, the chapter ends with consideration of the scalability of the simulation.

4.1 Overview

The first approach considered was an implementation in Python using the SimPy package. Early in the development process, the functionality was ported to C++. The primary reasons for an implementation in C++ are:

- When iterating data, C++ has a significantly higher performance potential. Native loops in Python provide a good performance compared to interpreted loops, but native loops are limited to very static operations.
- The reduction of the memory footprint of the basic functionality of GACS because Python had a noticeable memory overhead.
- Only the scheduling functionality of SimPy was required, which can straightforwardly be reproduced using a priorty queue.

To reduce the complexity, GACS is not built as a library but as a single, extendable program. In the case of a large number of model implementations, this might result in slightly more work of organising the code. The models are typically specified by user code. The GACS code is publicly available on GitHub [Weg21].

4.1.1 Software Dependencies

The vast majority of the functionality of GACS is based on the Standard Template Library (STL). GACS requires compilation with the C++17 standard, including the C++17 functionality of the STL. The used features include file system access, random number generation, time measurements, thread creation, various data structures, and various constants such as data type limits.
The STL provides various classes to generate random numbers. The random generator used by GACS can be exchanged. By default, the minimal standard generator is used. This is based on a linear congruential generator using the sequence $x_{i+1} = (ax_i) \mod m$. In GACS, the default values for the multiplier a and m are used, which are $a = 48271$ and $m = (2^{31}) - 1$. The period of the sequence is m.
For time measurements, the STL provides the `std::chrono` namespace. It provides the `high_resolution_clock` class to receive timestamps with the highest possible resolution for the used platform. Furthermore, the `std::chrono` namespace provides classes to calculate durations and transform them between different units.
Three types of data container from the STL are used in GACS. First, sequence containers that allow sequential access, e.g., dynamic arrays and lists. Second, ordered associative containers that keep elements sorted, such as, maps and sets, which are implemented as red-black trees. Last, unordered associative containers that implement hash-based element access.
Arrays and lists are used if index-based element access or fast iteration is required. Arrays are preferred if it is not required to maintain the relative order for insert and remove operations. Lists provide the advantage that references to elements stay valid if the list is changed. Maps and sets are used if the order is relevant and sorted insertion and iteration is required. In addition, they allow for efficient searching of elements. Unordered associative containers are used when iteration is less important but fast element access using an element identifier is required.
Another feature used from the STL are the smart pointers. Most of the objects in the simulation have a strictly defined ownership hierarchy implemented using unique pointers from the STL. The simulation engine object represents the root object, which creates and owns, e.g., the Rucio object. The Rucio object creates and owns site and file objects. A reference to an object is generally implemented as a native object pointer. If an object uses references, it must be considered that the referenced object might be deleted by its corresponding owner object.

For the configuration files, the JSON format is used. JSON was chosen because the format fits intuitively into the data structures in the code, such as lists, maps, or values. Parsing JSON with a library, like the used nlohmann library [Loh21], requires

small development effort. Furthermore, the JSON format does not require as much formatting text and many formatting characters as, for example, any Extensible Markup Language (XML) based format.
As mentioned in Section 3.2.1, by default the output module contains a dummy and two database backends. The sqlite backend requires that sqlite is included into the compilation process. This can either be done by linking against the external library file or by integrating the single source file of the sqlite library into GACS. The psql backend requires including the libpq library to be able to use the PostgreSQL API. This also requires the connection to a Postgres server.

4.1.2 Design Patterns

Each component of the modules from Section 3.2.1 is implemented in its own C++ class. The classes provide member attributes and class methods in respect to the required data properties and functionality defined in Section 3.2.1. Certain classes are implemented using various object-oriented design patterns. The following describes the used design patterns and explains the benefits of using them.

A commonly used pattern in the simulation are singleton classes, i.e., classes that prevent the creation of more than one object of themselves. For example, the `CConfigManager` and `COutput` classes are singleton classes. Typically, the implementation is done by declaring the constructor private and providing a static public method to receive the class instance. The advantages are that at most one object exists at the same time, and it can be globally accessed.

The cloud manager objects are created using the abstract factory pattern. Two interfaces must be provided for each cloud implementation. Access to a cloud implementation is possible through an implement of the `IBaseCloud` interface. On the other hand, the `ICloudFactory` interface must be implemented to create objects of the the `IBaseCloud` implementation.
Each cloud implementation must implement the `ICloudFactory` interface and must register the implementation in the `CCloudFactoryManager` class. This class is designed as singleton and stores all available cloud factory objects. The factory object for each cloud is stored in a global static variable of the translation unit containing the cloud implementation. The factory object registers itself at the factory manager when the static variable is initialised.
The advantage of the abstract factory pattern for this part is that new cloud implementations can be added straightforwardly without a need to change code in another module. Furthermore, it allows associating an arbitrary identifier with a cloud. The identifier eases the referencing of the cloud provider in configuration files.

The storage element implementation uses the delegation pattern. The base class `CStorageElement` declares the functionality of a storage element and is used to access it. However, the implementation of the functionality is placed in a delegate object, i.e., methods called on a `CStorageElement` object call the corresponding methods of the delegate object, which contains the actual implementation of the functionality.
A delegate object implements the `IStorageElementDelegate` interface. By default, there are two implementations of this interface. One implementation allows multiple replicas of the same file. The other implementation contains checks that prevent the creation of duplicated replicas.
This pattern was chosen to allow extending the storage element functionality, e.g., for the cloud module, while maintaining the option to dynamically use the base storage element functionality of the delegate implementations. For example, cloud buckets should be accessible through the `CStorageElement` definition but must be able to add cost calculation functionality. Using inheritance instead of delegation, a separate bucket class would be required instead of each delegate. Using delegation, a single bucket class is required to add functionality to the `CStorageElement` class while being able to use an arbitrary delegate.

4.1.3 Simulation Configuration System

The configuration system is mainly used during the simulation initialisation procedure to allow specifying certain parts of the simulation by using configuration files. There are three important parts of the configuration system. First, the nlohmann library provides a JSON class, which represents parsed JSON data and provides various operators to access and iterate the data. Second, the `IConfigConsumer` interface class declares the pure virtual method `LoadConfig`, which expects a JSON object as argument. Every class that is required to load configuration data must implement this interface. Last, the `CConfigManager` class simplifies the loading of configuration files and profiles.
The `CConfigManager` class is implemented as singleton. It provides utility functionality to load files into JSON objects either from an absolute path or from various configuration directories. Furthermore, the config manager implements functionality to resolve sub configuration files into JSON. For example, the infrastructure configuration and the network links configuration are both loaded from the profile file. To prevent having all configuration options in one large profile file and to allow using the same, e.g., network links configuration in different profiles, the profile configuration can use a special configuration key to point to a separate file containing the

configuration options. This will automatically be resolved by the config manager.

```
{
"profile": "simEval",
"output": {
    "dbConnectionFile": "psql_connection.json",
    "dbInitFileName": "psql_init_default.json"
    }
}
```

Listing 4.1: Basic simulation configuration file. It defines the profile configuration to load and the output configuration.

The first configuration file that is opened directly after starting the simulation software is the simulation configuration. A basic simulation configuration file is shown in Listing 4.1. The file is expected to be located at the path config/simconfig.json relative to the working directory of the simulation software.

The simulation configuration contains two important options. First, the *profile* to load. The configuration system uses profiles to keep different configurations for various simulation scenarios organised. A profile is the name of a directory that contains at least a profile.json file. This profile configuration file contains all information for a certain simulation scenario.

The simulation will read the value of the `profile` key and use the config manager to open the profile. The config manager expects that the value of that key matches a directory in the profiles directory, which contains the profile.json. For example, given the Listing 4.1 the config manager would try to load config/profiles/simEval/profile.json.

The second option that must be configured in the simulation configuration is the output module because it is the first module that is initialised. The configuration in Listing 4.1 assumes the postgres output module is used. For this reason, a database connection file is specified. This file is not uploaded to GitHub because it contains the connection string including the username and password to the database server. However, the readme file on GitHub states an example of how to configure this file. Furthermore, the output section in the simconfig file allows specifying an optional database initialisation file, which can contain queries to execute once at the simulation start and once at the simulation shutdown. For example, this is useful to create required tables before the simulation starts and create the index of the tables in one operation at the end of the simulation.

```
{
"maxTick": 5166000,
"clouds": [{
      "id": "gcp",
      "name": "gcp_default",
      "config": {"_file_": "gcp.json"}
    }],
"rucio": {
   "config": {"_file_": "rucio.json"}
 },
"links": {
   "config": {"_file_": "links.json"}
    },
"dataGens": [],
"transferCfgs": [{
      "manager": {
        "type": "bandwidth",
        "name": "DefaultTransferMgr",
        "tickFreq": 1,
        "startTick": 5
      },
      "generator": {}
    }]
}
```

Listing 4.2: Basic profile configuration file. It contains the configuration for the infrastructure, the clouds, and the network links. Furthermore, it allows specifying and configuring the initial events to execute.

After the simconfig is loaded and the output module has been initialised, the profile configuration will be loaded from the specified profile.json file. As outlined in Section 3.2.2, the main parts of a configuration include the infrastructure configuration, the network link configuration, the data generator configuration, and the transfer configuration. Optionally, a cloud configuration can be provided.
Listing 4.2 shows a very basic example of a profile configuration. The first key `maxTick` can be used to define the time in simulation time after which the simulation should shut down.
The clouds key can be used to set up a cloud configuration. The list can contain one object for each cloud to configure. In the example, setting the value of the key `id` to gcp results in the creation of a Google cloud object. The key `name` passes a name to the Google cloud instance, which is used for the general output and bill output. The value of the `config` key can be used to define the main part of the

configuration for the cloud. In the example, the special key `_file_` is used. This key allows referencing a separate file that contains the configuration that should be applied. The config manager will automatically resolve those keys internally.
The `rucio` key allows providing the configuration for the infrastructure module. The referenced file contains the configuration for the used sites and storage elements. Similarly, the `links` key contains the configuration for the network links.
The `dataGens` key allows providing a list of objects. Each object contains the configuration for a data generator event described in Section 3.2.3. Data generators can be configured for a one time data generation or a regular execution.
Finally, the `transferCfgs` key allows the configuration of the transfer manager and transfer generator events. Using the configuration system, each transfer generator is associated with its own transfer manager. Both can be configured in an object of the `transferCfgs` list. The object can contain the `manager` key to configure the transfer manager and the `generator` key to configure the transfer generator.
The transfer manager can be configured with a name, the tick frequency, and the start tick. These properties can be configured for every event, including data generators and transfer generators. The property special to the transfer manager is the `type`. Currently, the simulation supports one of the values bandwidth or fixedTime. As explained in 3.2.3 this specifies whether the transfers are updated based on a bandwidth configuration or a transfer duration configuration.
The infrastructure, network links, data generators, and transfer configurations will be explained in more detail in Section 4.3 using the simulation validation scenario as an example.

```
1 "fileSizeCfg": {
2     "type": "exponential",
3     "lambda": 0.026,
4     "minCfg": {"type": "minAdd", "limit": 0.009765625},
5     "maxCfg": {"type": "maxModulo", "limit": 134.0}
6 },
```

Listing 4.3: Value generator object configuration. This provides a flexible way to configure simulation properties with numeric values.

Another commonly used concept in configuration files is the flexible configuration of numeric values by using value generator objects, as shown in Listing 4.3. In this example, the property `fileSizeCfg` is configured with an object specifying a value generator object. The most important property for every value generator object is the `type`. The value of this property determines how the value generator will actually generate the numeric values.
Depending on the configured type, different configuration options are required. In

the example, the type is set to exponential, which defines that the value generator will yield exponentially distributed random numbers. This type allows setting the parameter of the exponential distribution using the `lambda` key.
Another example would be to set the type to `fixed`., A value generator configured with the fixed type will always yield the same constant value. Instead of the lambda key, a fixed type value generator requires the `value` property to be set to the constant that should be yield. A full list of available value generator types and their parameters can be found in the documentation [1], which is also reachable through GitHub.

$$limit(x, low, up) = \begin{cases} low, & x < low \\ up, & x > up \\ x, & otherwise \end{cases} \tag{4.1}$$

In addition, all value generator objects can be configured with rules to limit the generated values to a minimum and/or maximum. Listing 4.3 shows an example of the limit configuration in line 4 and 5. Both, the minimum and the maximum limit calculation can be configured with different approaches. Equation 4.1 describes a straightforward approach to limit a value x so that $low < x < up$. This behaviour can be set by configuring the value of the `type` key for the minCfg or the maxCfg to the value minClip or maxClip, respectively.

$$limitUp(x, up) = \begin{cases} x - \lfloor x/up \rfloor \cdot up, & x > up \\ x, & otherwise \end{cases} \tag{4.2}$$

However, limiting the values by minClip or maxClip can lead to spikes in the value distribution at the lower and the upper limit. An alternative that is also used in Listing 4.3 is to add the minimum value to the generated value to ensure the value is larger than the lower limit. Afterwards, Equation 4.2 is applied to limit the value to a value that is smaller than the upper limit.

Internally, a value generator is represented by the `IValueGenerator` interface class that provides a unified way to generate numbers using configurable methods. The `IValueGenerator` interface declares the pure virtual `GetValue` method that is implemented by various subclasses. Each subclass implements a different way to generate a number.

[1] `https://twatgh.github.io/class_i_value_generator.html`

The `IValueGenerator` class provides a static function that takes a JSON object as parameter and uses it to generate an instance of the corresponding subclass. Additionally, the function automatically sets the corresponding limits if configured. This approach of generating random numbers is used because it fits well into the modular design of the architecture, provides flexibility, and is an optimal solution in terms of extendability and maintainability. However, for different use cases, such as the bulk generation of a large number of random numbers, it is most likely that this approach will not provide the best performance. In the case that this use case is required in the future, it would be possible to extend the `IValueGenerator` interface by a method to bulk generate random numbers. This method could then be implemented in the subclasses by high performance approaches, e.g., using parallel random number generation.

4.1.4 Simulation Run Time

The simulation run time can be divided in four stages:

1. The initialisation stage creates instances of all required modules and components. Various configuration files are used to decide which implementations to use and how to initialise them. Based on these configurations, the output- and simulation engine backends are created.

2. The setup stage allows the simulation engine to create simulated resources in accordance to the model.

3. The simulation stage runs the actual simulation. It uses the event scheduling- and time advance algorithm to execute events and increase the simulation clock. The default implementation runs until the simulation clock reaches a configured time, until the event schedule becomes empty, or until an event explicitly requests a shutdown.

4. The finalisation stage allows all simulated objects to pass their last data to the output module. The output module ensures that all data is properly stored.

Figure 4.1 illustrates the four run time stages and the actions of each stage. The first action at program start is setting up the configuration system. Therefore, the first step is defining the path to the config directory and the profile directories in the `CConfigManager` class. This class provides helper functions to load single config files or full profiles into a single JSON object.
After the paths are set, the main simulation configuration file is parsed. This file

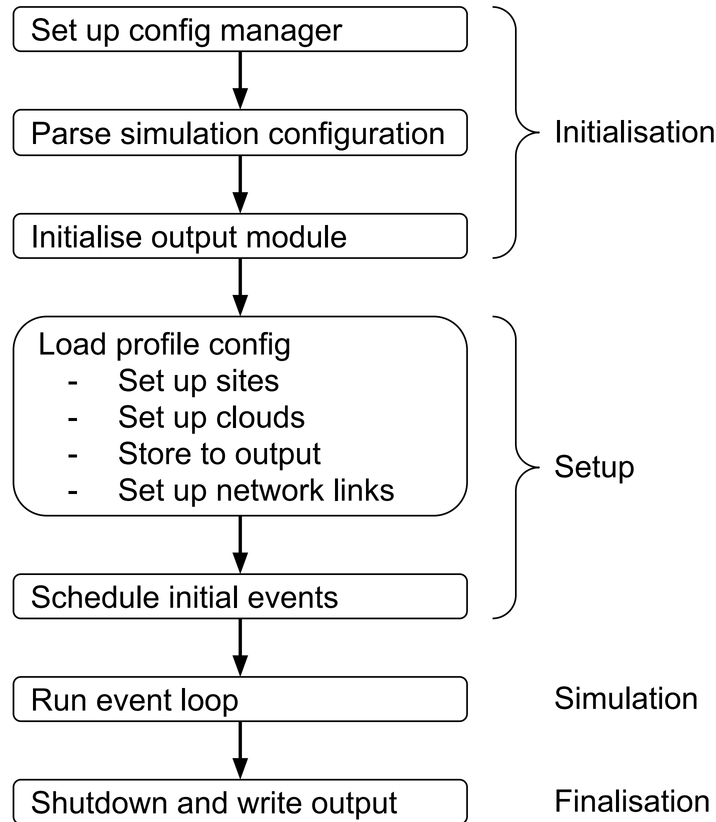

Figure 4.1: Overview of the four run time stages of the simulation. The stages are written on the right. The boxes contain the executed actions from top to bottom.

contains basic information for the initialisation of the simulation. Mainly, the information specify how to initialise the output module and which profile to use. The profile is parsed directly afterwards.

The next step is the initialisation of the output module. This must be done early at program startup to enable the other modules to output initial data. For the output module, different backends can be configured, as described in Section 3.2.1. The output module initialisation is done in two steps. First, the underlying library is loaded, which gives database based backends the opportunity to connect with the database server. The second step starts the output thread and executes optionally configurable initial output commands, e.g., initial queries to empty a database and create new tables.

After the output module was set up, the simulation engine is initialised using the profile JSON object. The default simulation engine implements functionality to set up a simulation scenario from the profile config. The order in which the simulation modules and their components are initialised is relevant.

First, the infrastructure module is initialised. This is done by using the Rucio component. The Rucio component implements the `IConfigConsumer` interface, and thus can be set up from a JSON object. The Rucio component creates all sites and storage elements configured in the corresponding JSON object.
Second, all configured clouds are initialised. This is done by first using the cloud object factory to create the required cloud instances. Afterwards, the `IConfigConsumer` implementations of the cloud objects are used in the same way they are used with the Rucio component.
The third step is the writing of the created infrastructure and cloud objects to the output module. These objects provide the basis for most objects that are created afterwards. The output module should contain the fundamental objects prior to the creation of new objects, e.g., to comply with foreign key constraints.
Finally, the network link objects are created. The configuration file allows specifying the network link details given the source and destination storage element names. The network links are written to the output module directly during their creation.
After all simulation components are set up, the initial events are created and queued into the schedule. The default simulation engine implements functionality to create the built-in transfer generators, transfer managers, and the events described in Section 3.2.3.
When the setup stage completed successfully, the simulations starts running the event loop. The loop runs until one of the explained exit conditions is met, and then the simulation enters the finalisation stage. This stage starts with informing each event left in the schedule about the simulation shutdown. Afterwards, finalisation queries are sent to the database, which can be optionally provided by a configuration file. These queries can be used to instruct the database to build all table indexes at once. Finally, the output thread is requested to write the remaining data and stop execution.

4.2 Module Implementations

This section explains the most relevant implementation details for each of the modules described in Chapter 3. First, the infrastructure module is explained, which primary provides the data structures for files, replicas, sites, and storage elements. Second, it is explained how the cloud module can be used to create a cloud implementation by extending the infrastructure module. Third, the simulation module is explained in more detail. This includes a description of the default event loop and the default transfer managers. Last, the output module is described in more detail.

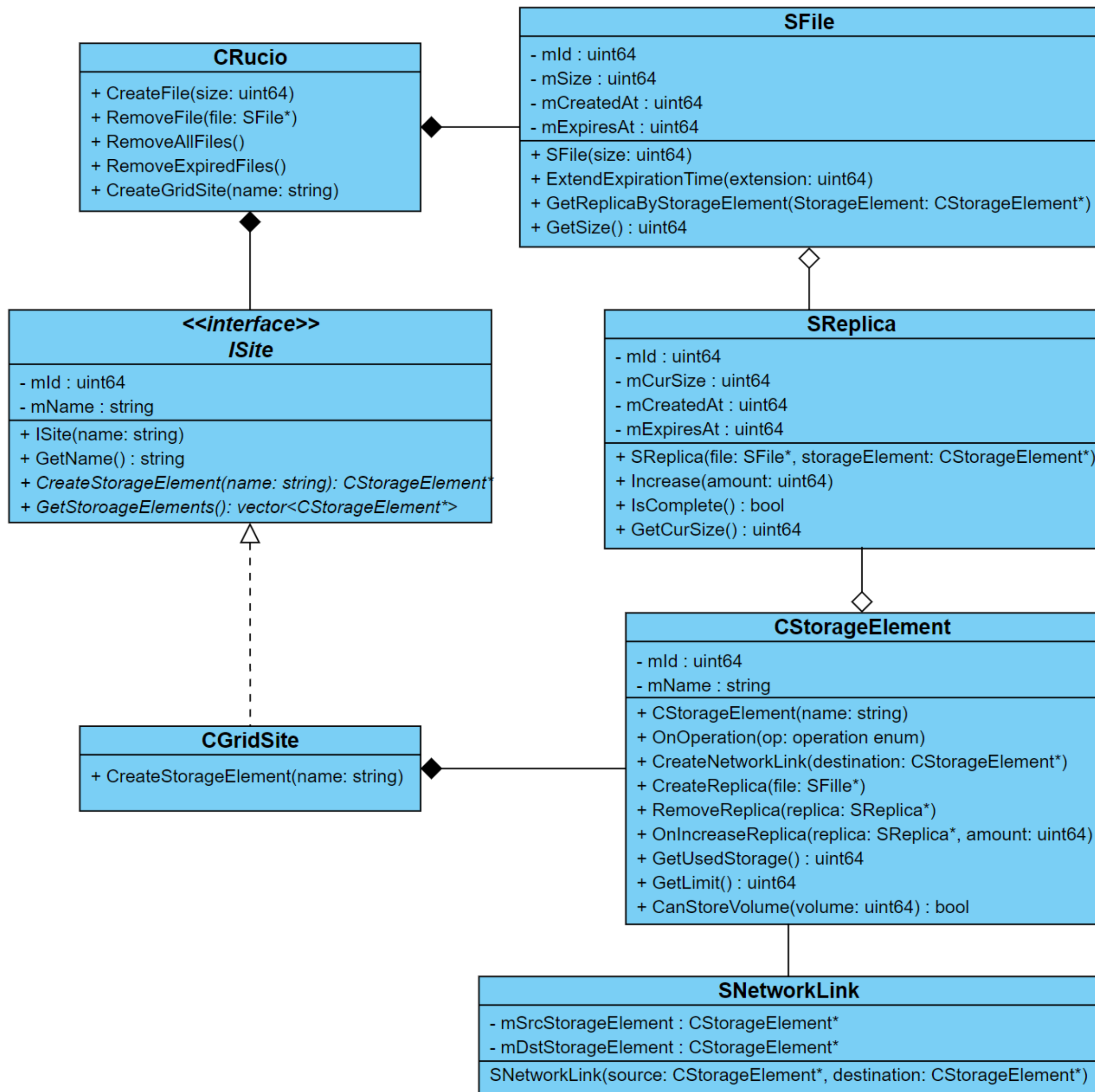

Figure 4.2: UML class diagram of the infrastructure module. Not all attributes and operations are shown to improve the visibility, e.g., certain get methods that allow read-only access are not illustrated.

4.2.1 Infrastructure Module

Since the infrastructure module is a passive module, it mainly comprises classes and functionality to access or modify the data. There are various classes that implement the components of the infrastructure module mentioned in Section 3.2.1. Figure 4.2 illustrates a UML class diagram of the infrastructure module. The class diagrams and flow charts of this thesis were created using Visual Paradigm Online [Par]. Not all attributes and operations are illustrated to keep the diagram size reasonable. For example, the classes provide read-only access through methods to most of the private attributes, such as the ID. Not all of these methods are illustrated. However, the most relevant classes and their most relevant attributes are shown.

The Rucio component of the infrastructure module is represented by the `CRucio` class and provides the central access to the infrastructure module. The `ISite` interface is implemented by the `CGridSite` class to represent grid sites. In addition, there is an implementation of this interface in the cloud module to represent cloud regions. The `CStorageElement` class provides access to simulated storage elements. Finally, there are three relevant data structures in the infrastructure module. The `SFile` structure to represent simulated files, the `SReplica` structure to represent simulated replicas, and the `SNetworkLink` structure to represent network links between storage elements.

Files and Replicas

Most of the methods provided by the `SFile` structure are used for the read-only data access. However, a few additional methods exist to improve usability and keep the data consistent. Each file contains an array of pointers to its replicas. Consistency of this array must be ensured in case a replica gets created or deleted. This is done by two methods. First, `SFile::PostCreateReplica()` which is called by a storage element after the creation of a new replica. Second, `SFile::PreRemoveReplica()` which is called right before the removal of a replica.
The last relevant method is `SFile::GetReplicaByStorageElement()`. This method takes a storage element pointer as paramter. The method iterates over the replica references of the file to check if a replica exists at the given storage element. Since a file typically has only a few replicas, compared to a storage element that stores potentially millions of replicas, it is more efficient to check a file for replicas at the storage element than to check the storage element for replicas of a file.
The `SFile::mSize` property specifies the size of the file and must be known at construction of the file. Moreover, the attribute must not be changed after construction, thus `SFile` provides read-only access to the file size.
In contrast to a file, the size of the replica can change. After creation, the size is zero. The replica can be increased by a given amount using the `SReplica::Increase` method. This method also notifies the storage element that a replica was increased, and thus more storage is consumed.

Sites and Storage Elements

The `ISite` class is abstract because it has two methods that must be implemented. First, a method to create `CStorageElement` objects and second, a method to get references to the created storage element objects. The `ISite` class has at least the

CGridSite implementation in the infrastructure module to represent grid sites. The cloud module can specialise this class to create cloud regions.

The CStorageElement class represents a storage area of a site and implements the tracking of storage consumption, including replica creation, deletion, and size changes. Thus, a storage element comprises variables to describe storage limits, used storage, allocated storage, and an array to keep references to all replicas associated with this storage element. Furthermore, a storage element provides methods to create, delete, and increase replicas. These methods are implemented to also invoke related action interfaces, and thus keep all objects consistent.
In addition, storage element objects are used to create CNetworkLink objects. For this purpose, the storage element class provides a method that takes a reference to the destination storage element object and additional parameters describing the network link, such as bandwidth.
In terms of the ownership relations explained in Section 4.1.1, the Rucio object has the ownership of site and file objects and has to manage their life cycles. Site objects have the ownership of storage element objects, which have ownership of replica objects and network link objects.

File and replica creation process

Figure 4.3 shows exemplarily the usage of the different entities for the creation of new files and replicas during the simulation. Since the infrastructure module is passive, an active module action is required to start the creation. In the Figure, this action is represented by an event. For example, this could be a data generator event.
The event uses the Rucio instance to create a file object. This will simply create a new SFile object, register it in an array in the Rucio instance, and notify the action listener interfaces. The new file object can then be passed to the CreateReplica method of an arbitrary storage element instance.
The replica creation inside the storage element instance will first check the storage constraints, i.e., whether the replica fits on the storage element. If sufficient storage is available, the amount given by the file size will be allocated on the storage element. Afterwards, a new SReplica object is created and registered in an array of the storage element. Subsequently, the file is notified first that a new replica was created. Finally, the action interface is called to notify potential listeners that a new replica was created.

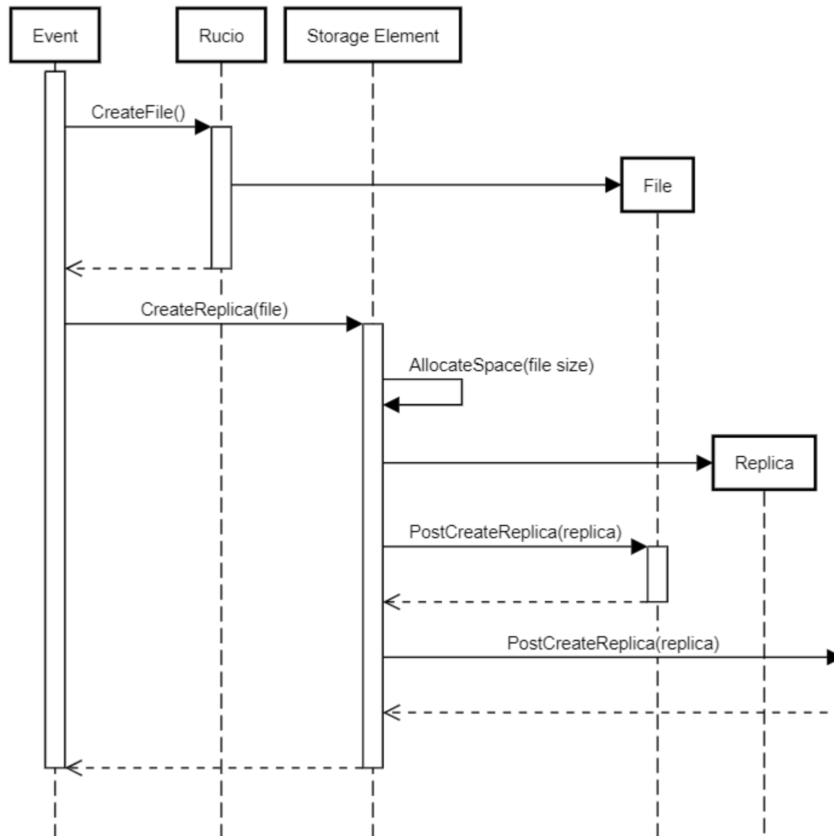

Figure 4.3: Schematic of the different method calls and class involvements to create a file and a replica of the file.

4.2.2 Cloud Module

As explained in Chapter 3, the cloud module extends the infrastructure module to provide certain cloud storage functionality. As shown in Figure 4.4, GACS provides a common interface for all clouds. This interface consists of a cloud factory to create objects of differently implemented cloud. All cloud implementations must implement the base cloud interface. The base cloud interface can be used to create regions. These regions represent the cloud implementation of the `ISite` interface from the infrastructure module. Furthermore, the base cloud interface describes a method used to process the billing. The bill is represented by another interface, the cloud bill interface.

GCP implementation

Beside the interfaces, Figure 4.4 also illustrates the GCP implementation of these interfaces. The first step to create a new cloud implementation in GACS is to create an implementation of the base cloud interface. For GCP, this is done using the

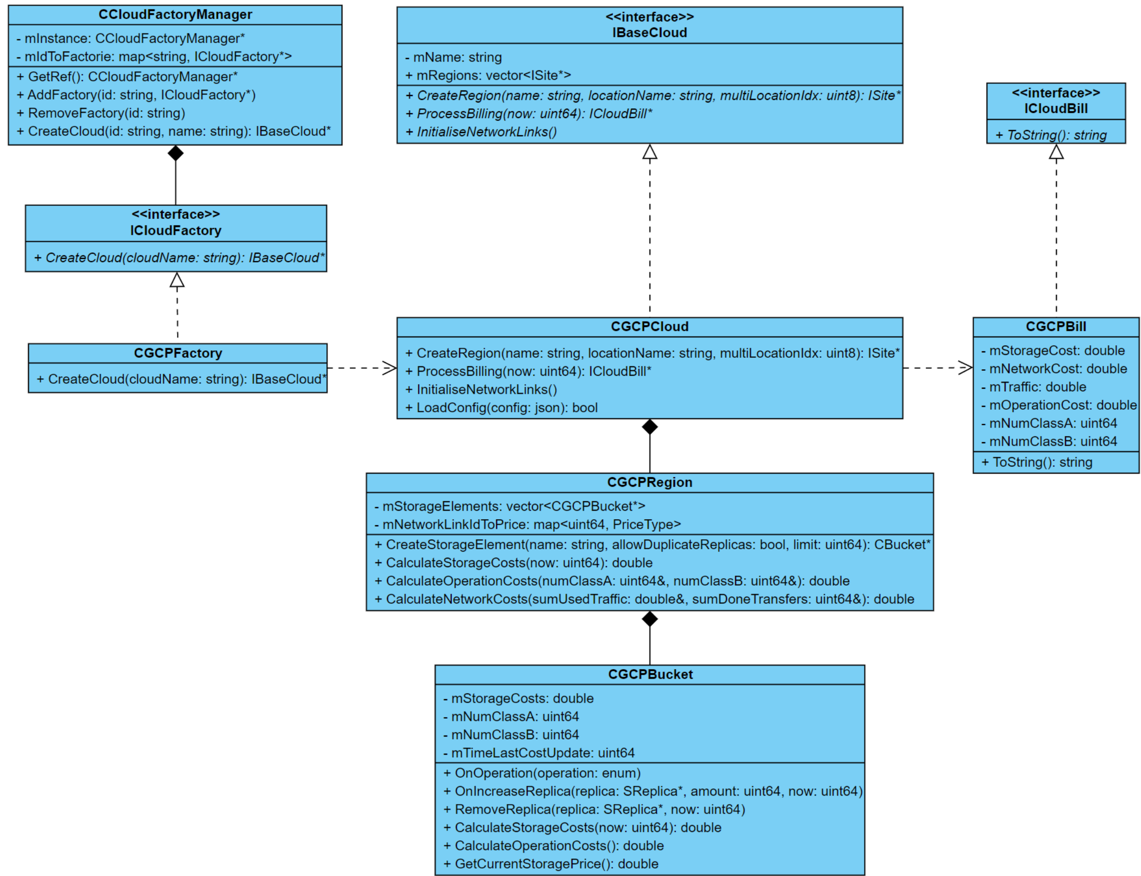

Figure 4.4: UML class diagram of the cloud module. Not all attributes and operations are shown to improve the visibility, e.g., certain get methods that allow read-only access are not illustrated. Furthermore, in the code, gcp is not used as class name prefix but as namespace.

`CGCPCloud` class. This class implements the methods required by the interface, such as `ProcessBilling` and `InitialiseNetworkLinks`.
Furthermore, the GCP cloud implements the `LoadConfig` method. This method is especially important to load the information of the implemented cloud services. As mentioned in Section 2.2.2, GCP describes the information of its services by SKU IDs. All SKU IDs with their information were exported through the GCP API and exported into a JSON file. The file containing the SKU ID information must be specified in the corresponding profile configuration of the cloud. The `LoadConfig` method will use the configured SKU ID file to load the pricing information for storage, transfers, and operations.

The next steps when creation a new cloud implementation are the implementation of a class inherting from `ISite`. For the GCP implementation, the `CGCPRegion` class is used for this. The `CreateRegion` method of the GCP cloud implementation will create and return objects of the GCP region.

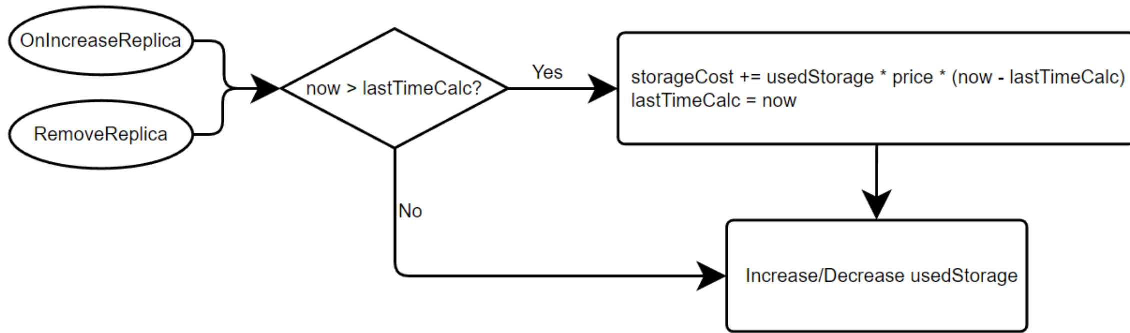

Figure 4.5: Flow chart that shows the storage cost tracking of the GCP implementation in GACS.

Since sites are used to create storage elements and the storage elements for a cloud must implement certain cost tracking features, the `GGCPBucket` provides a custom storage element implementation. In addition, the GCP region is implemented to create and return GCP buckets instead of grid storage elements.
As mentioned earlier, the main point of using specified versions of the site and storage element classes is to add cost tracking. The class diagram in Figure 4.4 shows that the region class and the bucket class contain numerous methods to calculate certain costs. In general, the cost tracking functionality consists of two parts. First, the tracking of costs or operations that introduce costs as they occur. Second, the summarisation of the costs and the resetting of the counter variables.

There are three types of cost that are considered for using GCP. These are storage cost, network cost, and operation cost. The tracking part of the storage cost is illustrated in Figure 4.5. The tracking is done for each bucket object. Each time a replica is increased or removed, the used cloud storage is potentially changed. A check is done whether simulation time has passed since the last time the storage cost were updated. If that is the case, the storage cost is updated by summing the so far calculated storage cost with the storage cost introduced since the last update. The unit of the price is cost per volume per time.
The network cost is not as explicitly tracked as the storage cost because compared to the used storage, the used network traffic can not be reduced but only increased. For this reason, the calculation of the network cost is done in the region objects. The process straightforwardly iterates through the network links of the buckets, collects the used traffic, determines the pricing information of a given network link, and sums the product of the traffic and the price.
For the calculation of the operational cost, the buckets have to track the number of operations explicitly again. However, since the operational cost is purely based on the number of operations and not time based in any way, the implementation is less

complex than the storage cost tracking. Each bucket implements the `OnOperation` method, which is called for each operation, such as inserting new data or accessing existing data. The bucket only needs to count the different operations. The actual cost calculation can be done straightforwardly by multiplying the number of operations with the corresponding pricing information.
The `CGCPCloud::ProcessBilling()` method is implemented to calculate all costs, reset all cost calculation variables, and start a new billing period. The implementation iterates through all region objects, letting each object calculate its storage, network, and operation costs. Afterwards, the results are stored in a `CGCPBill` object and returned.

The final step to implement a new cloud is to create an implementation of the `ICloudFactory` interface. This interface is used by the simulation engine to create the specialisation of the `IBaseCloud` interface. The factory object is instantiated as a static variable and registers itself in the cloud factory manager singleton. The cloud factory manager maps a cloud ID, such as `gcp` to a factory object, e.g., the GCP factory instance. This allows finding a specific cloud factory by name, e.g., using configuration files.

4.2.3 Simulation Module

The simulation behaviour is mainly defined by the simulation module. Figure 4.6 shows the main classes of the simulation module, containing the most important attributes and operations. The `CSchedulable` class provides the base class for every event that can be executed by the simulation engine. The `IBaseSim` interface defines the operations that each simulation engine implementation must provide.
The base simulation class and the schedulable class are in a composition relationship. That means when the base simulation class is shut down, e.g., because the maximum tick was reached, also all events related to this simulation instance are shut down. Furthermore, in the more uncommon case where no more events are left, e.g., because the events decided not to request a rescheduling, the simulation would shut down too.
The schedulable base class provides two overridable methods. The `OnUpdate()` method is used by the simulation engine to execute the event when its scheduled time was reached. The scheduled time point is indicated by the `mNextCallTick` variable. This variable is used to indicate the simulation engine the next time point the corresponding event should be executed.

Figure 4.7 outlines the execution and rescheduling mechanic of an event. When the simulation clock reaches `mNextCallTick` the `OnUpdate()` method of the event will

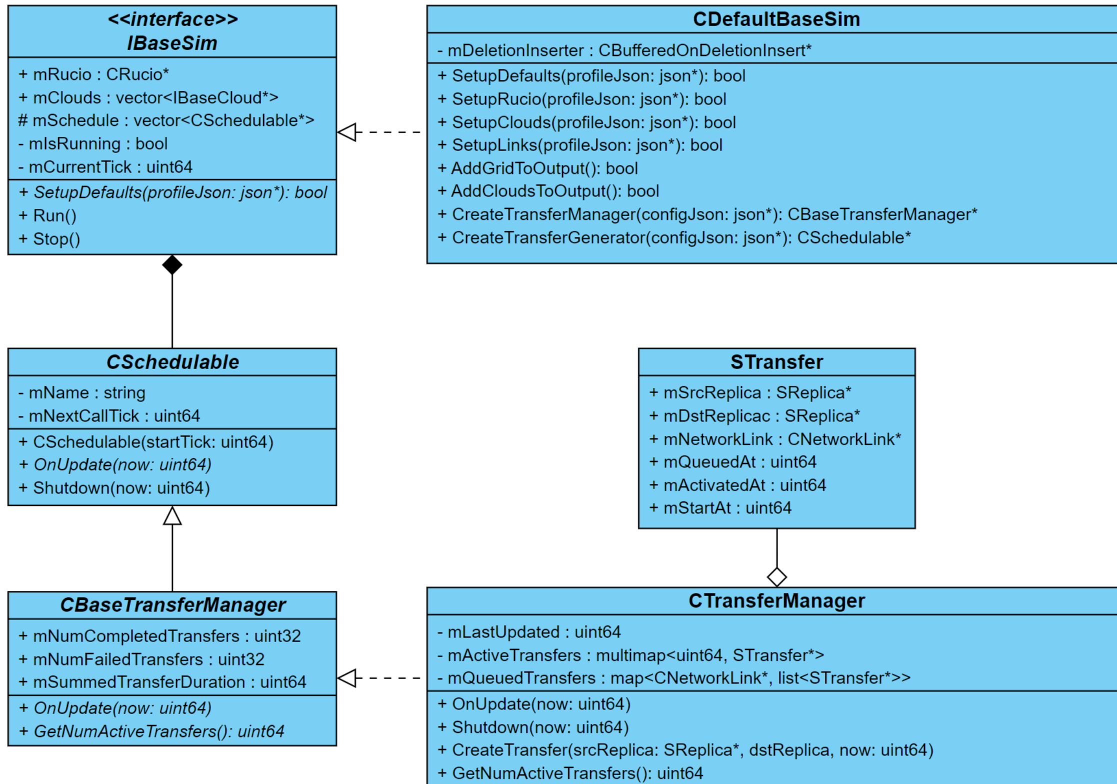

Figure 4.6: UML class diagram of the simulation module. Not all attributes and operations are shown to improve the visibility, e.g., certain get methods that allow read-only access are not illustrated.

be called. The event executes its implementation dependent payload. Afterwards, the event determines whether and when it must be rescheduled by the simulation engine and sets the next call tick variable accordingly. If the next call tick variable was not updated to a future time point, the overrideable method `Shutdown()` will be called, and the event gets deleted.

The approach of how the simulation scenario is set up, how the configurations are applied, and how the events are scheduled and executed is determined in the `IBaseSim` implementation. The most relevant attributes of this class are the Rucio instance reference providing access to the infrastructure module, the array of cloud implementations, the `mCurrenTick` variable representing the simulation clock, and the array of schedulables representing the schedule.

The `SetupDefaults()` method is required to be implemented by each simulation engine. Optionally, an implementation can override the `Run()` and `Stop()` methods, to customise the engine loop and the stopping behaviour, respectively.

The `SetupDefaults()` method receives a JSON object as input, which provides the profile configuration loaded at program start. The method is supposed to create and configure the infrastructure, clouds, network links, and the initial simulation

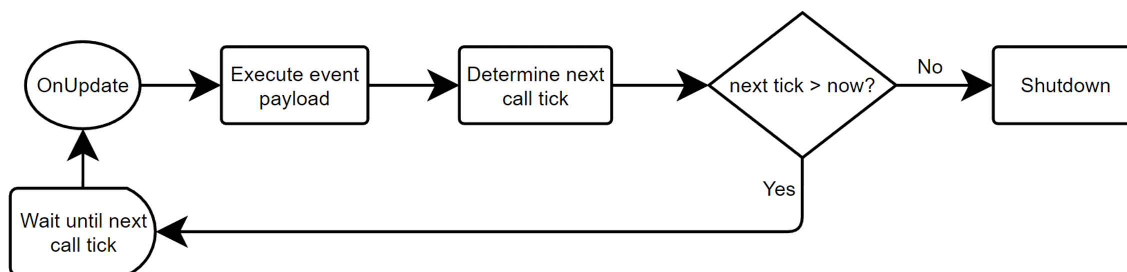

Figure 4.7: Illustration of the execution and rescheduling of an event.

events. The `Run()` method is used to start the actual simulation loop.

```
1 void IBaseSim::Run()
2 {
3   while( mIsRunning
4      && (mCurrentTick <= maxTick)
5      && !mSchedule.empty() )
6   {
7     CSchedulable* event = mSchedule.pop();
8
9     mCurrentTick = event->mNextCallTick;
10    event->OnUpdate(mCurrentTick);
11
12    if(event->mNextCallTick > mCurrentTick)
13      mSchedule.push(event);
14    else
15      event->Shutdown(mCurrentTick);
16  }
17  mIsRunning = false;
18
19  while(!mSchedule.empty())
20    mSchedule.pop()->Shutdown(mCurrentTick);
21 }
```

Listing 4.4: Default implementation of the event loop in the IBaseSim class.

The base simulation interface provides a generic default implementation of the event loop. This default implementation is shown in Listing 4.4. The listing shows the exit conditions of the event loop in the head of the while loop, which were already discussed in Section 3.2.1. The time advance algorithm is implemented by extracting the next event from the schedule in Line 7 and setting the simulation clock to the time of this event in Line 9. Subsequently, in Line 10 the event is executed. If the event requires to be scheduled again, it increased its `mNextCallTick` variable. This is tested in Line 12 and the event is either rescheduled or removed. At the end of the simulation, all existing events are informed about the end of the simulation.

The code in the listing used the priority queue adaptor for the `mSchedule` variable, which is based on a binary heap. Another approach would be a multiset, which would result in a similar event loop code. However, a multiset would be based on a red-black tree.

As shown in the class diagram in Figure 4.6, GACS comes with default simulation engine implemented in the `CDefaultBaseSim` class. It was mentioned earlier that the default simulation engine allows a full set up via configuration files. The way this is done is by implementing the `SetupDefaults()` method to subsequently initialise all modules and their components from the passed-in profile JSON object.
The profile loading is done in various steps. Each step is implemented in a separate method shown in the class diagram. First, `SetupRucio()` is called, which uses the infrastructure module to create grid sites and storage elements. Subsequently, all potentially configured clouds are configured. After this point, all sites and storage elements should have been created. The next action should be to add all grid information and all cloud information to the output module to ensure that the output module can resolve all references when the network links are created. Then, it is safe to create the network links. If all these actions were executed successfully, the simulation infrastructure should be fully configured.
Having a fully configured infrastructure, the active parts of the simulation can be created. This includes mainly the creation and configuration of events. The first events that are created come from the transfer configuration, as explained in Section 4.1.3. The default simulation provides methods that allow creating and configuring a transfer manager and a transfer generator from a given JSON object.
The next events that are created are the data generator events. Since there is currently only a single data generator object, the configuration is directly loaded and applied in the `SetupDefaults()` method. Subsequently, potential cloud billing events are created. Finally, a heartbeat event is created to regularly print simulation statistics to the standard output during runtime.

Default Transfer Manager

As described in Section 3.2.3, GACS provides a default implemented bandwidth-based and duration-based transfer manager. Except for the progress calculation, both transfer managers are identical. The class of the bandwidth based transfer manager is illustrated in the class diagram in Figure 4.6.
The most relevant attributes of the default transfer manager are the transfer queue, the map of active transfers, and the last update time. When a new transfer is created using a transfer manager, the transfer manager puts the new transfer in a queue

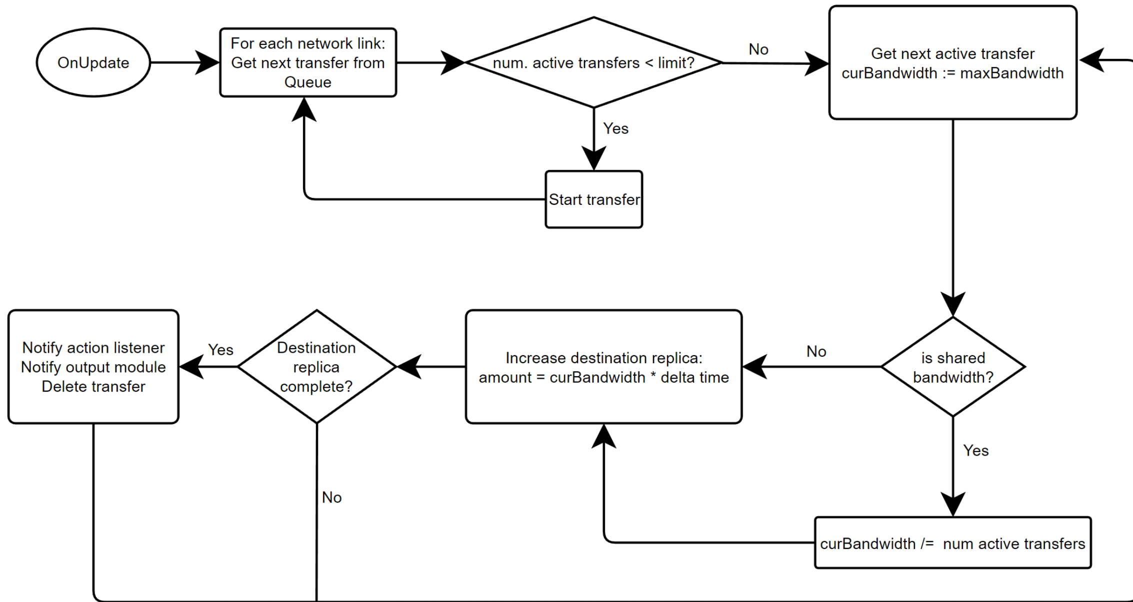

Figure 4.8: Flow chart that outlines the process how the default transfer manager implementation activates and updates transfers.

instead of starting it directly. This is done because the network link required for the new transfer might be limited to a certain number of active transfers. For this reason, the transfer queue in the transfer manager is implemented as map, which contains a list of transfers for each network link.

For the active transfers, a multimap has been chosen as data structure. The multimap groups transfers by their starting time. The starting time of a transfer might be in a future point of simulation time. This is required because, e.g., a storage element might be configured with a certain access latency. That means after a transfer is activated, the actual updating of the transfer only starts after the starting time. Since the multimap provides an ordered data structure, only the first elements of the multimap are considered for updating until the starting time of an active transfer is larger than the current simulation clock.

Furthermore, the transfer manager implements several operations. The operations mainly serve three purposes. First, it allows creating transfer objects given a source and destination replica. This operation consists of creating a new `STransfer` instance using the given replicas and queuing the instance in the transfer queue associated with the corresponding network link. Second, the transfer manager transits created transfers through different states and simulates the transfer progress over time during the event updates. Finally, it notifies registered receivers about transfer completions and failures.

Figure 4.8 shows a flow chart outlining the event update procedure of the default transfer manager. As first action, the queued transfers for each network link are

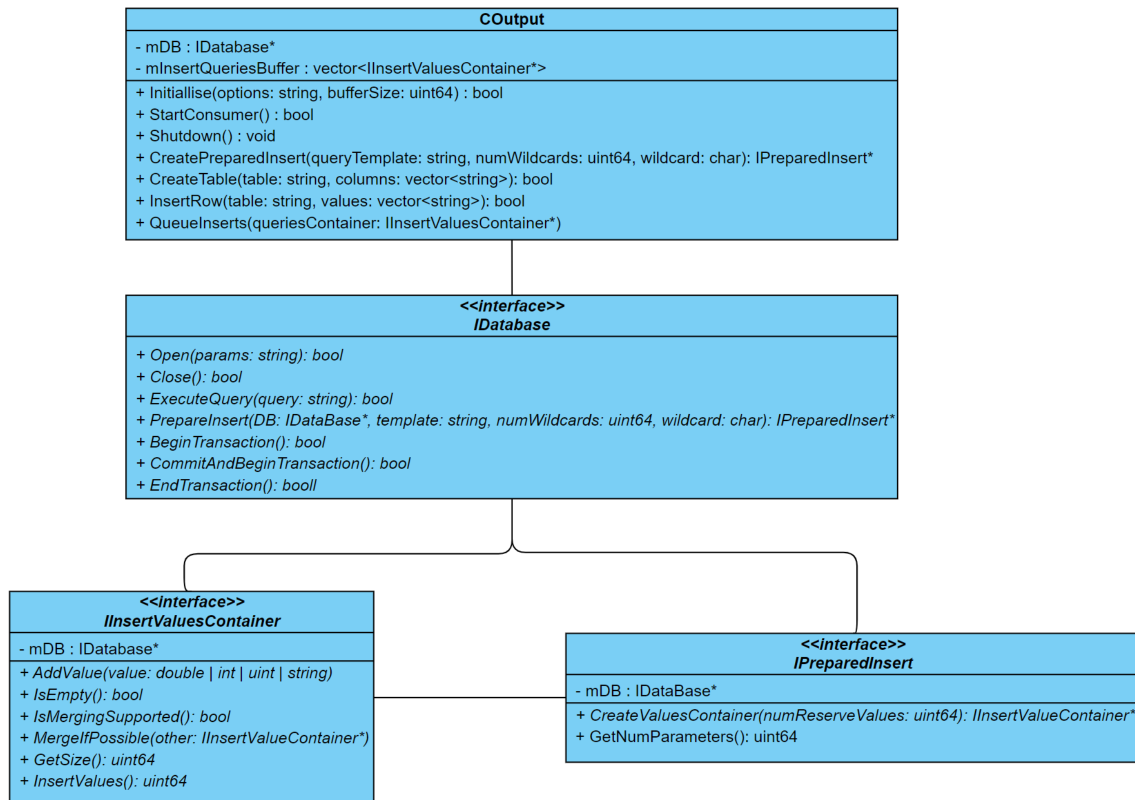

Figure 4.9: UML class diagram of the output module. Not all attributes and operations are shown to improve the visibility, e.g., certain get methods that allow read-only access are not illustrated.

started depending on the number of active transfers and the limit of active transfers allowed.

After no further queued transfers can be started, the active transfers are updated. First, the delta time is calculated by using the last updated time and the current simulation time. Then, the map of active transfers is iterated until the starting time is larger than the current simulation time.

For each iterated transfer, the amount of data to transfer is calculated. This is done by using either the shared or unshared bandwidth, depending on the network configuration. Afterwards, the destination replica of the current transfer is increased by the calculated amount.

Finally, it is checked whether the destination replica was completed during the current update. If this is the case, the action interface listeners are notified, the transfer is stored in the output module, and the transfer itself gets deleted.

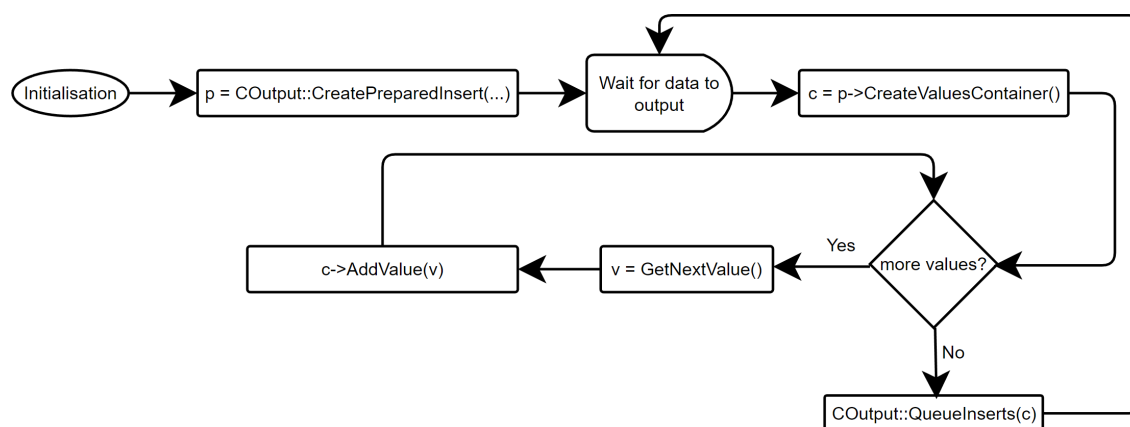

Figure 4.10: Flow chart diagram outlining the process of how to persistently store simulation values using the output system.

4.2.4 Output Module

Figure 4.9 shows the main classes and interface definitions of the output module. The `COutput` class is implemented as a singleton class and globally provides access to the output functionality during the simulation runtime. As mentioned in Section 4.1.3 the output module is the first module to be initialised. This means that directly after the output configuration files were parsed, the `Initialise()` method will be called. Internally, this method creates an implementation of the `IDatabase` interface class and calls its `Open()` method.

If the initialisation succeeds, the output singleton can be used to create implementations of the `IPreparedInsert` interface. The prepared insert object contains information about the format and destination of values that can be written to the output system. However, it represents only a template for the data and no actual values.

To actually output data, a matching prepared insert object can be used to create an object of a `IInserValuesContainer`. This interface represents an actual container for values. The format is specified by the prepared insert object. The value container interface defines several methods to add values of different types to the container.

Figure 4.10 shows a flow chart that outlines the process of writing values to the output system. The best practice is to once create all required prepared insert objects in the initialisation phase of the simulation. Advanced database implementations, such as Postgres, allow precompiling insert statements for improved performance.

When the point is reached that data must be written to the output system, a value container can be created using the prepared insert statement. After all values were

added to the container, the container can be queued for storing it persistently using the output singleton instance.

As mentioned before, the output module uses a separate thread to avoid performance drops during I/O operations when storing simulation results. The output module uses a lock free single consumer single producer queue to guarantee thread safe data exchange. The main thread runs the simulation, and thus represents the producer, i.e., the simulation produces the data and adds them to a value container. When the value container is queued to the output system, the container is internally added to a shared buffer. The consumer thread takes the value containers from the shared buffer and calls their `InsertValues()` method, which stores them persistently in the database.

```
1 void COutput::ConsumerThread()
2 {
3   while(mIsConsumerRunning || (mConsumerIdx != mProducerIdx))
4   {
5     while(mConsumerIdx != mProducerIdx)
6     {
7       IInsertValuesContainer* container = mBuffer[mConsumerIdx];
8       mConsumerIdx = (mConsumerIdx + 1) % BUF_LEN;
9
10      container->SaveToDisk();
11    }
12
13    this_thread::sleep_for(chrono::milliseconds(5));
14  }
15 }
16
17 void COutput::AddData(IInsertValuesContainer* Data)
18 {
19   const std::size_t newProducerIdx = (mProducerIdx + 1) % BUF_LEN;
20   while(newProducerIdx == mConsumerIdx)
21     this_thread::sleep_for(chrono::milliseconds(10));
22
23   mBuffer[mProducerIdx] = Data;
24   mProducerIdx = newProducerIdx;
25 }
```

Listing 4.5: Simulation output thread

Listing 4.5 shows the C++ code for a basic implementation of the single consumer, single producer buffer. The consumer thread from line 1 to 15 runs in parallel to the rest of the simulation. The producer part from line 17 to 25 is called directly from the simulation. The most relevant elements are the shared buffer `mBuffer`, the

consumer index `mConsumerIdx`, and the producer index `mProducerIdx`. The shared buffer is a fixed-size array storing pointers to the containers that contain the output data. The consumer and producer index are two atomic integer variables. The index calculation considers the array size to use the buffer as a circular buffer.
The producer index points at the location in the buffer where the next produced element will be stored. In other words, this position is reserved for the producer thread. The consumer index points at the element in the array that will be consumed next.
Two relevant conditions are used in Listing 4.5. First, if the consumer index and the producer index are equal, the buffer is empty. This is used in line 3 and 5 to check if more elements can be consumed. Second, if the next producer index is equal to the consumer index, the buffer is full. This is used in line 20 to check whether new data can be added. If the buffer is full and new data needs to be added, the simulation must wait until the consumer thread removed an element from the buffer.
The consumer thread uses two while-loops. The outer while-loop ensures that the thread stays active as long as the output module is used or as long as more data can be consumed. In other words, it is ensured that all data will be persistently stored even if the output thread is requested to shut down. The inner while-loop takes the next element, advances the consumer index, and instructs the output backend to store the element persistently.
The implementation assumed that the producer is typically faster than the consumer. Thus, the consumer thread rarely leaves the inner loop and enters the sleep. If this assumption was proven wrong, the implementation could be further improved by using a conditional wait construct.

4.3 Simulation Tool Validation

To validate the basic functionality of GACS, the transfers of the ATLAS derivation input data were simulated. Since this is a process that is already running in production, there are sufficient data available to calculate parameters for the simulation and to provide a scale for the output.
This section firstly describes details about the setup of the simulation for this scenario. Then, the used monitoring data are explained and the method to extract the simulation parameters from them. Finally, the results are evaluated.

4.3.1 Simulation Setup

The validation scenario was implemented in GACS by using the `CFixedTransferGen` transfer generator. This transfer generator is generically implemented in GACS and can be fully configured by configuration files. The main part of the configuration for this transfer generator is represented by a mapping between source storage elements and destination storage elements. In addition, each source/destination storage element pair contains a value generator object to calculate a number of transfers to generate.

The `CFixedTransferGen` generator implements the `PostCompleteReplicas()` action interface, which enables the transfer generator to be notified about finished transfers. For example, this is required when the replicas should be deleted again after transferring.

Since a transfer generator is an event, it can be regularly executed by the simulation engine. Each time the transfer generator is executed, it processes several steps. First, it processes the replicas whose transfers were completed since the last execution, e.g., deleting them again, so they can be transferred multiple times. Then, it iterates through source/destination storage element pairs. For each pair, it calculates the number of transfers to generate.

Subsequently, a random replica of the source storage element is selected for each transfer to generate. If the selected replica is already transferring, the replica array is sequentially iterated starting from the selected replica until a proper source replica is found. A new replica object is created at the destination storage element using the file associated with the selected source replica. Finally, a new transfer is created using the source and destination replicas.

4.3.2 Parameters calculation and configuration

Table 4.1 lists all parameters and their values that were used to simulate this scenario. The data to calculate values for these parameters were taken from the ADC monitoring system. This monitoring system only provides the data of the past two months. For this reason, the simulated time frame was set to almost two months, as shown in the table. Specifically, the transfer data from 2020-05-30 10:00:00 to 2020-07-29 05:00:00 [2] of the three sites with the highest number of transfers during this period were collected.

Five data samples of the monitoring data were considered for the validation. A sam-

[2]The distributed computing infrastructure and activity was not unduly impacted by the global COVID-19 pandemic.

Parameter	Value/Configuration
Simulated time	59 days 19 hours
Transfer mgr. update interval	1 s
Transfer gen. update interval	10 s
No. initial replicas	1000 per site
Throughput	8.10 MB/s per network link
File size	exponentially distributed: $\lambda = 0.61972$ 10.23 MB $\leq$ size $\leq$ 13.73 GB
No. transfers generated	exponentially distributed: $\lambda = 3.33437$

Table 4.1: Parameters and their configuration for the simulation validation.

ple of the file size, a sample of the number of transfers, and a sample of the transfer throughput were used to calculate the parameters for the model. The samples of the transferred volume and transfer duration were used to validate the output of the model.

The parameters of the model were determined by finding a distribution function that can be properly fitted to the real world data. An alternative approach would be to investigate and apply certain curve fitting techniques. In this case, the input parameter would be described by a polynomial. However, the first analysis of the data suggested that the data can be well described by distribution functions. Also, first tests showed that the fitting results can be considered more than sufficient to describe the data.

Since the transfer manager is an event, a tick frequency can be configured as explained in Section 4.1.3. The tick frequency of the transfer manager can potentially influence the run time of the simulation. However, first tests and the scale of the parameters indicated that the validation scenario would not reach a scale that significantly increase the simulation run time. This allowed setting the transfer manager update interval to the lowest value of 1 second to achieve the highest resolution.

The transfer generator update interval configures how frequently the transfer generator is executed. This effectively controls how frequently the random distribution for the number of transfers is sampled. If the chosen parameters allow only the generation of numbers between 0 and 1, it could be possible that no transfers are generated

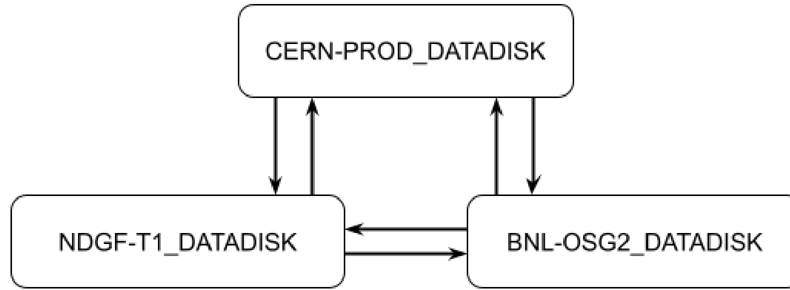

Figure 4.11: Schematic of the infrastructure configuration used for the validation. Each box represents a storage element. The arrows represent the directional network links.

at all because the minimum number of transfer to generate is 1. To prevent this, the numbers could be rounded upwards. However, this would introduce an error. For this reason, the number after the decimal point are summed and considered by subsequent executions of the transfer generator.
Choosing an excessively large transfer generator update interval would result in a high but infrequent number of transfer generations. Conversely, a too small update interval could result in a too small number of transfers to generate. An update interval smaller than the time between two transfer generations does not improve the resolution. The mean of the distribution function of the number of transfers to generate is $1/\lambda = 1/3.33437 \approx 0.3$. This means that on average, using the 10 seconds interval, a new transfer is generated every third update for each storage element pair.
At the start of the simulation, each storage element is initiated with 1000 replicas. After a completed transfer, the destination replica is deleted immediately to allow transferring the replica again. Thus, the 1000 replicas provide a sufficient pool of selectable replicas.

Infrastructure

Figure 4.11 illustrates how the infrastructure was configured for the validation scenario. The used monitoring data were collected from the three storage elements shown in the figure. Not included in the figure are the three corresponding sites because only one storage element of each site was taken. The monitoring data was taken from transfer data in all directions among the three storage elements.

```
1 "rucio": {
2     "sites": [{
3         "name": "CERN-PROD",
4         "storageElements": [{
5                 "name": "CERN-PROD_DATADISK",
6                 "allowDuplicateReplicas": false,
7                 "limit": 0
8         }]
9     }]
10 }
```

Listing 4.6: Part of the infrastructure configuration for the validation scenario showing the most important site and storage element configuration options. The configuration is located in the rucio configuration file.

Listing 4.6 shows a part of the used infrastructure configuration file. For the simulation validation experiment, the full profile configuration is also uploaded to GitHub [Weg21]. The file at config/profiles/simEval/profile.json contains the profile configuration. For the infrastructure configuration part, the profile points to the separate file at config/profiles/simEval/rucio.json file using a special configuration key, as explained in Section 4.1.3.
As shown in Listing 4.6, the rucio configuration object contains a key `sites` that contains a list of configuration objects. Each object in the list will result in the creation of a new site object during simulation runtime. The example listing only shows the configuration of the CERN-PROD site. Each site configuration object contains a list of storage element configuration objects. Each storage element configuration object will result in the creation of a storage element associated to the corresponding site.
The main configuration properties of a storage element are the name, which is commonly used in other configuration files. For example, the network link configuration uses the storage element name to reference the source and destination storage element. The `allowDuplicateReplicas` property can be used to allow multiple replicas of the same file on a single storage element. Another storage element configuration property is the storage limit. A value of 0 means the storage is unlimited.

File size parameter

First, the parameters for the file size distribution were calculated. The monitoring system provided the data in the form of a histogram showing the number of files for a certain file size range. The minimal available bin width of the histogram was 128 MB. To analyse the histogram in more detail and calculate parameters for the

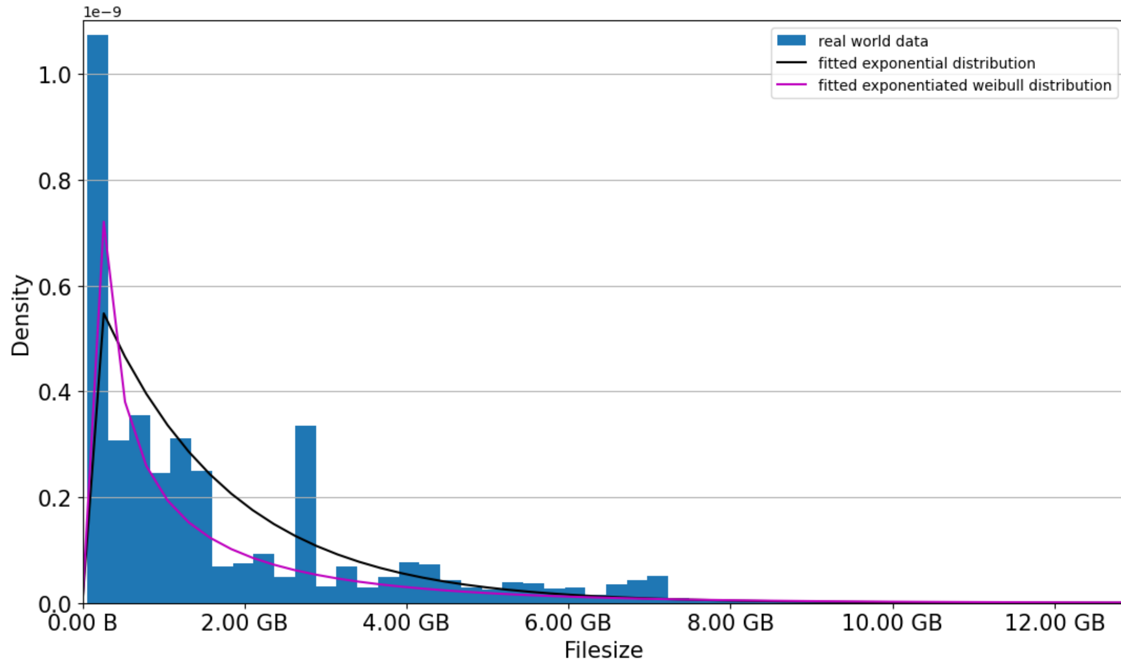

Figure 4.12: Comparison among the real world data file size distribution, the fitted exponential distribution function, and the fitted exponentiated Weibull distribution functions.

simulation, the data were exported into a CSV file.
The CSV data were loaded in a Python script. Therefore, a custom function was written that allowed to do basic filtering or resolve gaps in the CSV data. Resolving gaps might be required because the export from the monitoring system did not export bins that had a count of 0. Thus, if the code using the exported data expects the bins to be equally spaced, the empty bins have to be added first.
After loading the data, distribution functions to describe the file size and the number of transfers were searched. The main Python packages used to analyse the data and find proper distribution functions were Matplotlib [Hun07] for visualisation, NumPy [HM+20] for efficient array operations, SciPy [VG+20] especially the SciPy stats module to represent the distribution functions, and distfit [Tas20] to compare various distribution functions.

Figure 4.12 shows the real world data from the monitoring system and the fitted exponential and exponentiated Weibull distribution functions from the SciPy package. The process of how the distribution functions were fitted and how the parameters for the simulation were configured will be explained in the following.

```
1 # rwd = [(bin1, count1), (bin2, count2), ...]
2 rwd = GetCsvValsFromFile('filesizes.csv')
3
4 # calculate bin width
5 rwdStepSize = rwd[1][0] - rwd[0][0]
6
7 # get a list of only the counts
8 counts = list(zip(*rwd))[1]
9
10 # assume sizes are uniformly distributed wihtin one bin
11 curBin = (rwdStepSize / 2) / ONE_GB
12
13 subRwds = []
14 for count in counts:
15     subRwds.extend([curBin] * int(count))
16     curBin += rwdStepSize / ONE_GB
17
18 distFit = scipy.stats.expon.fit(subRwds)
19 print('lambda = {}'.format(1/distFit[-1]))
```

Listing 4.7: Python snippet used to transform the file size distributtion real world data and calculate the parameters for an exponential distribution function

Since the fitting function of SciPy expects an array of data samples and not pre aggregated data like a histogram, the data from the monitoring system had to be transformed first. The most straightforward approach, for transforming the data, was the approximate recreation of the data samples from the histogram. Listing 4.7 shows the code used to transform the CSV data so that the data can be passed to SciPy for calculating the fitting parameters.
Line 2 uses the afore mentioned custom function to load the CSV data. By default, gaps are resolved and no filtering is applied. The returned format is a list of tuples. The first element of each tuple contains the bin value, e.g., the file size. The second element of each tuple contains the count of the bin, e.g., the number of files.
Line 5 calculates the bin width assuming the bins are equally spaced. As mentioned before, the file size histogram from the monitoring system has a bin width of 128 MB, thus `rwdStepSize` is 128 MB. Line 7 uses the zip function to transform the CSV data into a tuple with two lists. The first list contains all bin values. The second list contains all count values. The list with all count values is the required list for the subsequent code.
The next lines generate a list with a file size entry for each file. Assuming the number of files within each bin are uniformly distributed, the expected value for the

Distribution	RSS file size	RSS file size filtered	RSS num transfers
Erlang	0.99	0.85	20.23
Expo. Weibull	0.99	0.78	0.29
Gamma	1.79	0.97	13.20
Chi square	2.52	1.65	13.59
Alpha	2.64	2.00	1.34
Exponential	3.03	1.59	1.75
Beta	3.20	1.88	0.65
Cauchy	4.06	2.41	5.33
Normal	4.65	2.81	10.78
Uniform	5.44	3.52	32.38

Table 4.2: RSS score of various distribution functions fitted to the file size and number of transfers data using the distfit package. The RSS column shows the scores for the original data, while the RSS filtered column shows the scores for the data where the size of the first bin was halved to show the influence of the first bin on the fitting results.

first bin would be 64 MB. This is calculated in line 11 and transformed to gigabyte. Line 13 declares the list that will contain a file size entry for each file. Line 14 to 16 generate the file size entries. The loop is executed for each count value of the real world data. Line 15 extends the `subRwds` list by `count` entries with a value of `curBin`. Afterwards, the value for the next bin is calculated.
Finally, in line 18 and 19 the data are passed to SciPy and fitted to the exponential distribution function. The last element in the return value of the fit function is the scale parameter that corresponds to $1/\lambda$.

Based on the shape of the real world data shown in Figure 4.12, the exponential distribution was intuitively expected to be a good candidate to describe the real world data. However, measured by the residual sum of squares (RSS), several distributions showed good fitting results. The exponentiated Weibull distribution and the Erlang distribution delivered the best RSS values of approximately 1. The fitted exponential distribution resulted in an RSS value of approximately 3. Several other distributions delivered an RSS value between 1 and 3. Table 4.2 shows the

considered distribution functions and their corresponding RSS values.
For the file size, the main difference among the fitted distribution functions is how well they estimate the first bin. The first bin is an exceptional large bin containing the number of files with the smallest file size. In absolute numbers, real world data of approximately 932k files were available. The first bin counted approximately 180k files. This can be attributed to the fact that the first bin also counts the files with a file size of 0. For example, this might happen if the monitoring system was not fully able to record or transmit the file size.
Figure 4.12 shows that the exponentiated Weibull distribution estimates the first bin very well. However, it generally tends to underestimate afterwards. On the other hand, the exponential distribution clearly underestimates the first bin and tends to overestimate afterwards.
The influence of the first bin on the fitting results can be observed when manually limiting the size of the first bin, e.g., using a count of 90k files instead 180k. As shown in the RSS file size filtered column of Table 4.2, the RSS of the exponentiated Weibull fitting is reduced to approximately 0.78 while the RSS of the exponential distribution is reduced to approximately 1.6 when limiting the first bin. That means, different distribution functions could be used to better address different parts of the data.

```
1  "fileSizeCfg": {
2      "name": "BNLDataGen",
3      "storageElements": ["BNL-OSG2_DATADISK"],
4      "numFilesCfg": {
5          "type": "fixed",
6          "value": 1000
7      },
8      "fileSizeCfg": {
9          "type": "exponential",
10         "lambda": 0.61972,
11         "minCfg": {"type": "minClip", "limit": 0.009765625},
12         "maxCfg": {"type": "maxClip", "limit": 12.79106355}
13     },
14     "lifetimeCfg": {
15         "type": "fixed",
16         "value": 157680000
17     }
18 }
```

Listing 4.8: Primary properties of the used data generator configuration located in the profile configuration file

Once the parameters for a distribution function are calculated, the simulation

can be configured accordingly. The file size distribution must be configured in the data generator event. As explained in Section 4.1.3, the data generator event configurations are located in a list in the profile JSON file. For the validation scenario, this list contains three configuration objects, one for each storage element. Each configuration object will instantiate and configure a data generator object at runtime.

The most important configuration options of a data generator are shown in Listing 4.8. The only differences among the three configuration objects are the name, which is only used for console output and the storage elements, which specify where the simulated files should be generated. The values configured in the `storageElements` key must match the storage element names configured in the rucio.json file. Since no tick frequency is configured for the data generator events, the events are only executed a single time.

The `numFilesCfg` key allows specifying an value generator object as described in Section 4.1.3. This object is then used by the simulation to describe the number of files the generator should generate. For the validation, a fixed number of 1000 files was used.

The `fileSizeCfg` key allows specifying the value generator object describing the file size distribution. For this object, the parameters of the exponential distribution function fitted to the real world data are used. The limits were also taken from the real world data.

Finally, the `lifetimeCfg` key allows specifying a value generator for the lifetime of the file, i.e., the time after which the file should be deleted. However, since a fixed number of files was used, automatic deletion of files was not used by setting the lifetime larger than the simulated time frame.

```
1 "fileSizeCfg": {
2     "type": "expoweibull",
3     "a": 0.38797,
4     "c": 0.90383,
5     "l": 0.06399,
6     "minCfg": {"type": "minClip", "limit": 0.009765625},
7     "maxCfg": {"type": "maxClip", "limit": 12.79106355}
8 }
```

Listing 4.9: Alternative file size distribution configuration using the exponentiated Weibull distribution fit

The configuration system allows exchanging parameters without recompilation and without large effort. Listing 4.9 shows an alternative file size distribution configuration using the parameters of the exponentiated Weibull distribution fitted to

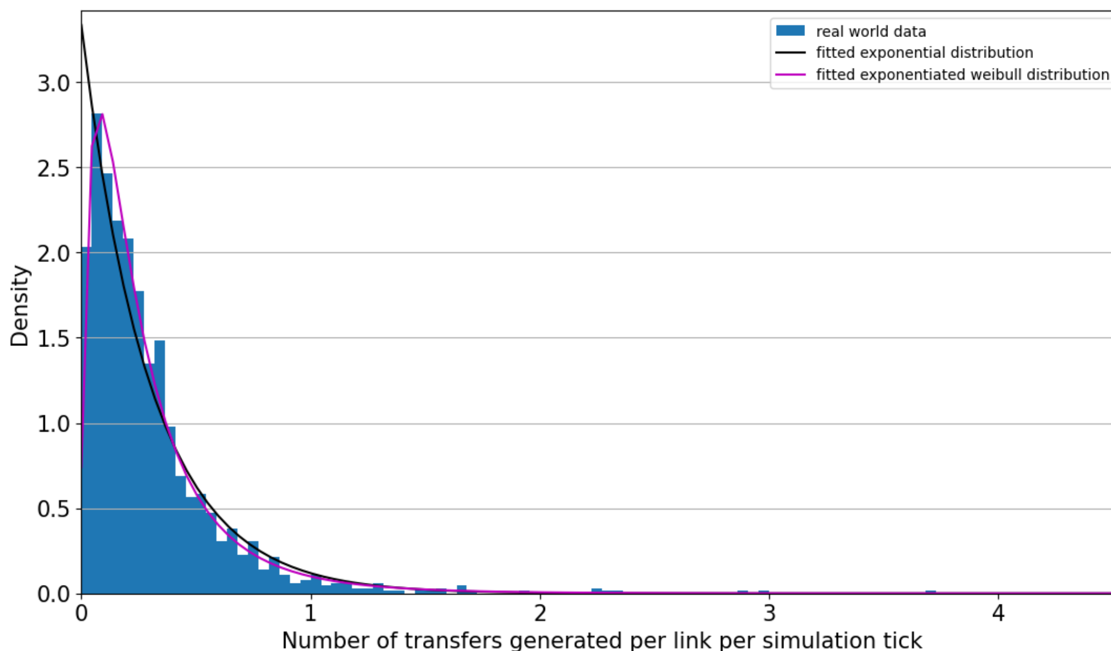

Figure 4.13: Number of transfers per hour from the monitoring data. These data were used to calculate parameters for the validation simulation.

the real world data. This allows running the simulation with different characteristics of the distribution functions. For example, the exponentiated Weibull distribution which better estimates the files with a very small file size as explained earlier.

Number of transfers parameter

The second parameter that was required for the simulation validation was the number of transfers to generate. As explained in Section 4.3.1, the `CFixedTransferGen` was used for this scenario. This transfer generator requires a value generator object configuration for each source/destination storage element combination. This value generator object is then used to determine the number of transfers that should be generated between the given combination.
In the following, the calculation of the configuration parameters and how the configuration was applied to the simulation will be explained in more detail.

Figure 4.13 shows the transformed real world data and the exponential distribution function used for the simulation scenario. Similar to the file size distribution, the exponential distribution was the first candidate chosen based on the shape of the real world data.
However, using the distfit package, the RSS score of several distribution functions was calculated. The RSS num transfers column in Table 4.2 shows the RSS values for the considered distribution functions. Again, the exponentiated Weibull distri-

bution shows the best score and is able to estimate the beginning of the real world data histogram better than the exponential function.

The process of calculating a distribution function for the number of transfers to generate was similar to the one for the file size distribution. The monitoring data were exported into another CSV file. However, for the number of transfers, the monitoring data were in the form of a date time histogram. That means the histogram contained one bin for each hour, with each bin describing the number of generated transfers during this hour. On the other hand, the desired histogram should show the frequency per number of generated transfers, i.e., one bin for each number of generated transfers, with each bin containing the frequency this number of transfers occurred. In addition, the monitoring data represented the sum of all done transfers among the three sites with the most transfers during the observed time.
For these reasons, the monitoring data had to be transformed before estimating the parameters. The monitoring data provided data of the transfers among three sites, each with two transfer directions, e.g., site A to site B and site B to site A. That means the data had to be distributed among 6 directions. For the validation, it was assumed that the number of transfers were equally distributed between the sites.

```
1 # rwd = [(bin1, count1), (bin2, count2), ...]
2 rwd = GetCsvValsFromFile('numTransfersPerHour.csv')
3
4 # calculate bin width ( 3600 = 1 hour )
5 rwdStepSize = rwd[1][0] - rwd[0][0]
6 simTickInterval = 10
7 ticksPerHour = int(rwdStepSize / simTickInterval)
8
9 # get only counts into a NumPy array
10 counts = np.array(rwd)[:, 1]
11
12 # equally distribute counts among 3 sites * 2 directions
13 counts = counts / 6
14
15 # transform: num/hour -> num/tick
16 counts = counts / ticksPerHour
17
18 distFit = scipy.stats.expon.fit(counts)
19 print('lambda = {}'.format(1/distFit[-1]))
```

Listing 4.10: Python snippet used to transform the real world data and calculate the parameters for an exponential distribution function

Listing 4.10 shows the Python snippet that was used to do the required trans-

formation of the real world data and estimate the parameter for an exponential distribution function. The snippet starts similar to the file size fitting snippet. In line 2 the helper function is used to read the CSV values. Line 5 calculates the bin width, which is 1 hour. Thus, `rwdStepSize` will equal to 3600 seconds. In line 6, the information is set that the transfer generator of the simulation will be executed every 10 seconds in simulation time. In line 7, this information is used to calculate the number of times the transfer generator will be executed per bin width, i.e., per hour. For example, `ticksPerHour` = 360 means that the transfer generator in the simulation will be executed 360 times per simulated hour.
Starting from line 10, the data transformation is done. Only the count values are required. Thus, the CSV data are loaded into a NumPy array and the column with the count values is extracted into the `counts` variable. Subsequently, in line 13, the number of transfers per hour is equally distributed to 6 transfer directions. Afterwards, by dividing by `ticksPerHour` in line 16, the time unit is transformed from count per hour to count per transfer generator execution.
After the transformation, the data can be passed to SciPy to calculate parameters for a random distribution. In line 18, the lambda parameter for the exponential distribution function is calculated and printed in line 19.
To fit the data to a different distribution function, only the last two lines must be changed. For example, replacing `scipy.stats.expon` by `scipy.stats.exponweib` would fit the data to the exponentiated Weibull function instead of the exponential function. The output must be changed to print all required parameters.

```
1 "generator": {
2     "type": "fixed",
3     "name": "FixedTransferGen",
4     "tickFreq": 10,
5     "startTick": 5,
6     "infos": [{
7         "storageElement": "BNL-OSG2_DATADISK",
8         "destinations": {
9             "CERN-PROD_DATADISK": {
10                "type": "exponential",
11                "lambda": 3.33437
12            },
13            "NDGF-T1_DATADISK": {
14                "type": "exponential",
15                "lambda": 3.33437
16            }
17        }
18    }]
19 }
```

Listing 4.11: Primary properties of the used transfer generator configuration located in the profile configuration file.

Listing 4.11 shows the most important part of the transfer generator configuration that was used for the simulation. As mentioned earlier, the number of transfers to generate is the main parameter for the transfer generator. The transfer generator can be configured in the profile configuration file. This is the same file where the data generators can be configured with the file size distribution, i.e., config/profiles/simEval/profile.json.

The profile configuration file contains the key `transferCfgs`. The value of this key is a list of objects. Each of these transfer config objects contains the `manager` key and the `generator` key, which can be used to specify the configuration of the transfer manager and the transfer generator, respectively. The generator will automatically use the transfer manager configured within the same transfer config object.

The manager config is not shown in the listing, but only contains four configuration options. The most important of it being the manager type and the execution frequency. The type defines whether the transfer manager updates the transfers based on the bandwidth or on a configurable transfer duration, as explained in Section 3.2.3. The execution frequency defines how frequent the transfer manager event is executed. The two other properties are the start tick, i.e., when the transfer manager is executed the first time and a name for the console output.

The validation scenario required a single transfer config object in the `transferCfgs`

list because one `CFixedTransferGen` generator is able to generate all transfers among the three sites. In line 2 of Listing 4.11 it is specified that the fixed transfer generator should be used. Line 3 to 5 configure the common properties that can be configured for every event.

Starting from line 6 the actual transfer configuration can be defined. The `infos` list contains one object for each storage element that provides a source for transfers. The listing only shows the configuration for the BNL-OSG2_DATADISK storage element as an example. The full configuration actually contains three objects in the `infos` list, one for each source storage element. As for the data generators, the provided storage element names must match the storage elements configured in the rucio.json file.

The `destinations` key of the source storage element allows specifying the destination storage elements. The format allows providing a value generator compatible configuration object for each destination storage element name. Since the parameter calculation assumed the transfers among the three storage elements were uniformly distributed, each destination is configured with the same exponential distribution parameters.

```
1 "DstStorageElement": {
2     "type": "expoweibull",
3     "a": 3.93496,
4     "c": 0.61761,
5     "l": 0.08224
6 }
```

Listing 4.12: Alternative configuration using the exponentiated Weibull distribution

As mentioned earlier, the configuration system allows changing the distributions and their parameters without a lot of effort and without recompiling. Since the exponentiated Weibull distribution showed very good fitting results, a second validation was done with using the exponentiated Weibull distribution for the number of transfers. The used parameter configuration for the second validation is exemplarily shown in Listing 4.12.

Throughput parameter

The last input parameter that was taken from real world data is the network link throughput. Similar to the sample of the number of transfers, the sample of the throughput from the real world data was available as a date time histogram. This histogram describes the mean throughput per hour among all three sites. Like with

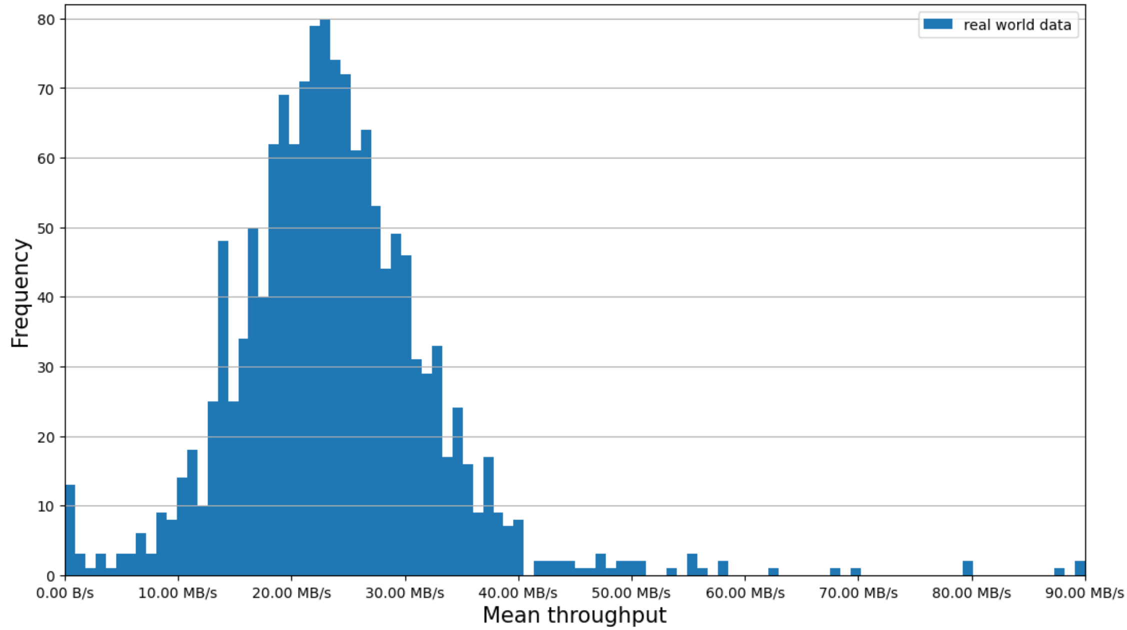

Figure 4.14: Histogram showing the frequency per mean throughput per day of the real world data. One data point was considered as an outlier and has been manually filtered from approximately 800 MB/s to 90 MB/s.

the sample of the number of transfers, the throughput data could be transformed to show the frequency per mean throughput. These transformed data are shown in Figure 4.14.

The shape of the real world data suggest the fitting of a normal distribution. However, as discussed in Section 3.3.2, the simulation accepts only fixed values for the bandwidth or throughput configuration currently. Thus, the mean value of the throughput real world data has been used, and the links were configured with a throughput instead of a bandwidth.

A straightforward approach to improve the modelling of the throughput data would be to use bandwidth information and a background noise generator. However, this approach would require the information through which network routes the files have been transferred in the real world. If this information was available, the bandwidth details of the corresponding network routes would be required additionally. Furthermore, the usage of a background noise generator would require a separate model and configuration. Those data are difficult to acquire and would contradict the requirement to simulate from a data management perspective. For this reason, the mean throughput value is used and further improving the network modelling capabilities is left subject to future work.

```
"CERN-PROD_DATADISK": {
    "NDGF-T1_DATADISK": {
        "throughput": 8105274,
        "maxActiveTransfers": 0,
        "receivingLink": {
            "throughput": 8105274,
            "maxActiveTransfers": 0
        }
    }
}
```

Listing 4.13: Part of the network link configuration located in the links configuration file.

Listing 4.13 shows a part of the network link configuration where the throughput parameter is set. As suggested in Section 4.1.3, the link configuration is specified in a separate file, which is also available in the GitHub repository [Weg21] at the path config/profiles/simEval/links.json.
The notation for the network link configuration is to provide an object for each source storage element. Each of these first level objects can contain keys corresponding to the names of storage elements that define the destination of a link. For example, in Listing 4.13 CERN-PROD_DATADISK specifies the source storage element. It provides an object containing the key NDGF-T1_DATADISK, which specifies the destination storage element. With this configuration, a network link from CERN-PROD_DATADISK towards NDGF-T1_DATADISK will be created by the simulation.
Furthermore, the destination storage element defines the `receivingLink` key. This key can be used to indicate to the simulation that another link in the opposite direction must be created with the provided properties. This allows conveniently creating links in both directions with different configurations.
The main properties of each network link configuration is the maximum number of active transfers. A limit above 0 means the transfer manager will keep transfers in the queue until the number of active transfers falls below the specified limit. No value or a value of 0 means an unlimited number of transfers can be active.
The last main property for a network link is the bandwidth or throughput. Using the `throughput` key leads to setting the network link into throughput mode, i.e., each transfer will be updated based on the throughput and time and independent of the number of active transfers. Using the `bandwidth` key instead leads to setting the network link into shared-bandwidth mode, i.e., the configured bandwidth will be divided by the number of active transfers.

4.3.3 Evaluation

The previous sections explained how the validation scenario was configured. Especially, the calculation of simulation parameters for the file size, the number of transfers, and the throughput based on real world data. As mentioned earlier, two more real world data samples were available, which are used only for the output comparison.
The simulation results that were used for the evaluation are the mean values of 10 different simulation runs. Using that number of runs, the standard error of each metric did not exceed 0.8 %. The standard deviation of each observed metric was below 0.06 %, except for the traffic and transfer duration metric. These metrics had a standard deviation of 2.5 %.
As mentioned before, the first validation evaluation used the exponential distribution to describe the file size distribution and the number of transfers to generate. However, since the exponentiated Weibull distribution showed even better fitting results, another validation run was executed using the exponentiated Weibull distribution for the estimation of the number of transfers to generate. This also allowed giving an impression of how the results may behave when changing input variables.
Since the extracted monitoring data were limited to three sites, an evaluation of a scenario with an arbitrary number of sites would not lead to a reasonable result for a validation. The first consideration for a scenario with more than three sites would be the network link configuration. The more sites, the more unrealistic becomes the scenario that each site generates the same number of transfers to each other site. This must be validated with further monitoring data. Another requirement would be the acquisition of sufficient monitoring data for the output metrics.
For these reasons, the validation scenario was limited to three sites. Removing this limitation would require model adjustments and significantly more monitoring data.

Table 4.3 shows a summary of the simulation results. The table allows comparing the values of all five metrics for the given real world data samples, the simulated data using the exponential distribution, and the simulated data using the exponentiated Weibull distribution for the number of transfers.
Noticeable is the difference of the number of transfers between the Sim and Sim Weibull column. Reconsidering the differences of the fitting results between the exponential and exponentiated Weibull distribution, the fit of the exponential distribution slightly underestimates after the first bin of the real world data histogram. Afterwards, the exponential distribution is more likely to generate larger numbers than the exponentiated Weibull distribution. This might result in the difference of this metric between the two validation runs.

Metric	RWD	Sim	Sim Weibull
File size	1.74 GB	1.74 GB	1.73 GB
No. transfers	0.177 No./s	0.180 No./s	0.171 No./s
Throughput	8.10 MB/s	8.11 MB/s	8.12 MB/s
Traffic	0.30 GB/s	0.31 GB/s	0.29 GB/s
Transfer duration	212.18 s	212.60 s	211.3 s

Table 4.3: Summary of the results of the simulation validation. The RWD column shows the real world values. The Sim column shows the simulated values using the exponential distribution for the file size and number of transfers. The Sim Weibull column shows the simulated values using the exponentiated Weibull distribution for the number of transfers.

As input parameter, the throughput is directly based on the real world data. It describes the mean value of the throughput of all transfers equally distributed to the three sites. Compared to the throughput, the traffic is the summed data volume that is transferred with each transfer manager update. Thus, the throughput metric is calculated per transfer, while the traffic is calculated for all transfers based on a time frame. The traffic is dependent on the file size and number of transfers.
The transfer duration is the mean duration of all transfers. It is calculated by taking the difference between the transfer start and end time. The transfer duration is dependent on the file size and throughput metric.

Traffic comparison

Figure 4.15 shows the mean traffic per day averaged for the 10 different simulation runs. The aura around the lines indicates the corresponding standard deviation of the simulation runs. Furthermore, the results are shown for the validation run using the exponential distribution and the one using the exponentiated Weibull distribution. This provides an impression of the influence of the different distribution function on the traffic metric.
As mentioned previously, the traffic value depends on the number of transfers. Fewer transfers mean less traffic. Table 4.3 already showed that the exponentiated Weibull distribution tends to generate fewer transfers than the exponential distribution. The results shown in Figure 4.15 confirm this dependency of the metrics.

Figure 4.16 shows the comparison of the traffic distribution between the real world

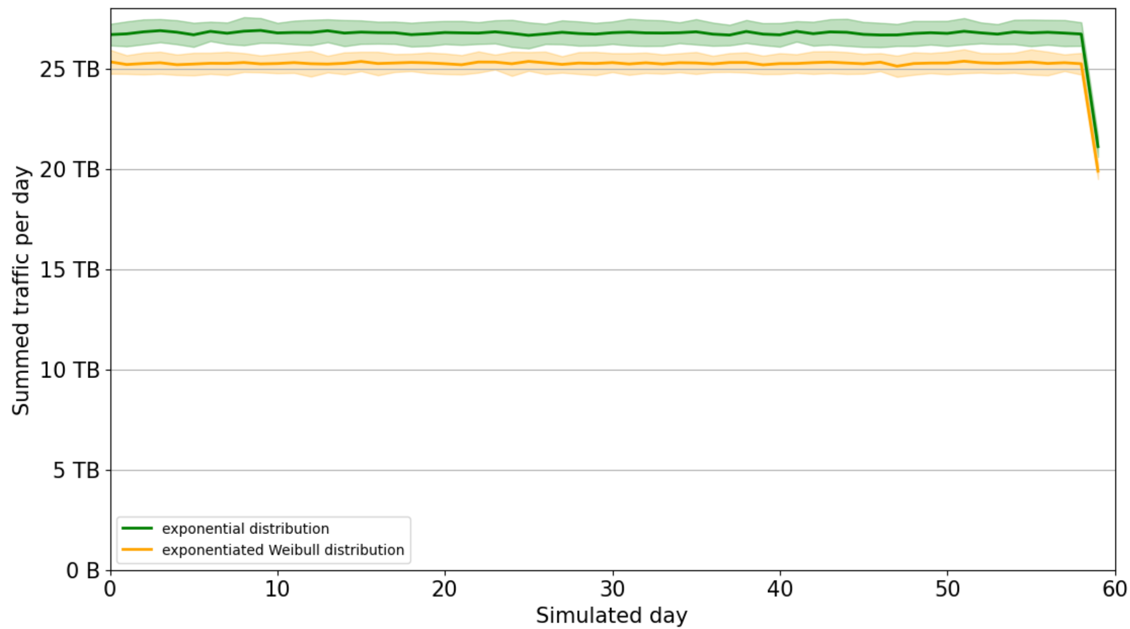

Figure 4.15: Comparison of the summed daily traffic between the two different validation runs. The lines indicate the mean of the summed traffic of multiple simulation runs. Each aura illustrates the standard deviation of the multiple simulation runs.

data and the simulation results. On the scale of GB/s the difference between the exponential distribution and exponentiated Weibull distribution is barely visible. But as shown previously, considering the traffic per day, the difference becomes clearer.

Compared to the real world data, the simulated data have a more steady shape, while the real world data show more fluctuations in the front and centre of the curve. However, the overall shapes of the real world and simulated data can at least be considered similar, which emphasises the significance of the mean values shown in Table 4.3.

Transfer duration comparison

Figure 4.17 allows comparing the difference of the mean transfer duration for the different number of transfers distribution functions. The data illustrated in the figure were generated by first calculating the mean transfer duration of all transfers per day. These mean values were taken from 10 simulation runs and averaged accordingly. The auras illustrate the standard deviation of the 10 simulation runs, and thus give an impression of the consistency of the simulation.

Furthermore, the figure allows comparing the influence of using the exponential distribution versus the exponentiated Weibull distribution. As expected, a clear

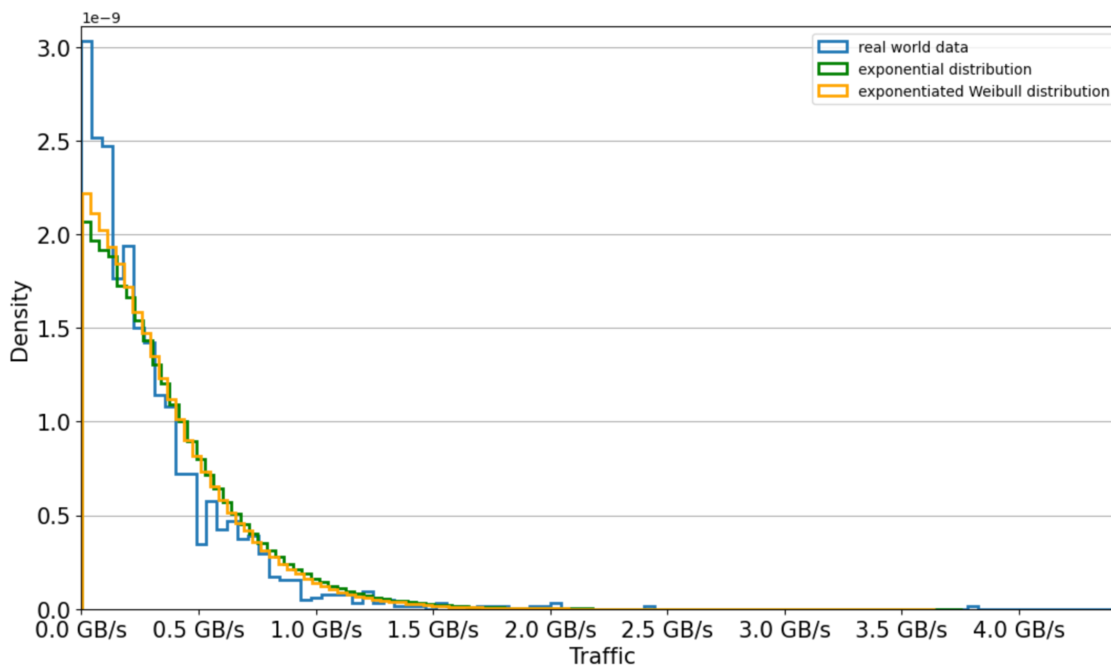

Figure 4.16: Comparison of the traffic distribution among the real world data and the two different validation simulation runs. The simulated data is the mean of multiple simulation runs.

difference is not visible. For the configure scenario, the transfer duration is not expected to be significantly impacted by the number of generated transfers.

Figure 4.18 shows the comparison of the transfer duration between the real world data and the simulation results. As expected, the difference between the results for the exponential and exponentiated Weibull distribution are barely visible. The real world data has slightly more fluctuations than the simulated data, but overall the shapes can be considered very similar. The main visible difference is that the simulated data has more noticeably more transfers with a very short transfer duration.

4.4 General Scalability Considerations

The scalability of GACS, in terms of runtime and memory consumption, largely depends on the implemented model. This makes it difficult to provide a specific statement on the scalability of GACS. However, there are certain operations and objects that are commonly used by model implementations, which can be evaluated in more detail. Especially, four metrics were considered relevant for the scalability of GACS: the number of events, the number of transfers, the number of files, and the number of replicas.

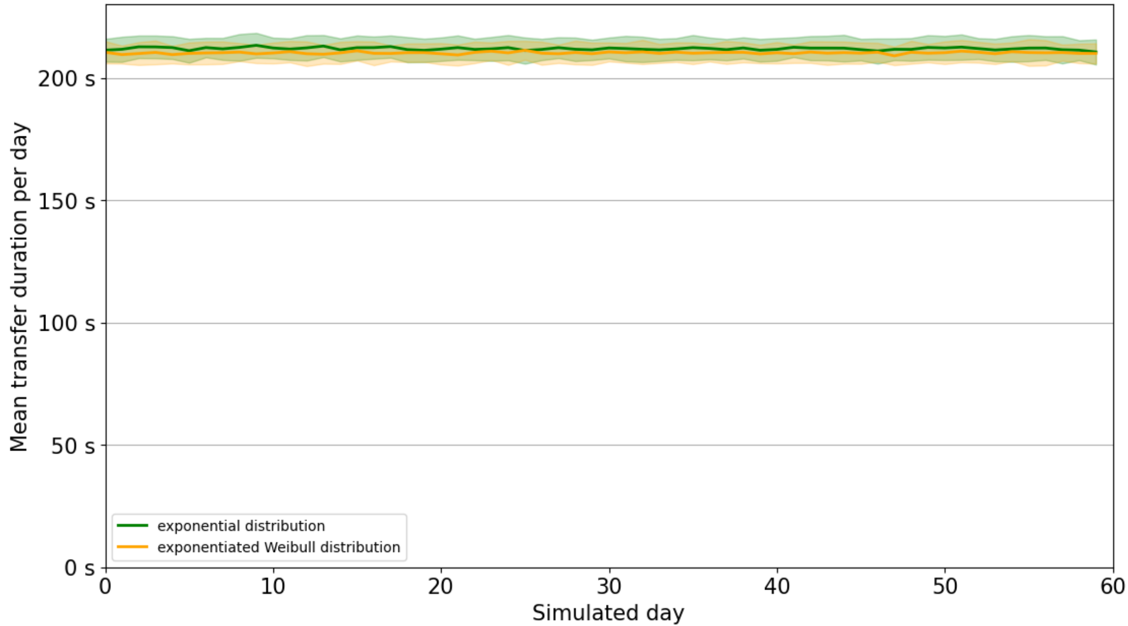

Figure 4.17: Comparison of the mean transfer duration per day between the two different validation runs. The lines indicate the mean transfer duration of all transfer per day of multiple simulation runs. Each aura indicates the standard deviation of the multiple simulation runs.

As explained in Section 3.3, the models implemented for this thesis use few, large events rather than numerous, small events. This makes the scalability of the implemented models less dependent on operations related to the schedule execution. However, since GACS also supports numerous events, the influence on the scalability must be considered.

The time complexity and memory consumption for events depend mainly on the user implementation. The parts that all events share are the insertion and extraction from the schedule. An event requires at least to store a scheduling time in order to be scheduled. In addition, a pointer to the event is required to be able to use the run time polymorphism. Since the events are only referenced in the schedule, this results in a linear memory requirement with at least 16 bytes for each event, assuming a 64-bit compilation.

Regarding the insertion and extraction from the schedule, there are two implementations available. First, using the STL priority queue adaptor as the schedule. For insertion and extraction, this results in a worst-case logarithmic time complexity based on the number of events in the schedule, i.e., in the priority queue. The implementation based on the priority queue does not allow searching for events and rescheduling of events before they are executed.

The second implementation uses a multi-set from the STL. This container is based

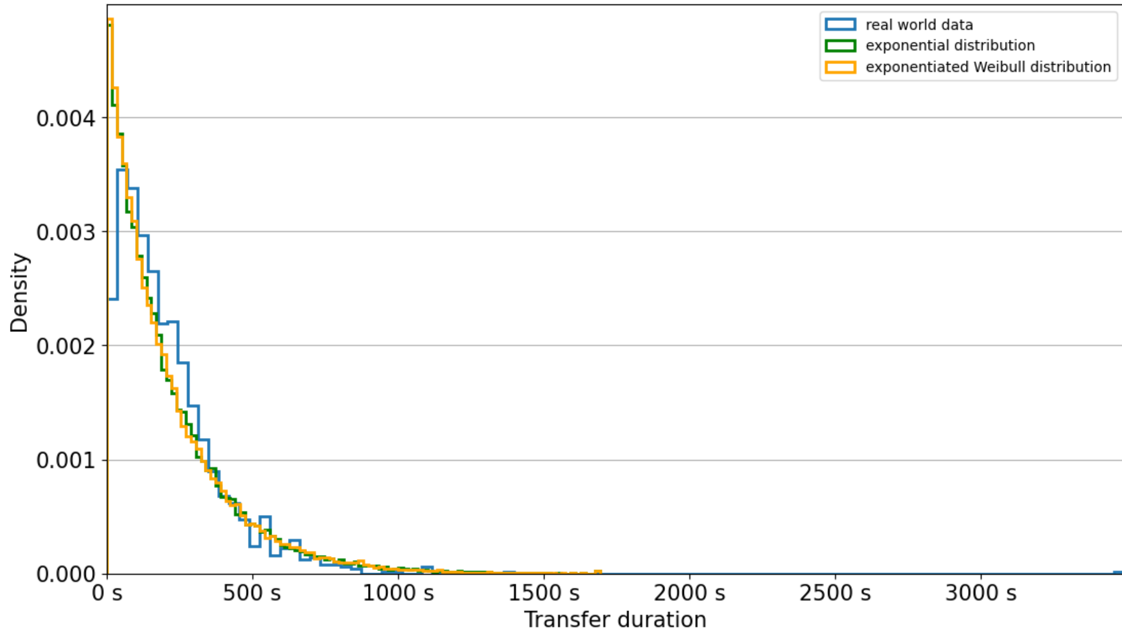

Figure 4.18: Comparison of the mean transfer duration distribution among the real world data and the two different validation simulation runs. The simulated data is the mean of multiple simulation runs.

on a red-black tree and allows keeping the events sorted by their scheduling time. The extraction of the next event is done in amortised constant time complexity, depending on how often the tree needs to be rebalanced. The insertion of an event requires logarithmic time complexity. Compared to the priority queue, the multi-set allows searching and rescheduling events. However, maintaining the tree structure introduces additional memory requirements.

In terms of the memory complexity, both solutions scale linearly in the number of events. Although, the multi-map approach has a slightly larger constant factor. The time complexities depend on whether the priority queue or the multi-set are used. The multi-set provides constant time complexity in the number of events for insertion, extraction, and searching.

The scalability of the number of transfers can vary depending on the used transfer manager. For the simulated scenarios, the default bandwidth-based transfer manager was used. In terms of memory requirements, each transfer is described by a 60 bytes large object. Moreover, a transfer object is only referenced in the transfer manager, resulting in a linear space complexity.

A straightforward approach of the default implemented transfer manager would be to store all transfer objects in a list and iterate through all transfer objects each time the transfer manager is executed. This would result in a linear time complexity in respect to the number of transfers.

However, the implemented approach uses two containers to separate the transfer objects and update them in two steps. The first container is a hash table that maps a network link to a list of transfer objects. The list for each network link contains the queued transfers that wait for activation based on the limit of active transfers of the network link. The second container is a multimap that maps a transfer start time to a transfer object that was activated.
The first step of each transfer manager execution is to activate queued transfer objects if possible. This is done by iterating through the network links and extracting as many queued transfers as possible using the associated list in the hash table. For each extracted transfer, the transfer start time is calculated, and the transfer is inserted into the multimap container.
The number of operations for the first step of the transfer manager update is limited by the number of network links and the limit of active transfers per network link. Each network link is visited once. For each network link, at most the number of transfers that can be activated is extracted from the list. Each extraction is a constant time complexity operation. For each extracted transfer, the insertion into the multimap costs logarithmic time complexity based on the total number of active transfers. The advantage of this solution is that the limit of active transfers is typically significantly smaller than the possible number of queued transfers.
The second step of each transfer manager execution is the updating of activated transfers. The multimap container was chosen because it allows keeping the transfer objects sorted by their starting time. This means not all activated transfers must be visited, but only the transfers that were started already. After a transfer is finished, the transfer is removed from the multimap, which requires amortised constant time complexity.
Thus, regarding the number of transfers metric, the overall memory complexity scales linear in the number of transfers. The time complexity scales linear with the limit of possible active transfers, which is assumed to be significantly smaller than the number of queued transfers.

The numbers of simulated files and replicas are the most relevant for common operations in GACS. Typically, they provide the largest share of the memory consumption. Each file object requires at least 76 bytes. In addition, a file object requires a reference to each of its replicas. Each replica object also requires exactly 76 bytes.
The file objects are stored in an unordered dynamic array in the Rucio object. The replica objects are stored in an unordered dynamic array in the corresponding storage element. Typically, the storage element prevents duplicated replicas. Thus, at

each storage element, a hash table based data structure contains an entry for each replica to prevent the duplication of a replica for a given file.

The most common operations that are influenced by the number of files and replicas are, the creation and deletion of files, the creation and deletion of replicas, and the index based selection of replicas. Generally, the creation of new files and replicas are implemented as insertions into the arrays, which requires amortised constant time complexity. For replicas, there are two additional operations. The insertion of a reference in the associated file object, which has an amortised constant time complexity. The insertion into the hash table, which has constant time complexity in the average case. However, the creation of new files and replicas also invokes corresponding action interfaces, which can introduce additional user-defined complexity.

The first step for the removal of a replica is to remove the reference to it in the file object. These references are stored in a dynamic array. For the removal of a replica, this results in a linear time complexity for searching the reference to remove. However, the number of replica references per file is typically in the order of a single digit. The actual removal from the array is done by swapping with the last element in the array and reducing the array size by one. The second step is to remove the replica from the hash table in the storage element. This has an average-case constant time complexity and a linear time complexity in the worst-case. The last step is to remove the replica from the array, which is also done with constant time complexity by swapping with the last element and decrementing the array size.

To remove a file, all replicas of the file are removed first. Afterwards, the file object is removed from the array in constant time complexity.

The last operation that is commonly used is the index based selection of replicas. For example, this is required when transfer generators randomly select their source replicas. Typically, a transfer generator maintains its own array of replica references. Index based access is mandatory for the selection. Furthermore, replicas can be added or removed. Thus, insertion of a new element as well as searching and removing an existing element is required.

Storing the references in an unordered array allows index based access and efficient insertion. However, searching requires linear time complexity. For this reason, GACS provides a data structure that uses an array of replicas and a hash table that maps a replica to the corresponding index in the array. This structure allows index based access using the array and searching of an arbitrary replica using the hash table. Inserting and removing elements has constant time complexity in the average case. However, the hash table introduces additional memory requirements.

In summary, the memory complexity scales linearly for both files and replicas. When neglecting the complexity of the action interfaces, the time complexity for creation of files and replicas scales constantly on average. The time complexity for the deletion of a file scales linearly in the number of replicas of this file. The deletion of a replica scales linear in the number of replicas using the same file. The index based selection of a replica has constant time complexity.

4.5 Run time scalability of the validation

The run time scalability was investigated in more detail using the validation scenario described in Section 4.3. The first tests aimed to investigate the behaviour if the number of sites increases. However, increasing the number of sites requires also to specify how these sites are connected and how transfers are created. Otherwise, the run time would not be significantly affected because sites are only passive objects. For this reason, the first approach was to continue the pattern of the validation scenario and connect all sites with each other. Since this results in a complete directed graph, there would be $N * (N - 1)$ links required for a configuration with N sites. Except the number of sites and the resulting number of network links, all other configuration options were equally adopted from the validation scenario. In other words, each link is used to create the same number of transfers using the same throughput as the validation scenario.

Figure 4.19 shows the results of the first run time test. The number of transfers and the number of links seems to grow linearly relative to each other, which is expected because the transfer generator uses the same number of transfer distribution for each network link. In contrast to that, the shape of the run time curve indicates that the growth rate of the run time increases at 30 sites and 60 sites. This indicates that the run time does not scale linearly with the number of transfers nor with the number of network links.
However, the increase of the growth rate still can be considered rather reasonable taking into account the simulated scenario. The scenario assumes a simulation of up to 80 sites, each site being connected with each other site, and a transfer generation rate comparable to the three largest grid sites.
Further investigations of the results showed that the largest share of the run time is spent in the transfer manager. The console output of the simulation showed that for all simulation runs, at least 85% of the run time is spent in the transfer manager. Compared to that, the transfer generator requires at most 13% of the run time.
Reconsidering the transfer manager configuration shows that the transfer manager

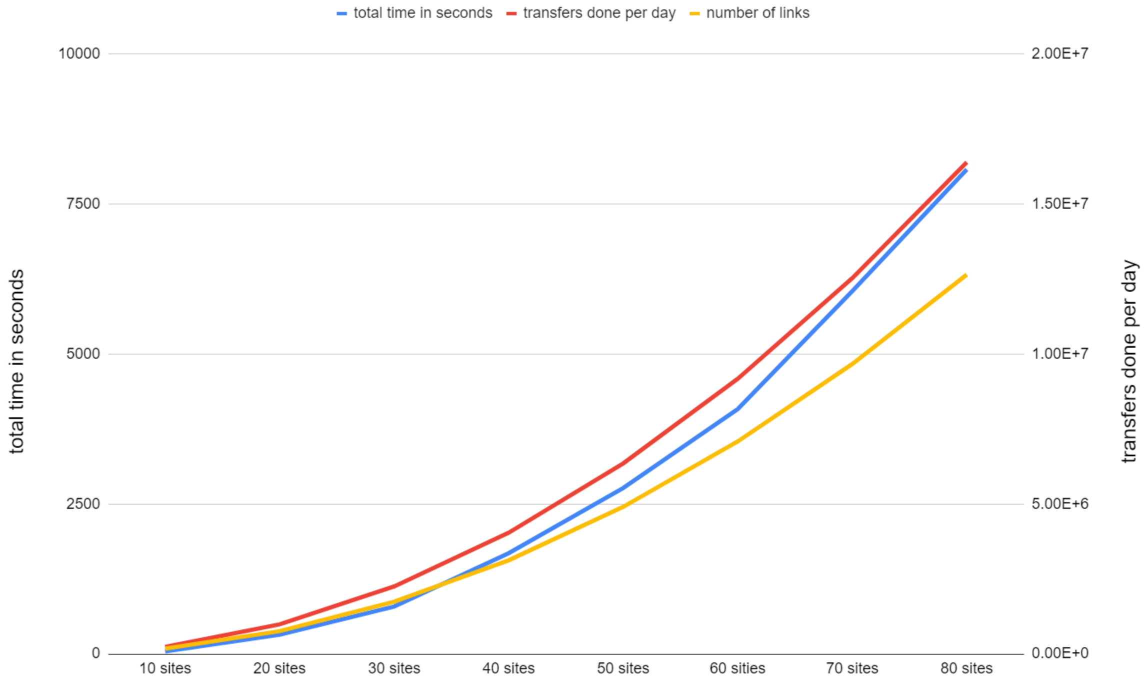

Figure 4.19: Results of the first run time test. The blue line belongs to the left axis and shows the mean overall run time of the simulation. The red line belongs to the right axis and shows the mean number of completed transfers per day. The yellow line indicates the increase of number of network links, which can be calculated with $N * (N - 1)$ for N sites.

is configured with a tick frequency of 1 simulated second, while the transfer generator only ticks every 10 simulated seconds. In the case that the transfer manager introduces an unacceptable run time limit, an option would be to reduce the tick frequency. However, this comes with the cost of a reduced simulation accuracy.
For example, using a tick frequency of 10 seconds instead of 1 second for the transfer manager, the same simulation scenario requires approximately 55% of the run time for the transfer manager and 40% for the transfer generator. The required total run time of simulation runs using 50 sites is reduced from approximately 2760 seconds to 640 seconds. However, the influence on the accuracy must be considered. For the given scenario, the worst case error would be presumably 10 seconds around the transfer duration.

The configuration of the scenario used for the first run time test was straightforward. However, the approach of how the sites are connected is not very realistic considering the large number of transfers generated for each network link. In Section 4.3.3, it was already discussed that the main concern about increasing the number of sites is the consideration of how the sites should be connected.
In the WLCG, the sites are more used in a hierarchical approach, e.g., the Tier 0

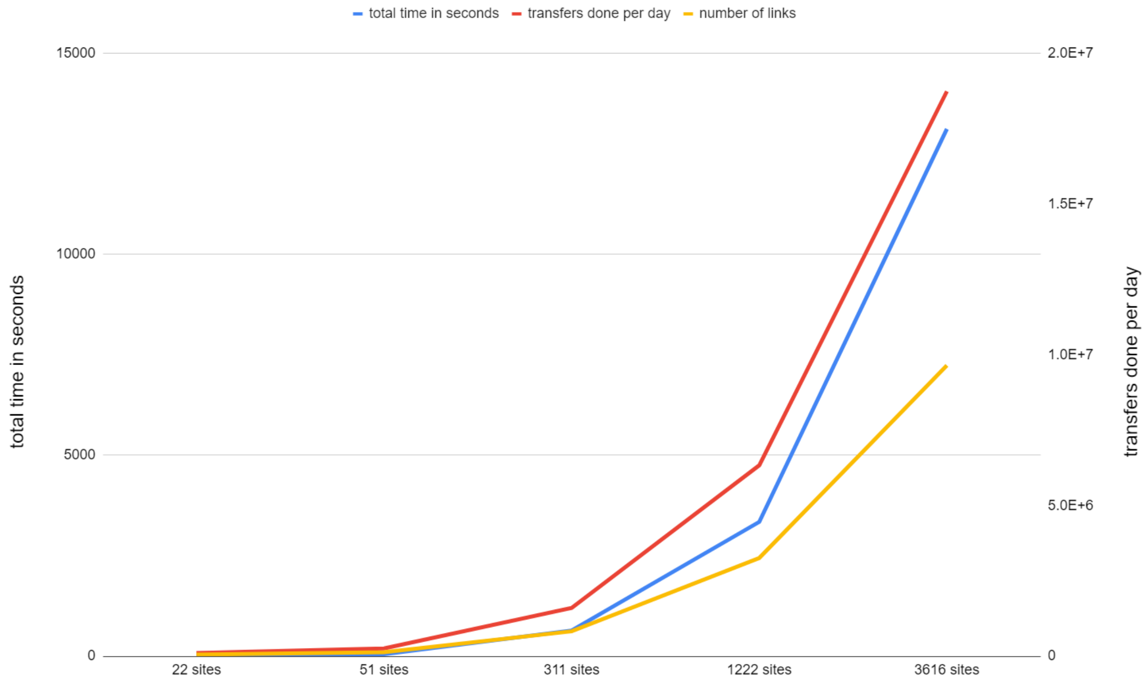

Figure 4.20: Results of the second run time test. The blue line belongs to the left axis and shows the mean overall run time of the simulation. The red line belongs to the right axis and shows the mean number of completed transfers per day. The yellow line indicates the increase of number of network links.

site exchanges most transfers with the Tier 1 sites one level below. Further, the Tier 1 sites complete numerous transfers with a set of Tier 2 sites, and so on. A second run time test tried to adapt this model by connecting the sites in a hierarchical approach. However, this introduces two new concerns. First, it must be decided how many sites of each tier exist. Second, it is not as easily possible to set the number of total sites as with the previous test because the total number of sites is a result of the sum of the children of each tier.

Figure 4.20 shows the results of the second run time test. For this test, the number of sites was not set explicitly. Instead, a number of Tier 1 sites was defined, a number of Tier 2 sites per Tier 1 site, and a number of Tier 3 sites per Tier 2. This represents a completely different scenario compared to the validation, but the approach of how sites are connected is more realistic and allows simulating up to 3616 sites in a reasonable amount of time.
The yellow line indicates the number of network links. Comparing the increase rate of the number of links to the run time shows that the run time still increases with a higher rate than the number of links.

5 HCDC model

This chapter explains the HCDC model as the combination of the Data Carousel model and Hot/Cold Storage model. First, an overview is given, elaborating on the motivation and possible variations of combining the models. Afterwards, the implementation of the HCDC model into GACS is explained. Mainly, this includes the description of the used configuration, the implementation of the transfer generator, and the used parameters. Finally, the evaluation method as well as the results of the evaluation are presented.

5.1 Overview

The first part of this section states the details about the motivation behind the creation of the HCDC model. Furthermore, it explains the model in detail and which variations are possible for certain parts of the model.

5.1.1 Motivation

In general, data-intensive computing workflows require the use of performance-oriented storage, such as disks, because of the lower response time and better performance in concurrent and random access mode. This often results in maintaining at least one permanent copy of each input file on a disk system.

These workflows could be scheduled to be infrequently executed in bulk processing campaigns, e.g., whenever a new derivation software version is released. In this way, input files have to be read only once per campaign. Given this case, the input data would be preferably stored solely on tape. However, for larger collaborations, such as ATLAS, it is challenging to optimally organise those workflows into campaigns entirely. For this reason, there are continuous derivation workflows that read input data more frequently.

To allow a continuous derivation workflow to have a proper throughput of input data, a common solution is to keep at least one copy of the vast majority of input files on a disk storage system. For example, almost all input data for the production of derivation data for ATLAS have one permanent copy on both tape and disk

storage.

As explained in Section 2.3.1, the Data Carousel model implements an approach that tries to take advantage of the infrequent usage of the input data for derivation production campaigns. The concept of the Data Carousel model is to transfer the input data from tape storage to disk storage, start processing the data, and continuously replace the data that has been processed by new data coming from tape.
Using the Data Carousel model, only a limited number of input files are required on disk storage at any one time. This allows removing the permanent copies of the input files from the disk storage system and storing the input files solely on tape storage. In this way, disk storage requirements are reduced to save cost or provide disk storage for other types of data.
The Data Carousel model was developed to improve storage usage and tape performance if the derivation workload is structured into campaigns so that the input files are required only once per campaign. For the continuous derivation workflow, the input files are accessed frequently which would result in using the tape storage in concurrent random access mode and thus, significantly reduce the tape performance.
To allow using the Data Carousel model with continuous workflows, it can be extended by the Hot/Cold Storage model to the HCDC model. The HCDC model aims at minimising the disk storage required for derivation campaigns as achieved by the Data Carousel model, while mitigating the negative impact on the tape storage performance for continuous derivation workflows.

5.1.2 Model Variations

The HCDC model combines the Data Carousel model with the Hot/Cold Storage model. These models were explained in Section 2.3. The Hot/Cold Storage model adds the cold storage layer between the tape and the disk storage. The combination of the two models can be done in various configurations. The main differences among possible variations are explained in this section. The variation that was evaluated is explained in the next section.
First, the HCDC model can be considered for different workflows. However, for this work the model was designed and configured with respect to the ATLAS derivation workflow, which is explained in Section 2.1.1. Using the HCDC model for the derivation workflow provides significant disk storage saving potential, e.g., in the case that all AODs could be migrated to tape storage. In addition, the data access pattern and the input file size distribution of derivation workflows were considered promising.

The HCDC model considers two sources of storage resources. First, on-premises

storage, which is provided by the institution or company using the model. On-premises storage is considered to be permanently available, i.e., it is not required to be allocated on demand. For example, the institutions that are associated with the ATLAS experiment provide a pledged amount of resources for a certain time period. This means there is a defined limit of resources, and whether the resources are fully used or partly used does not unduly affect the cost. On-premises storage costs are mainly based on the hardware acquisition, maintenance, and administration.
The second source of storage resources are commercial cloud providers. These resources are considered under typical cloud behaviour, i.e., they can be allocated and deallocated on demand to an arbitrary extent. However, depending on the usage of commercial cloud resources, different kinds of costs are induced. Regarding storage resources, there are at least the cost for the stored volume per time, which depend on the storage type. Furthermore, there are typically costs for operations like writing, reading, deleting, or changing metadata of files. Another cost factor is the network traffic. Usually, cloud providers only charge egress and traffic between different regions within the cloud. The amount charged for traffic within the same cloud depends on the source and destination endpoints. Egress traffic out of the cloud to the internet is the most expensive.

As mentioned before, the derivation workflow can be executed continuously or in organised campaigns. The HCDC model can be used for both workflow types. A first consideration when specifying the model implementation should be whether it is based on bulked processing campaigns or executed continuously.
Organised campaigns result in an infrequent, predictable requirement of the input data. This corresponds to the requirements of the Data Carousel model because the data access pattern performs well in combination with tape storage. Hence, the Data Carousel model part of the HCDC model is expected to be most effective, whereas the Hot/Cold Storage part should be less impactful. In campaign processing, the potential benefit from the Hot/Cold Storage model part would be to use the cold storage as a buffer and prefetching area. This avoids that the performance of the tape storage becomes a bottleneck after the start of the campaign.
Continuous derivation production leads to a more frequent and less predictable demand for the input data. This reduces the potential performance of tape storage, which reduces the impact of the Data Carousel model part. However, the Hot/Cold Storage part should become more important in providing a cache for the processed data and reduce the data access on tape storage. Another point to consider is which storage category of the Hot/Cold Storage model part is implemented by the cloud storage. This depends on the different storage types offered by the cloud

	Hot	Cold	Archival
Disk	Reduces WLCG disk storage required; With WLCG computing: high performance, no extra network cost	Flexible volume requirement; Most effective with GCS hot storage and GCE computing	Large amount of most infrequently accessed data; No optimal use of disk storage.
GCS	Highest cost/volume; With GCE computing: extra compute cost, reduced network cost; With WLCG computing: high egress cost	GCS suits the flexible volume requirement; Egress cost depend on the recall rate	Largest amount of cloud storage; Storage required permanently; Cheaper cost/volume ratio
Tape	Unsuitable because of high access latency and insufficient performance	Flexible volume requirement; Potentially throughput bottleneck depending on recall rate	Optimal use of tape storage for the requirements of archival data

Table 5.1: Possible mappings from the storage types to the corresponding storage categories of the model.

provider and should be decided based on the QoS properties described in Section 2.3.2. Table 5.1 compares each of the different storage types with the corresponding model category.

Using the cloud storage as an archive, would require the permanent acquisition of cloud storage for the largest amount of data among all three storage categories. Some cloud providers offer different storage types with different pricing policies. For example, GCS offers storage types with a reduced cost per volume ratio, but with additional cost for each data access. Since the archival storage category is used for the least frequently accessed data, the additional access cost could be negligible and the reduced cost per volume ratio could be beneficial.

Using the cloud storage as hot storage, would reduce the required cloud storage volume. Since hot storage requires high performance storage, the cost per volume for the storage type would typically be higher. Furthermore, the egress cost could be

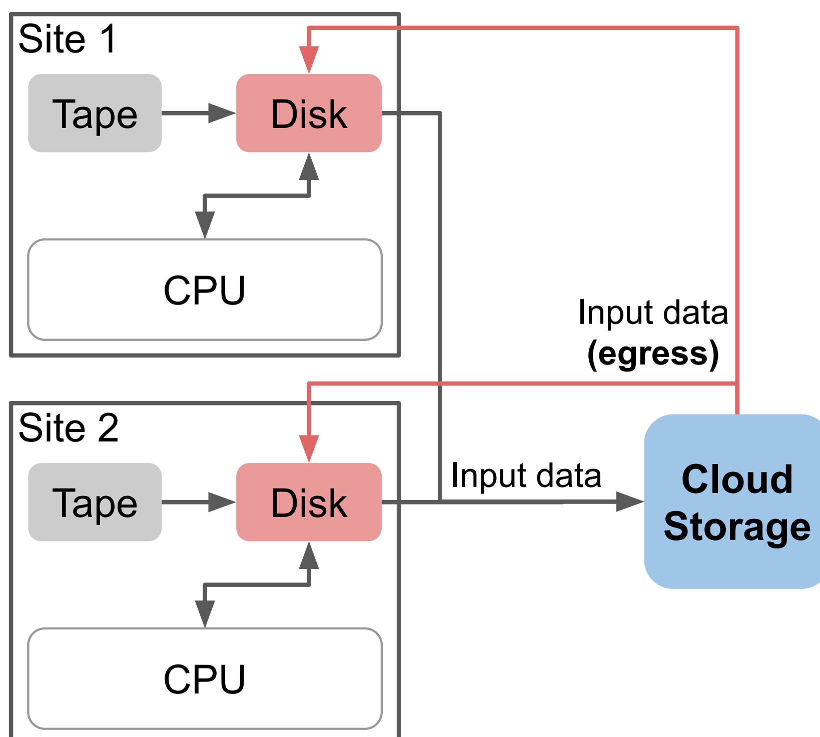

Figure 5.1: Schematic of the combination of the Hot/Cold Storage model and the Data Carousel model with two sites. The on-premises tape storage provides the archival storage, the on-premises disk storage provides the hot storage, and the commercial cloud storage provides the cold storage.

very high if the data is processed outside the cloud because the hot storage contains the most popular data.
Cold storage has a more flexible volume requirement, depending on the popularity metric and the available hot storage. With cloud storage as cold storage, the egress cost would depend on the number of reusages of the data and the available amount of hot storage.
Using the cloud for cold storage enables the sliding window of the Data Carousel model to become dynamic. This means that, in the case that the hot storage is full, the sliding window can be extended by using cold storage to keep an optimal performance of the archival storage.

Figure 5.1 illustrates a possible combination of the Hot/Cold Storage model and the Data Carousel model. The storage categories of the Hot/Cold Storage model part are assigned straightforwardly using on-premises tape storage as archival storage, on-premises disk storage as hot storage, and commercial cloud storage as cold storage. This particular combination is investigated in more detail throughout this

chapter.
It is assumed that at least one replica of all input data is available at the tape storage. In the evaluated model, the tape storage is considered to be read only, i.e., data is not added nor removed to the initial existing data at tape. This assumes that adding new derivation input data to the tape storage takes place in a separate phase.
The CPU box in the graphic represents the computing nodes with their local storage areas. To process input data, the data need to be transferred to the local computing node storage. To allow the computing nodes downloading input data to their local storage, the input data must be available at the disk storage of the associated site. The output data is uploaded to a disk storage area of the site.
Input data that are not available on disk storage can be transferred from the tape storage or from the cloud storage if available. A transfer from the tape storage potentially requires more time than from the cloud storage. However, transferring from the cloud storage induces egress cost. As mentioned before, the input volume for derivation workflows is typically larger than the output. Thus, the egress cost could become significant.
When input data are no longer required on the disk storage, they are migrated to the cold storage prior to their deletion on the disk storage. The ingress is assumed to be free of charge. The deletion of the data at the cloud storage could be implemented either with an expiration time or based on a storage limit and the popularity metric.

5.2 Simulation Implementation

This section describes the implementation of the HCDC model in GACS. Since the HCDC model defines special rules of how data is migrated, the HCDC transfer generator was implemented in GACS. The transfer generator is configurable with several parameters and implements the behaviour of the HCDC model. Furthermore, the HCDC transfer generator requires a certain configuration of the storage and network infrastructure. For example, at least one hot, cold, and archival storage element must exist.
This section starts by describing the used configuration of the storage and network infrastructure. Mainly, this includes the properties of sites, storage elements, cloud buckets, and network connections. Afterwards, the implementation of the HCDC transfer generator is described in more detail. Then, the parameters used to simulate the model are explained. Last, a short impression of the consumed memory and the required run time is given.

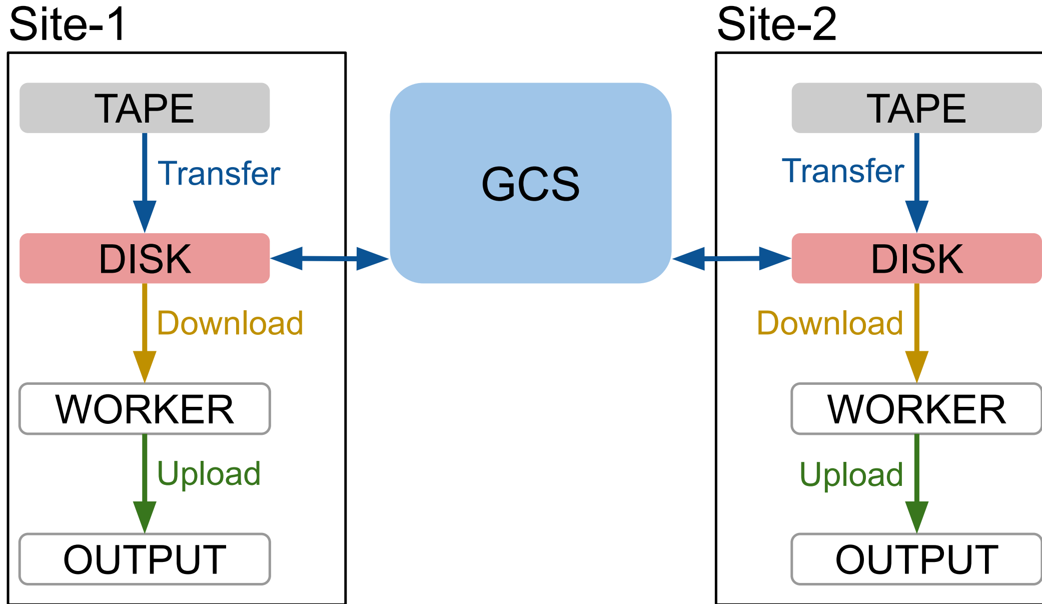

Figure 5.2: Implementation of the HCDC model in the simulation. It shows the configured storage elements for each of both sites and the network link setup. The network link labels name the type of transfer. GCS illustrates a single cloud bucket used by both sites.

5.2.1 Infrastructure Configuration

Figure 5.2 shows the used storage and network configuration for the simulation of the HCDC model. It is based on two grid sites Site-1 and Site-2 each with four different storage elements. The simulation uses one transfer generator instance for each site, i.e., one HCDC transfer generator instance is associated to exactly one site. The HCDC transfer generator requires that each site associated with an HCDC transfer generator provides at least the four illustrated storage elements and the illustrated network connections. In addition, at least one globally accessible cold storage element is required.

The used globally accessible cold storage element is implemented as GCS bucket and represented by the GCS box. In the simulated scenarios, each site uses only the data originating from its own tape storage element, although both sites have access to all the data on GCS. To enable sites processing data from other sites, a study would be required of how to split the workload exactly. A possible approach would be to use a given share, e.g., 80% of the jobs use local site data and 20% of the jobs use remote site data. Another approach could use the popularity metric to additionally select remote site data.

In the following, the purpose of each of the four storage element types is explained.

TAPE storage elements provide tape storage that represents the archival storage of the Hot/Cold Storage model. The tape storage contains one replica of each input file, and thus represents the origin of all input files. Typically, requests to tape are kept in a queue for some time, if the tape is not mounted, to optimise the reading requests. Additionally, there is a latency for mounting, positioning, and dismounting the tape. These delays are simulated by configuring the tape storage elements with an access latency. That means when a queued transfer from tape storage becomes active, the start of the actual data transferring is deferred based on the access latency.

DISK storage elements provide disk storage exclusively for input data and represent the hot storage of the Hot/Cold Storage model. The disk storage is used as source for the downloads of the derivation production input files to the worker nodes. Only files on disk storage elements can be downloaded to the worker nodes. The disk storage represents the storage area of the sliding window in terms of the Data Carousel model.

WORKER storage elements are used to simulate the local storage of the worker nodes. The simulation was designed to represent transfers with storage and network resources, so there is no default functionality for computing resources. It is assumed that the input files always fit entirely on the worker node. Thus, worker storage elements must not have a limit set. In general, the HCDC model can also be used without this assumption, but the transfer generator implementation would require several adjustments, e.g., to support streamed data input.

OUTPUT storage elements that represent a storage area exclusively for output data. Output storage elements are used to store the output files of the jobs which will be uploaded from the worker node storage.

GCS buckets are a special type of storage element used to represent the cold storage of the Hot/Cold Storage model. The cloud module of the simulation contains a GCS implementation. This implementation allows creating GCS buckets. GCS buckets are storage elements that are extended by certain functionalities like storage increase/decrease tracking, ingress/egress tracking, and cost calculation. These functionalities implement the cost model of the cloud provider.

As in the real world system explained in Section 2.1.3, the simulation also uses transfers, downloads, and uploads to copy data. The simulation implements downloads and uploads as a special type of transfer. The transfers from disk storage elements to worker storage elements are called download instead of transfer. The transfers from the worker storage element to the output storage element are called upload instead of transfer. The difference between a download and a transfer is that

the downloaded replica is not managed by the data management system. Respectively, an upload creates a new managed replica in the data management system. Furthermore, in the simulation downloads and uploads are not processed by the transfer manager, and they are stored in a different format in the output module.

5.2.2 Transfer Generator Implemenntation

As explained in Section 3.2.3, a transfer generator implements the main part of a model in the simulation. For this chapter, the HCDC transfer generator is implemented and configured to simulate the continuous production of derivation data. This task can be split in several steps:

1. A certain amount of input data are selected for processing.
2. If necessary, transfers are created to make the data available at the disk storage element.
3. The data are downloaded to the worker storage.
4. The data are processed at the worker node.
5. The output data are generated and uploaded to the output storage element.
6. All input data that are no longer required at the disk storage element are transferred to the GCS and deleted at the disk storage.

Since the transfer generator is implemented as a simulation event, these steps are executed regularly based on a configured frequency until the simulation stops.

To keep track of the different steps for each selected input file, a state object is used. These objects transit through states similar to the states of jobs in the Pilot system explained in Section 2.1.3. Moreover, the state objects represent similar information. For these reasons, the state objects are referred to as *job object*. The first step, the selection of input data, is called job submission. In this step, a new job object is created for each selected set of input data.
The HCDC transfer generator keeps track of all job objects and transits them through a state machine. Technically, this is implemented by using one doubly linked list for each state. A doubly linked list is used because for some states it is required to maintain an order for the list. Thus, it is required to be able to insert elements in front or behind an arbitrary element in the list. Furthermore, the singly linked list of the STL does not provide a way to add an element to the end of the

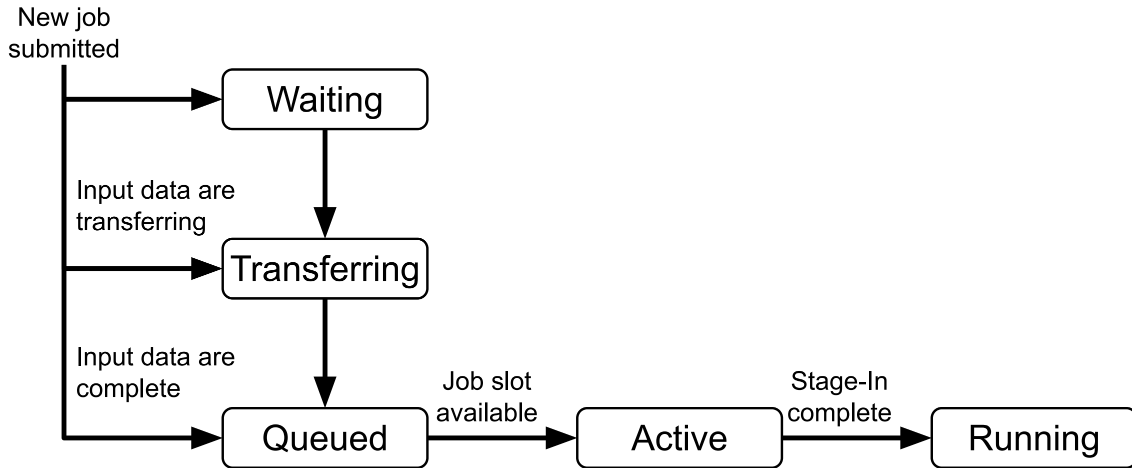

Figure 5.3: State transitioning of jobs during production phase.

list.
To represent the state machine using the lists, a job object is stored in exactly one of the state lists at any time. The transfer generator updates state after state by iterating through the lists and updating each job object. Job objects transit into other states by migrating the object from one list to another.

Figure 5.3 illustrates the state machine implemented by the HCDC transfer generator. Each job object is in one of the states waiting, transferring, queued, active, or running. The arrows indicate the transition among the states. The arrow labels are the conditions for a state transition.
As mentioned before, the first step is the submission of new job objects. The first step is calculating the number of jobs to generate. This is done by using a configured random distribution. Afterwards, for each job to generate, a file is randomly selected based on the popularity metric. Each selected input file is associated with a new job object, which is created and registered in the transfer generator. The initial state of a job object depends on the availability of the input data. In the following, the various states and transitions are explained in more detail.
Waiting: The required input data is not available at the disk storage element. No transfer for the input data to the disk storage element exists. Available disk storage is insufficient to create a transfer of the required input data to the disk storage.
The objects in the waiting state are not all actively updated. When sufficient storage at the disk storage element becomes available, exactly as many job objects from the waiting state list are selected as possible considering the available storage. For each of the selected job objects, a transfer is created. If available, data is transferred from GCS. Otherwise, the data must be transferred from the tape storage element. After transfer creation, the job object and all potential other job objects that were

waiting for this input data are moved to the transferring state. Job objects in the waiting state are processed in first in, first out order.
Transferring: Jobs are in this state when the required input data is not completely available at the disk storage element, but a transfer is queued or running to replicate the data to the disk storage element.
Updating the transferring state is implemented by using an action interface. The HCDC transfer generator implements the `PostCompleteReplica` method of the storage element action interface. This method is called when the replica becomes complete at the disk storage element. The method uses a hash table to find the job objects that are waiting for the completed replica and transits the job objects into the queued state.
Queued: When at the time of the job submission the input data are already at the storage element, the job directly enters the queued state. Otherwise, the job enters the queued state as soon as a transfer for the input data is completed. In the queued state, the job waits for compute resources in the form of job slots to become available. The simulation can be configured to provide a specific number of job slots per site. If a job slot is available, the job object state changes to active.
The queued state is updated by calculating the number of free job slots and then migrating the corresponding number of job objects from the queued job list to the active job list.
Active: In this state, a job is occupying a job slot. A download of the input data from the disk storage element to the worker storage element is running. When the download is finished, the details about the download and the job object are stored to the simulation output by using the output module. Afterwards, the simulated duration of the derivation job is calculated using a configured random distribution and the job is moved to the running state.
Running: The job object simulates the derivation job execution. Job objects in the running state are not regularly updated. Instead, the job objects are sorted by their finishing time, which can be calculated using the calculated derivation job duration. Then, the current simulation time can be used to select the finished running jobs. For each finished job, output replicas are created and uploaded to the output storage element. The job object is deleted after the upload is completed.

The last part of the transfer generator is the deletion of data. Two types of deletion must be considered, depending on the underlying storage. First, the deletion of data at the disk storage element. When the last job associated with a given replica is finished, the replica is marked for deletion. At the next transfer generator update, all replicas marked for deletion are transferred to the GCS, if they do not exist already.

Parameter	Value/Configuration
Simulated time	90 days
Transfer mgr. update frequency	1 s
Transfer gen. update frequency	10 s
No. sites	2
No. initial replicas	10^6 per site
Popularity	geometrically distributed:
	$p = 0.1$
	$1 \leq x < 50$
Input file size	exponentially distributed:
	$\lambda = 0.026$
	9.76 MB $\leq$ size $\leq$ 134 GB
No. jobs submitted	normally distributed:
	$\mu = 0.63366$
	$\sigma = 0.37292$
	$n \geq 0$
Job duration	exponentially distributed:
	$\lambda = 0.00409$
	$t \geq 16.666$ minutes

Table 5.2: Parameters and their values for the simulation of the HCDC model.

After the transfer is complete, the replicas are deleted from the disk storage. To be able to successfully transfer the replicas to GCS, either GCS must be unlimited or data at GCS must regularly be deleted. The deletion of GCS data is the second type of deletion. If GCS storage is limited and data must be deleted in order to transfer new data, the data with the lowest popularity is deleted until sufficient storage is available.

5.2.3 Used Parameters

Table 5.2 shows the general and site specific parameter values used for the simulation of the HCDC model. The values were calculated using real world monitoring data. The monitoring data were taken from different systems. Data to calculate the bandwidth were taken from the transfer monitoring system. The other parame-

ters such as input file size distribution were taken from the job monitoring system. Contrary to the transfer monitoring data, the job monitoring data were not limited to the past two months. However, since the mean bandwidth is calculated from the data, it is assumed that the mean value is also representative for the time of three months. The monitoring data for two months were taken for the time from 2020-07-08 12:00:00 to 2020-09-06 12:00:00. The monitoring data for three months were taken for the time from 2020-06-08 12:00:00 to 2020-09-06 12:00:00.

First real world tests with the HCDC model would be limited to rather small-scale deployments, especially in terms of the number of sites. This prevents a large wasting of resources in case of issues with the implementation. Additionally, it allows acquiring an impression of the model without incurring excessively large cloud costs. The monitoring data showed that in total 80 sites processed ≈ 6.5 million derivation jobs during the observed three months. The two sites with the largest number of derivation jobs each processed ≈ 0.5 million jobs. To be able to use the same job submission configuration and to keep the first evaluation of the model clear, only these two sites were simulated. The simulated time was set to 90 days according to the monitoring data.
The required real time to simulate one day was typically below one second. Thus, it was possible to use a value of one second for the transfer manager update frequency to achieve the best possible precision. The vast majority of the simulation run time is spent in the transfer generator. The transfer generator update interval was chosen for the same reason as in Section 4.3 but based on the number of submitted jobs.
The file size distribution was calculated using the same approach as in Section 4.3. The used monitoring data did not provide the size of each input file, but only the total input volume of each job. Using this monitoring data makes the assumption that one transfer from the tape storage provides data for at least one job. That implies a reasonable data placement on tape and that the workflow management system is able to structure jobs to transfer data bunches from tape and process them. These implications are still challenging objectives. To improve their impact on the accuracy of the simulation, a more detailed model of the job submission would be required. Furthermore, a more detailed tape model including the used data placement strategies would be required.
Since one file corresponds to the input data of one job, the number of initial replicas can be based on the number of jobs. The number of finished jobs in the monitoring system was $\approx 10^6$. Thus, an equal number of files and replicas was created.
As mentioned before, the number of times data was processed is used as the popularity metric. This information can be collected from the central production database.

Site	Source	Destination	Max. active	Value
Both	GCS	Disk	100	294.00 MB/s
Both	Disk	GCS	100	500.00 MB/s
Both	Disk	Worker	∞	88.24 MB/s
Site-1	Tape	Disk	100	22.62 MB/s
Site-2	Tape	Disk	100	62.35 MB/s

Table 5.3: Network configuration of the simulated model.

Most data was processed once, exponentially falling-off to 50 times processed. A geometrically distributed random function can approximate this falloff with the chosen parameters of $p = 0.1$ and the limits of $1 \leq x < 50$.
The number of jobs to submit is generated by a normally distributed random function. The duration of each job is generated by an exponentially distributed random function. The estimation of the parameters of these functions follows the same approach as the number of transfer generation in Section 4.3. Since the number of jobs to submit is fitted to the monitoring data, no job slot limitation is configured.
No specific configuration for the number of output files and volume of the output data was used. This is because in this model, these metrics do not affect the bandwidth or cost. Only the job slot is blocked until all uploads of the output files are finished. Furthermore, the metrics depend directly on the number of jobs finished, and thus could be calculated offline after the simulation.

The other parameters define the network configuration. Table 5.3 shows the used values. The first two rows show the configured bandwidth for the network links between the GCS bucket and the disk storage elements. The third row shows the bandwidth for downloads. These links are configured equally for Site-1 and Site-2. For transfers managed by the transfer service, the maximum number of active transfers was limited to 100 according to the real world limits. The number of downloads is not explicitly limited.
For the bandwidth parameters, the mean value of the monitoring data was taken. Compared to the transfers between the disk storage element and the GCS bucket, the download bandwidth seems to be low, but it is used as unshared bandwidth.
For the bandwidth from and to the GCS bucket, only monitoring data from small scale manual tests was available. Because of the small scale, these values might contain uncertainties and must be adjusted when larger scale data are available.

Another configuration parameter to consider is the access latency of the tape storage elements. Creating a proper model to estimate the access latency would be a complex topic itself and would require detailed log data from real tape systems. For this reason, a normally distributed random value with a mean of 30 minutes and a standard deviation of 15 minutes was used, which were estimated based on operational experience.

The pricing information of the commercial cloud storage is defined in a configuration file. It contains the pricing details of different storage categories, different grades of cost depending on the stored volume, and the network cost depending on the egress destination. The file was created based on the public pricing data from the GCP documentation on the 2020-09-10. For the simulation, the standard storage class for a regional bucket was used.
ATLAS and the VR Observatory worked on research and development projects in collaboration with Google. During this work, special prices were negotiated. Furthermore, Google supports different peering methods than using the internet. Connection to GCP using a non-public network could immensely reduce the network cost. For example, the price for downloading to the internet in Europe is between 0.08 and 0.12 USD/GiB. Using the direct peering option, the cost is reduced to 0.05 USD/GiB in Europe. The interconnect peering option charges only 0.02 USD/GiB [Goo21]. These peering methods typically require a physical network connection to an internet exchange point.

5.2.4 Performance

The simulation was executed on a virtual machine with 8 GB memory and 4 CPU cores clocked at 2.4 GHz. The validation model required less than a second in real time to simulate one day. The more complex HCDC model required less than two seconds in real time for one simulated day. The memory consumption mainly depends on the number of files and replicas. Both types require 76 bytes for each object instance. The memory consumption of the HCDC simulation starts at $\approx$ 480 MB and peaks to $\approx$ 500 MB during run time.

5.3 Evaluation of the Results

This section elaborates on the results of the simulation of the HCDC model. First, the methodology used for the evaluation is explained. Second, the implementation of the used evaluation tools is described. Finally, the results are stated and evaluated.

Cfg.	**Disk limit**	**GCS limit**	**Tape limit**
I	N/A	0	N/A
II	100 TB	0	N/A
III	100 TB	N/A	N/A

Table 5.4: Different storage limits per configuration.

5.3.1 Methodology

Three different configurations of the HCDC model have been simulated. All configurations use the job submission, file, and network parameters, as explained in Section 5.2.3. Since these parameters are fitted to real world data, the limits are known to be achievable by the real world system. The differences between the configurations are the storage limits of the disk storage elements and the GCS bucket. Table 5.4 shows the storage limits per configuration.

Configuration I has a minimal limit set for the GCS bucket. This prevents the usage of the GCS bucket. Moreover, no limit is set on the disk storage elements. With this configuration, all input data is transferred from the tape storage element to the disk storage element and is kept at the disk storage element. The simulation results should be comparable to the current ATLAS continuous derivation production workflow. This is expected because after the initial transfer from the tape storage element to the disk storage element, the data will always be directly available at the disk storage.

Configuration II has the same minimal limit set for the GCS bucket. In addition, a limit of 100 TB is set on each disk storage element, which corresponds to $\approx 1.6\%$ of the used input volume. This configuration of the model shows the results when there is no cloud storage to cache the input data. In this scenario, the input data must be transferred from the tape storage element to the disk storage each time the data is required.

Configuration III is without a limit for the GCS bucket, but with the 100 TB limit set on the disk storage elements. In this way, the fully combined model is simulated with all storage areas usable. This configuration allows analysing the difference when adding the cloud storage as cache.

5.3.2 Evaluation Implementation

The output of the simulation was analysed using Python scripts. A module was written, which contains the `SimData` class. This class represents the data model for the simulation output, i.e., each object of this class represents the output of a simulation run. The class provides several utility routines, e.g., getting storage element information by name, getting only input or output replicas, getting transfers by source or destination.
For the analysis of the results, the module containing the `SimData` class could be imported into a Jupyter notebook. Objects can be created from a given data source. The objects then can be used to implement custom data aggregations and visualisations.
Two types of data source can be used to create `SimData` objects. First, the data from the Postgres database can be exported to a Hadoop Distributed File System. In this case, the analysis uses the Apache Spark engine through PySpark to run data aggregations on a cluster. Using PySpark, the `SimData` functions do not directly process the simulation data but modify data frame objects, which contain only a description of the real data. To receive the data, the data frame including modifications has to be prepared on the cluster and then send to the client. The possibility to use a distributed system for the analysis was implemented to allow the analysis of large output data without hitting memory or run time limitations of local computing nodes.
The second type of data source is directly specifying a Postgres database. In this case, the data are directly downloaded to the client. The data are stored in Pandas data frames, which allow an efficient aggregation. This approach has the advantage that no data export to a distributed file system nor a computing cluster is required. On the other hand, this approach might not scale to a simulation output size that is comparable to the real world.

Figure 5.4 illustrates the primary tables of the simulation output. Omitted are the sites, storage elements, and network links table to improve the visibility. Each value is stored as 8-byte integer. All storage and data volume related properties are stored using bytes as unit. This includes the traffic properties. Of note, the traffic properties require to be stored as 8-byte integers because they can likely exceed the 4-byte integer limits. Since the id property is unique among all tables, it is stored as 8-byte integer to improve the scalability. For example, the number of replicas managed by the real world system is already in the order of a billion. Thus, simulating a real world scale model could exceed a 4-byte integer limit.
There are redundant information, e.g., the source storage element id of a transfer

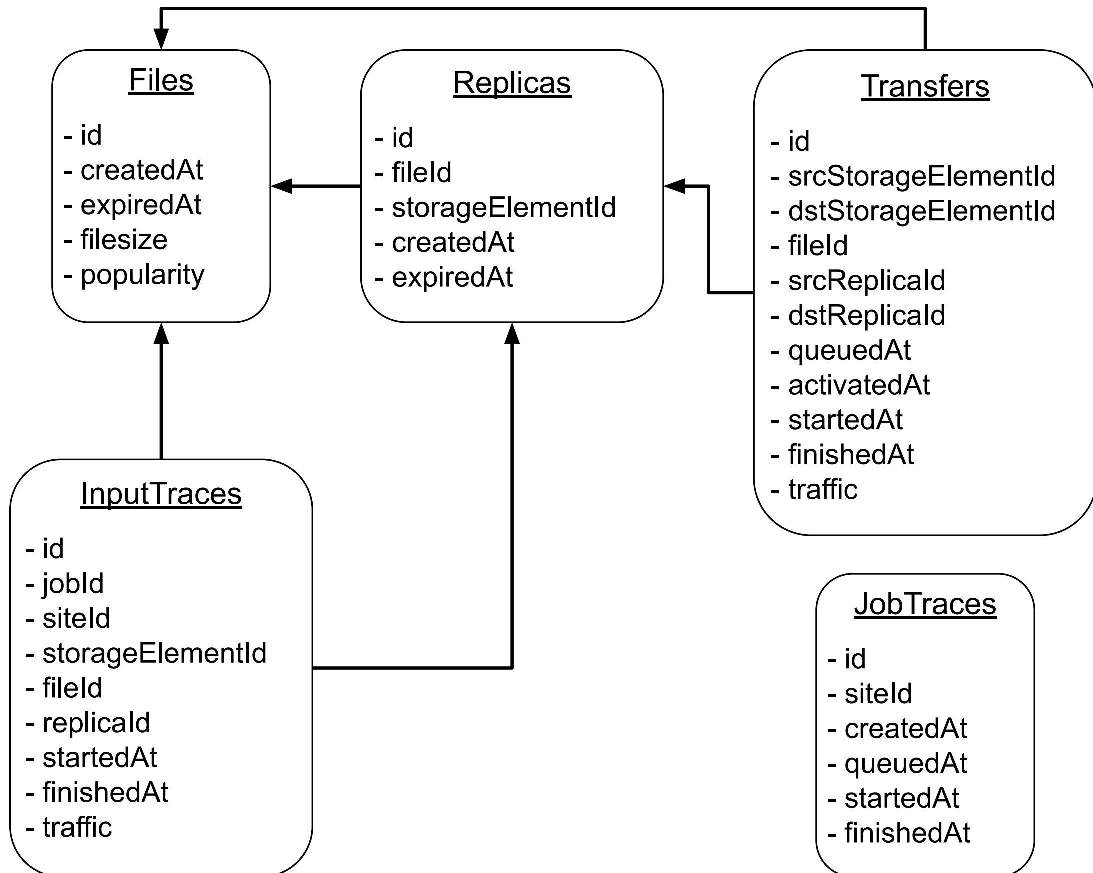

Figure 5.4: Illustrates the data model of the simulation output. Most of the properties represent 64-bit integer values. The id is unique among all tables. The site and storage element tables are omitted for improved visibility.

could be received by finding the source replica in the replicas table. However, storing these additional information enables the evaluation to reduce the number of join operations among the tables, which improves the performance.

One recurring problem during the evaluation was the calculation of the state of an object at a given time point. For example, the number of replicas on a certain storage element at a given time. The replica table only stores one row for each replica with a creation an expiration time. Thus, all created replicas up to the given time point must be considered. Moreover, all expired replicas up to this time point must be filtered out.

The used approach for this problem solves this by creating a temporary table with one row for each creation time and for each expiration time. The table has two columns, one for the time point and another for the number of objects that were created or deleted at this time point. A creation counts as +1, while a deletion

Cfg.	No. jobs done (SE)	Download volume (SE)
I	996k ± 0.05% (0.01%)	41.11 PB ± 0.20% (0.04%)
II	853k ± 0.11% (0.02%)	35.28 PB ± 0.24% (0.05%)
III	996k ± 0.05% (0.01%)	41.02 PB ± 0.38% (0.08%)

Table 5.5: Mean number of finished jobs and mean volume downloaded with their standard deviation for *configuration I, II,* and *III* of 20 simulation runs. The number in the brackets is the corresponding standard error (SE).

counts as -1. Using this approach, the number of existing objects at a given time can be calculated by accumulating the second column up to the given time point.

5.3.3 Results

The first metric that was evaluated is the number of finished jobs. From the data of the monitoring system, it is known that the number of finished jobs should be $\approx 10^6$. As explained in Section 5.3.1, it is expected that the results of *configuration I* are similar to the real world data. In *configuration II*, the data on the disk storage element are deleted after they have been processed because of the limited disk storage. In addition, the GCS bucket is not used. This leads to transferring the data from the tape storage element to the disk storage element each time the data are required. Compared to *configuration I* this results in an increase of total required transfers. With the increased number of transfers and the additional access latency for tape access, *configuration II* is expected to show a decrease in the number of finished jobs.

Table 5.5 shows the number of finished jobs and the total volume downloaded for *configuration I, II,* and *III*. Comparing the results of *configuration I* and *II* already gives an impression of the expected impact of the limit on disk storage. The limit results in $\approx 15\%$ fewer jobs finished and in $\approx 14\%$ less input volume downloaded.
It is conceivable that the difference between the number of finished jobs of *configuration I* and of *configuration II* increases as the simulated time frame increases. The explanation is that in the beginning both configurations behave similarly because all the data are on tape storage. At some point in *configuration I* all the data will be available on disk storage. Thus, the data can directly be downloaded from the disk storage without transferring them from tape first. This is not the case for *configuration II* because of the disk storage limit. With a limited disk storage, further

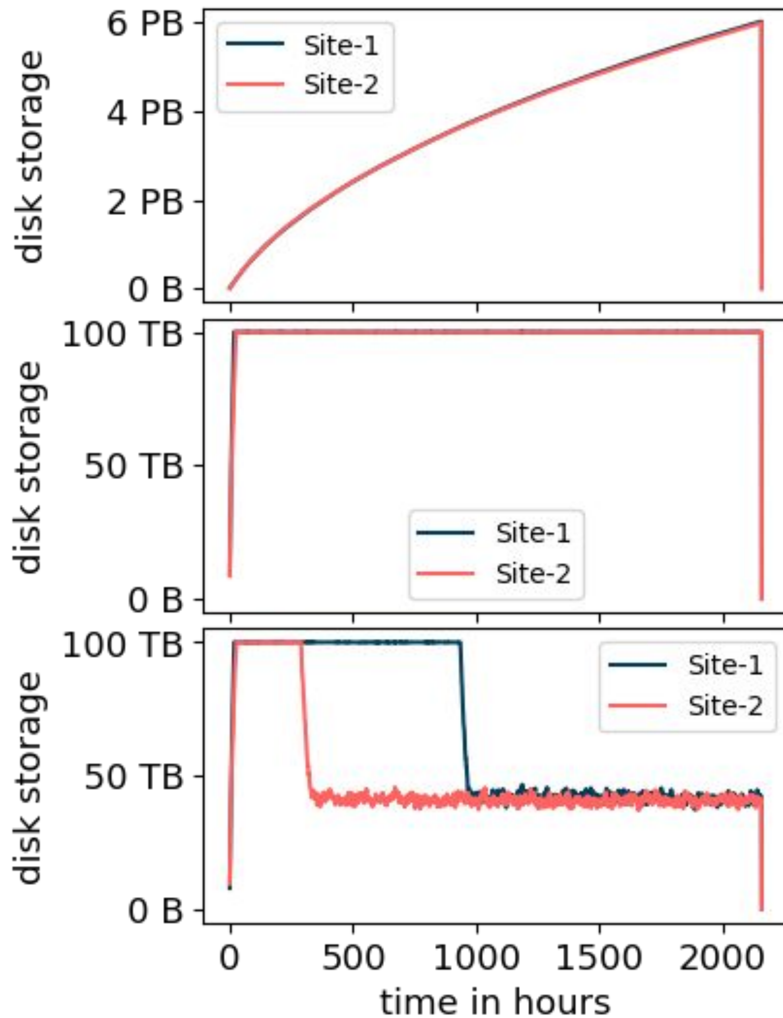

Figure 5.5: Increase of used storage of the disk storage element for *configuration I* (top), *II* (middle), and *III* (bottom). The used storage increase for Site-1 and Site-2 overlap because they are very similar for *configuration I* and *II*.

jobs can only start if sufficient disk storage space is made available by the running jobs.

Configuration III uses the same values as *configuration II* except that the GCS bucket has no limit. As Table 5.5 shows, the results are almost equal to *configuration I* in terms of the number of jobs finished and the volume downloaded. That means, in terms of these metrics, the cloud storage is able to compensate the limit of the disk storage element.

Figure 5.5 shows the increase of used storage over time for each configuration and each disk storage element. In *configuration I*, the disk storage element is unlimited. This results in a quick increase of used storage at the beginning. The more data are transferred to the disk storage, the more the increase flattens out.

The disk storage of *configuration II* is limited to 100 TB as reflected by the corresponding graph in Figure 5.5. The used storage is fluctuating slightly below 100 TB because of the continuous deletion of old replicas and the creation of new replicas. At the beginning of the simulation, *configuration III* is equal to *configuration II* because all the data are at the tape storage element and are required to be transferred to the disk storage element prior to their processing. During this time, more jobs are submitted on average than can be processed because of the tape performance and access latency. This results in a backlog of jobs waiting for disk storage. At some point, a sufficient amount of data are stored on GCS. After this point, fewer jobs are submitted on average than can be processed. This results in a quick processing of the backlog of submitted jobs and afterwards a reduced storage requirement. Site-2 requires less time to reach this point because the tape throughput is larger than at Site-1.

Approximately the same number of jobs are submitted for each configuration, but in the case of *configuration II* $\approx 15\%$ fewer jobs are finished. This leads to the conclusion that some jobs spent more time in early states of the job object state machine. Each job object stores different time points from its submission to its deletion. The *job waiting time* is the time span from the submission of a job until the job is queued. This includes the time the job must wait for disk storage, the time that the corresponding transfer spends in the queue, and the time to finish the transfer.

Figure 5.6 shows histograms for the job waiting time for each configuration. The top histogram shows the data from *configuration I*. As expected, the vast majority of jobs have a small job waiting time because the disk storage is not limited and the data are transferred only once from the tape storage element to the disk storage element. At the beginning of the simulation, the same backlog of jobs builds up that was explained earlier for *configuration III*. The transfers at the end of this backlog represent the outliers in the histogram with a job waiting time of ≈ 150 hours. The second histogram shows the data from *configuration II*. The histogram shows that more jobs have a larger job waiting time when the disk storage is limited. The job waiting time distribution is different between Site-1 and Site-2. The histogram shows that the job waiting time of Site-1 is distributed ≈ 2.5 times larger than of Site-2. As in *configuration III*, a backlog of waiting jobs develops in *configuration II*. In contrast to *configuration I* and *configuration III*, the job processing rate will not exceed the rate of newly submitted jobs because the disk storage is limited and GCS is not used. This means the data must be frequently transferred from tape. Thus, the job backlog will not be processed completely.

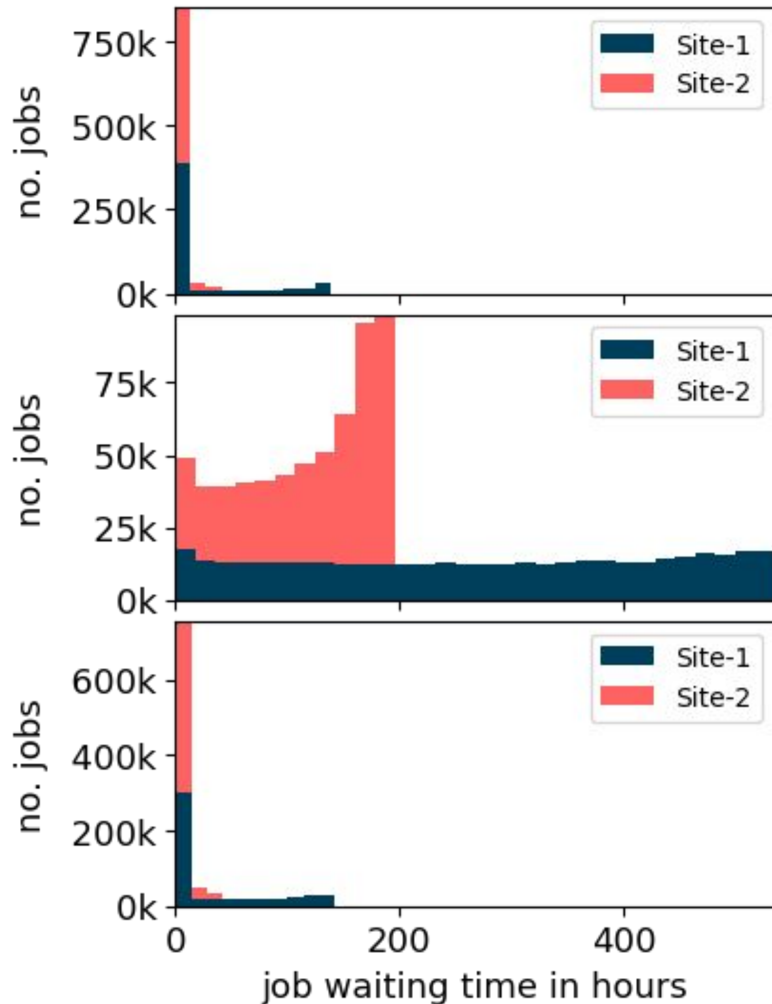

Figure 5.6: Job waiting time distribution of the HCDC model for *configuration I* (top), *II* (middle), and *III* (bottom). The number of bins in the top and bottom histogram was reduced from 30 to 10 to improve visibility of the first bin. The bins are stacked.

In *configuration II*, three types of jobs are responsible for the job waiting time close to 0 hours. First, the very first jobs at the beginning of the simulation. Second, jobs that require data that already exist at the disk storage at the time of the job submission. Third, jobs whose input data are already being transferred to the disk storage at the time of the job submission. Larger job waiting times indicate jobs that were inserted further towards the end of the backlog. In addition, larger file sizes, larger transfer times, and unpopular data potentially increase the job waiting time.

The third histogram shows the data from *configuration III*. The result is similar to the histogram of *configuration I*, but it contains more jobs with durations above 0 hours. This can be explained by the inclusion of the transfer duration into the

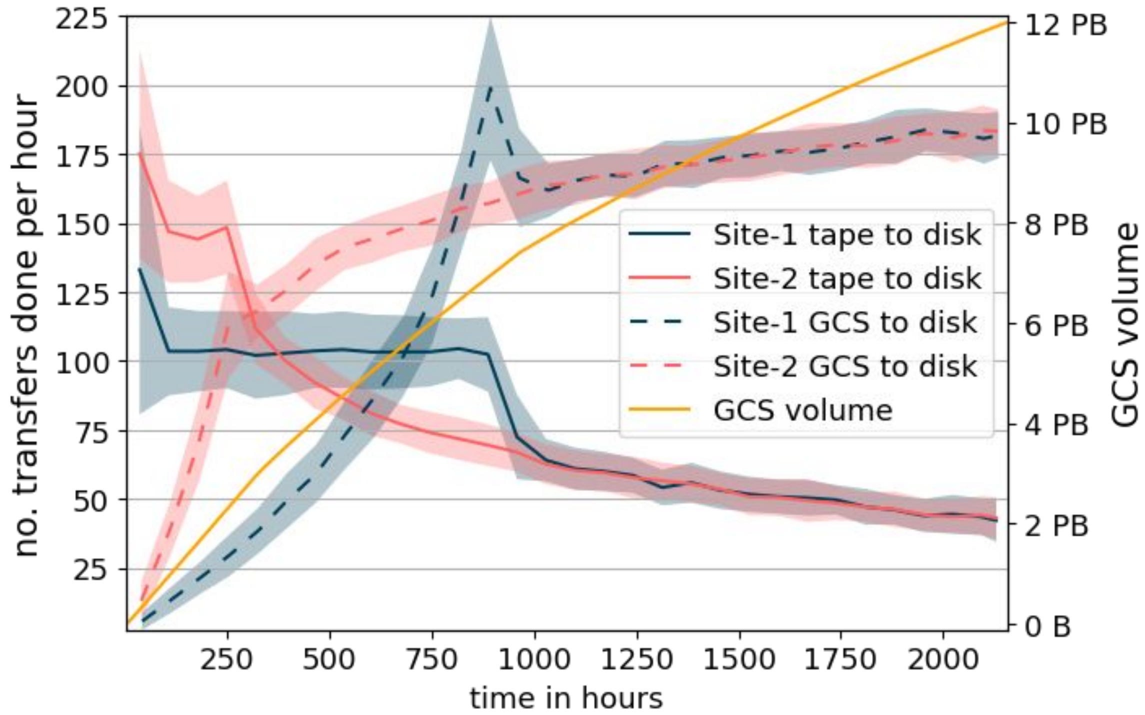

Figure 5.7: The solid blue and red line show the number of transfers from tape to disk per hour for each site. The dashed lines show the number of transfers from GCS to disk per hour for each site. The orange line shows the GCS volume used.

waiting time. In this case, the transfer time from the GCS bucket to the disk storage element is added.

Figure 5.7 illustrates the usage of GCS. This graphic is only available for *configuration III* because it is the only configuration using GCS. Data for the blue and red lines is aggregated in the form of count per hour. To reduce the fluctuations of these aggregated values, a mean filter was applied. The corresponding standard deviation is shown by the line contours.
The blue and red lines show the number of transfers for Site-1 and Site-2, respectively. The solid versions of these lines show the hourly number of transfers from tape to disk, while the dashed versions show the hourly number of transfers from GCS to disk. The orange line shows the used volume of GCS.
Since all the data are solely on tape storage at the beginning, the number of transfers from GCS to disk and the used GCS volume start at 0. The blue and red solid lines show that the most data are transferred from tape to disk storage at the beginning. All the data that are transferred from the tape to the disk storage are subsequently transferred from the disk storage to GCS. Furthermore, this implementation of the model does not delete the data at GCS. That means the orange line increases depen-

Cfg.	Site	Transfer	Volume (SE)
I	Site-1	tape to disk	6.75 PB ± 0.28% (0.06%)
I	Site-2	tape to disk	6.74 PB ± 0.29% (0.06%)
II	Site-1	tape to disk	8.85 PB ± 0.07% (0.04%)
II	Site-2	tape to disk	13.04 PB ± 0.20% (0.01%)
III	Site-1	tape to disk	6.74 PB ± 0.30% (0.07%)
III	Site-2	tape to disk	6.75 PB ± 0.19% (0.04%)
III	GCS	GCS to disk	24.99 PB ± 0.46% (0.10%)

Table 5.6: Mean and standard deviation of transferred volume between storage elements for *configuration I, II,* and *III* of 20 simulation runs.

dent on the dashed lines. At some point, the most popular data are stored at GCS. This is when the dashed line exceeds the corresponding solid line. That means, more data can be transferred from GCS to the disk storage than from the tape storage to the disk storage. The increase of the stored volume at GCS flattens as the most popular data are replicated to GCS.
The figure also reflects the filled disk storage in the first half of the simulation. As explained earlier, there is an initial job backlog due to the tape access latency and limited number of transfers. Because of this, the disk storage limit is reached. The filled disk storage leads to a steadily growing job backlog because new jobs must wait for disk storage space. Even new jobs whose data are already on GCS must wait for disk storage. As the amount of data at GCS is increasing, more and more of the newly submitted jobs use GCS as data source. This allows processing the jobs faster than new jobs are submitted on average. The peak in the dashed lines indicates the time when the backlog has been processed.

Table 5.6 shows detailed numbers of the transfer statistics. The transferred volume of Site-1 and Site-2 for *configuration I* is almost the same. This is because at some point the most popular data are available at the disk storage and do not need to be transferred again from the tape storage. About 13.5 PB of data were transferred to the disk storage. Comparing this amount to the volume downloaded from disk to worker storage of 41.11 PB from Table 5.5 reinforces the conclusion that data are reused.
The transferred volume for *configuration II* shows that significantly more data are

Month	Storage cost/USD (SE)	Network cost/USD (SE)
1	82k ± 0.10% (0.02%)	330k ± 0.39% (0.08%)
2	211k ± 0.16% (0.03%)	729k ± 0.31% (0.07%)
3	293k ± 0.23% (0.05%)	807k ± 0.25% (0.05%)

Table 5.7: Mean GCS costs for each month with the standard deviation and standard error for 20 simulation runs. Pricing information were taken in US Dollar from the GCP documentation at 2020-09-10.

required to be transferred from tape storage. This is because the data at the disk storage are deleted after processing and have to be re-transferred in case they are required again. The difference of the transferred volume between Site-1 and Site-2 makes clear that the tape-to-disk throughput is the bottleneck.
The transferred volume from tape to disk storage for *configuration III* shows similar numbers to *configuration I*. The volume of 6.75 PB indicates the storage required for the most popular data. Once this data are available at the disk storage for *configuration I* or at the GCS for *configuration III*, the tape-to-disk transfer performance becomes less important.
The volume transferred from GCS to disk is split between both sites, which makes a mean of ≈ 1.6 GB/s per site in 3 months. As mentioned before, the configured throughput for the network links is based on small-scale real world tests. This configuration must be adjusted when increasing the scale of the simulation in order to maintain realistic results. The real world tests of the VR Observatory [K20] resulted in a similar throughput estimation. Using a simple regression over 4 data points, they calculated a bandwidth of ≈ 1.5 GB/s without special performance tuning.
At the end of the simulation, there is ≈ 12 PB of data stored at GCS. ≈ 6.8 PB have not been transferred out of GCS during the simulated time. The number of times a file was recalled from GCS is in the range from 0 to 45. The files that were recalled less than 25 times are responsible for more than 90% of the traffic from GCS to the site disk storage.

Table 5.7 lists the mean cost of the cloud resources per month used by the simulated model. The pricing information were taken from the GCP documentation at 2020-09-10. The storage cost increase from the first month on because at the beginning the data are solely on tape and are gradually transferred to the cloud storage. With an increasing amount of data at GCS, more data can be transferred from GCS to a disk storage element. Thus, the network cost is increasing from

month to month. The network cost is in general larger than the storage cost, since the price for network traffic is typically significantly larger than the storage price.

Comparing these results to an approach that uses on-premises resources is challenging, since the costs of the WLCG resources are less explicit and infeasible to model. For example, the evaluation of commercial cloud resources done by the Tokyo site was mentioned in Section 2.2.3. Their estimation for the storage cost of an on-premises system is 1.4 million USD for 16 PB for three years, i.e., $\approx$ 39 thousand USD per month for 16 PB. However, these cost are for the hardware only, excluding potentially significant factors, such as power cost, maintenance cost, and additional infrastructure cost.

Assuming the on-premises storage cost estimated for the Tokyo site, 12 PB of on-premises storage would be about 30 thousand USD per month. On the other hand, the simulation results show 293 thousand USD for approximately the same amount of GCS. This significant difference suggests that the acquisition of additional on-premises disk storage is more beneficial than using GCS. However, it must be considered that: (i) the on-premises cost might be underestimated by neglecting certain cost factors (ii) the difference of the use cases is not considered, i.e., both, Tokyo and the simulation considered the GCS as permanent storage extension, although the simulation requires GCS only for storage bursting during peak requirements.

During the evaluation of the results, numerous thoughts of improving the model, including potentially significant cost reduction, emerged. These thoughts and approaches to continue the work on the HCDC model will be described in the next chapter.

6 Conclusion

The introduction in Chapter 1 explained the necessity to further expand the research for storage models of data intensive experiments. This is not only important for ATLAS, but for all scientific experiments that need to store increasingly large amounts of data.

To face the upcoming data challenges, numerous approaches are investigated. The Data Carousel model described in Section 2.3.1 is a promising concept to migrate large shares of the data to lower cost storage. However, for optimal performance, the Data Carousel model has certain constraints on the workflow organisation. Preferred are infrequent, predictable, bulked data accesses.

Often, the workflow requirements for the Data Carousel model cannot be achieved, and the data processing appears unpredictable instead. In this case, the Hot/Cold Storage model explained in Section 2.3.2 provides an approach to mitigate the decrease of the performance.

In combination, the models result in the HCDC model. The evaluation of an implementation of the HCDC model using commercial cloud storage is required in order to estimate the potential benefit for the future data challenges.

Prior to the implementation of novel storage models, such as the HCDC model, into a production system, a model could be analysed using a simulation software. Using a simulation software provides various advantages. Often, the implementation of a model into a simulation brings up new perspectives of the model, highlights potential issues, and yields optimisation concepts. In addition, a simulation provides a fault-tolerant environment for testing a model. Furthermore, a simulation allows reproducing results by using the same parameters.

As explained in Section 2.4, currently existing simulation software do not suit the needs for simulating data management models. For this reason, the GACS toolkit was planned and developed. It provides an architecture to simulate models related to data management approaches and allows simulating and evaluating the HCDC model.

The architectural goals and decisions of GACS were explained and discussed in Chapter 3. Chapter 4 described the implementation of the architecture. Further-

more, a workflow with sufficient real world monitoring data was simulated to validate the basic functionality of the simulation toolkit.

Chapter 5 explained the implementation of the HCDC model into GACS, the used parameters, and the produced simulation results. The implementation used GCS as cold storage and assumed a continuous production of derivation data. With a processed volume of 41 PB of input data in 3 months and a re-read rate of 25 times or less for 90% of the data, 12 PB of disk storage are required to be able to keep the data solely on tape storage without throttling the job throughput. Using the HCDC model, the on-premises disk storage limit can be further reduced to 100 TB by employing additional cloud storage without reducing the number of finished jobs.
The initial research questions stated in Section 1.2 have been confirmed by the findings from the simulation. The first statement was: using a tape storage system to implement a Data Carousel model for frequently and unpredictably accessed data reduces the processing efficiency by at least 10%. This is approached by *configuration II* of the simulation. The configuration showed that $\approx$ 15% fewer jobs were finished when using the tape based system.
The second statement was: adding a cache-aware model to the tape storage based Data Carousel model allows reducing the on-premises disk storage requirements to less than 5% of the unique input volume without reducing the number of finished jobs by more than 5%. For this statement, the inclusion of GCS as cold storage was simulated in *configuration III*. The statement requires an on-premises disk storage reduction to less than 5% of the unique input data. In the simulated scenario, the unique input volume for each site is $\approx$ 6 PB. Thus, the used disk storage limit of 100 TB is $\approx$ 1.6%. The difference in the number of finished jobs between *configuration I* and *configuration III* is close to zero and within the margin of error.
However, when evaluating the results, it must be considered that the GCS and network usage introduce additional costs. Whether these costs are worth the additional throughput has to be decided individually for each institution of a collaboration.
A simulation software, such as the presented GACS toolkit, can assist by calculating the parameters for the decision whether to use cloud resources. For the HCDC model, parameters can be considered as fixed or variable. Fixed parameters are dictated by the problem description and the existing resources, e.g., bandwidths, input volume to process, number of jobs to run, or the popularity of files. Variable parameters can be changed directly or change indirectly in dependence on other variable parameters. For example, the costs and the job throughput change in dependence

on a potential GCS limit. The time limit to process all the data change in dependence on the job throughput. The required GCS limit depends on the available disk storage limit. Given a specific use case with well-defined limits of the variable parameters, the simulation could be used to estimate the optimal balance among the parameters.
A typical consideration is to compare the costs introduced by the cloud provider against the benefits of the additional resources. In Section 5.3, the benefits of additional resources were measured in form of number of jobs finished resulting from the input volume throughput. From these values, more specific metrics can be derived, e.g., job slot saturation or the volume of output data. For example, *configuration I* required $\approx$ 12 PB more disk storage, but *configuration II* finished $\approx$ 15 % fewer jobs. *Configuration III* induced a cost of more than 2 million USD for three months.

There are two topics that should be prioritised in future work. First, implementing additional concepts to optimise the costs and performance of the model. Second, improving the used parameters and the underlying assumptions in order to achieve more accurate and realistic results. When these topics have been addressed, a specification could be defined for the variable parameters and the simulation can be used to search the optimal values. These values can be used for real world tests to analyse the accuracy of the estimation of the simulation.
In terms of the first topic, there are two missing concepts that should be prioritised in future work. Currently, it is possible to limit the GCS, but the deletion mechanism for GCS is not used. This is an essential concept to be able to define narrower limits on GCS and subsequently reduce storage cost. The deletion is not used because a model is needed that describes how data are selected for deletion, e.g., based on the popularity.
The other concept that should be prioritised in future work is transfers between sites. Using WLCG resources, the data to process is typically distributed among various sites to benefit from different resources and optimally distribute the workload. The challenge of implementing this into the simulation is the creation of a realistic model that specifies the amount and the selection of data to transfer among sites.
When these two concepts are implemented, further adjustments to the HCDC model are possible. For example, the GCS egress cost is even more critical than the GCS storage cost. These could be reduced by improving the deletion at the disk storage element and making the caching strategy more intelligent. Moreover, the utilisation of the tape storage decreases as the GCS usage increases. Instead of always preferring GCS over the tape storage, both storage categories could be used optimally.

One approach for this would be to store the largest files only on the tape storage and not to transfer them to GCS. The access to the tape storage becomes more performant for larger files. The tape storage usage would be improved and the egress cost of the cloud storage would be further reduced.
Another approach to potentially reduce egress cost is to utilise cloud computing resources and derive data directly inside the cloud. Although this approach introduces extra costs for the computing resources, it also reduces the egress cost because only the smaller derived data have to be transferred out of the cloud.

The second topic that should be prioritised for future work is the improvement of the used parameters and models. The used parameters are very specifically fitted to the real world monitoring data. The parameters are realistic as long as the same monitoring data is used. However, changing one of the parameters might invalidate the other parameters. For this reason, the following improvements should be made. (i) As described in 5.2.3, the job submission model should be adjusted, e.g., based on a job slot limit per site. In the case of a job slot based model, it is also important to carefully consider other limitations of the storage systems, such as IOPS. (ii) The number of input files for each job must be generated based on a more realistic distribution. (iii) The network links should be configured with a shared bandwidth, which requires more detailed information about the real network topology. In addition, a background traffic should be added to shared bandwidth links, assuming the links are not exclusively used for the simulated scenario. (iv) The throughput between GCS and the WLCG was based on rather small scale tests and needs to be tested with larger transfer volumes or modelled differently. (v) The simulated HCDC model used a statically assigned popularity based on a fitted random function. This could be replaced by a dynamical assignment. A straightforward approach would be the least recently used concept, which is an established CPU caching technique. More advanced approaches could implement models of existing research about popularity based data replication [MPT18; TZ+12]. (vi) The tape access latency is a strong simplification of real tape systems. A more realistic model based on logs of a real tape system would be required. (vii) The effect of increasing the simulation time has to be investigated in future work. The simulation scales to much larger times, e.g., one year. However, this requires to implement the improvements of the parameters because the current parameters might not be valid for more than 3 months.

Since, the architecture of GACS allows integrating new approaches and concepts without significant effort, it is expected that most of the mentioned future work thoughts technically could be implemented quickly. Moreover, most of the points would be implemented in the HCDC model without the requirement of extending

GACS. Presumably, the only point that could require additions directly to GACS is the implementation of a proper tape system simulation.
In fact, the most challenging part of the mentioned future topics is the proper estimation of newly introduced parameters and the evaluation of their effect on the results. For example, a popularity-based deletion of GCS data. Furthermore, transferring data among the WLCG sites is already possible using GACS. However, the most complex part will be the modelling of reasonable parameters for these systems, e.g., which data to transfer among WLCG sites, the transfer rates, or the number of transfers. To model these, existing real world data and known system limitations must be thoroughly considered.

As of writing this thesis, the data management group at CERN has assembled an R&D programme for the modelling of storage utilisation and the estimation of data transfers. This process is complicated by the unpredictability of modelling; specifically, the estimation of data access patterns, transfer duration, and fluctuations in throughput. In addition, consideration of the upcoming surge in data quantity highlights the importance of investigating advanced storage models, of which the integration of commercial cloud storage was a focus in this thesis.
In mid-2022, a new collaboration project agreement between the US Department of Energy and Google was signed for the benefit of the ATLAS Collaboration. The invested amount in US Dollars is in the order of seven digits. This agreement covers the evaluation of the Google Cloud as a production ATLAS site, and covers several R&D programmes for the development of new integrative tools between high-energy physics and commercial cloud providers. GACS will be used during these programmes for the dataflow cost evaluation, and will be continuously enhanced as necessary.
With GACS as data management oriented simulation tool and the HCDC model as novel approach to integrate commercial cloud resources, this thesis provides a foundation for further research on advanced storage models. This foundation will be used in the R&D programme to continue the research and create optimal methods for the storage and data management at exabyte scale.

Bibliography

[Aab+16] Morad Aaboud et al. “Luminosity determination in pp collisions at $\sqrt{s}$ = 8 TeV using the ATLAS detector at the LHC”. In: *Eur. Phys. J. C* 76.12 (2016), p. 653. DOI: 10.1140/epjc/s10052-016-4466-1.

[AB+09] I. Antcheva, M. Ballintijn, et al. “ROOT — A C++ framework for petabyte data storage, statistical analysis and visualization”. In: *Computer Physics Communications* 180.12 (Dec. 2009), pp. 2499–2512. DOI: 10.1016/j.cpc.2009.08.005.

[AB+17] G. Apollinari, O. Brüning, et al. “High Luminosity Large Hadron Collider HL-LHC”. In: *CERN Yellow Report* arXiv:1705.08830. 5 (May 2017), 1–19. 21 p. DOI: 10.5170/CERN-2015-005.1.

[AS+14] A A Ayllon, M Salichos, et al. “FTS3: New Data Movement Service For WLCG”. In: *J. Phys.: Conf. Ser.* 513 (2014), 032081. 7 p. DOI: 10.1088/1742-6596/513/3/032081.

[Ass21] Assembla. *GroundSim Subversion Repository*. Accessed: 2021-03-31. 2021. URL: https://app.assembla.com/spaces/groudsim/.

[ATL08] ATLAS Collaboration. “The ATLAS Experiment at the CERN Large Hadron Collider”. In: *JINST* 3 (Aug. 2008), S08003. DOI: 10.1088/1748-0221/3/08/S08003.

[ATL10] ATLAS Collaboration. *ATLAS Fact Sheet : To raise awareness of the ATLAS detector and collaboration on the LHC*. 2010. DOI: 10.17181/CERN.1LN2.J772.

[ATL11] ATLAS Collaboration. “Overview of ATLAS PanDA Workload Management”. In: *J.Phys.Conf.Ser.* 331 (Dec. 2011), p. 072024. DOI: 10.1088/1742-6596/331/7/072024.

[ATL17a] ATLAS Collaboration. “ATLAS Distributed Computing experience and performance during the LHC Run-2”. In: *J. Phys. Conf. Ser.* 898.5 (Oct. 2017), p. 052015. DOI: 10.1088/1742-6596/898/5/052015.

[ATL17b] ATLAS Collaboration. “Experiences with the new ATLAS Distributed Data Management System”. In: *J. Phys. Conf. Ser.* 898.6 (Oct. 2017), p. 062019. DOI: 10.1088/1742-6596/898/6/062019.

[ATL19] ATLAS Collaboration. *The Analysis Model Study Group for Run-3 (AMSG-R3)*. Tech. rep. CERN, Apr. 2019. URL: https://cds.cern.ch/record/2672527.

[ATL20] ATLAS Collaboration. *ATLAS HL-LHC Computing Conceptual Design Report*. Tech. rep. CERN-LHCC-2020-015. LHCC-G-178. Geneva: CERN, Sept. 2020. DOI: 10.3204/PUBDB-2020-04972.

[Bar17] Martin Barisits. “Hybrid simulation models for data-intensive systems”. PhD thesis. Vienna University of Technology, Faculty of Informatics, Mar. 2017. URL: https://cds.cern.ch/record/2262420.

[BB+14] I Bird, P Buncic, et al. *Update of the Computing Models of the WLCG and the LHC Experiments*. Tech. rep. CERN-LHCC-2014-014. LCG-TDR-002. CERN, Apr. 2014. URL: https://cds.cern.ch/record/1695401.

[BB+19] Martin Barisits, Thomas Beermann, et al. “Rucio: Scientific Data Management”. In: *Computing and Software for Big Science* 3.1 (Aug. 2019). DOI: 10.1007/s41781-019-0026-3.

[BB+20] Martin Barisits, Mikhail Borodin, et al. “ATLAS Data Carousel”. In: *EPJ Web Conf.* 245 (2020), p. 04035. DOI: 10.1051/epjconf/202024504035.

[BC+04] Oliver Sim Brüning, Paul Collier, et al. *LHC Design Report*. CERN Yellow Reports: Monographs. Geneva: CERN, June 2004. DOI: 10.5170/CERN-2004-003-V-1.

[BC+08] M. Branco, D. Cameron, et al. “Managing ATLAS data on a petabyte-scale with DQ2”. In: *J. Phys.: Conf. Ser.* 119 (2008), p. 062017. DOI: 10.1088/1742-6596/119/6/062017.

[BE+15] Andrew Buckley, Till Eifert, et al. “Implementation of the ATLAS Run 2 event data model”. In: *Journal of Physics: Conference Series* 664.7 (Dec. 2015), p. 072045. DOI: 10.1088/1742-6596/664/7/072045.

[BM02] Rajkumar Buyya and Manzur Murshed. “GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing”. In: *Concurrency and Computation: Practice and Experience* 14 (Nov. 2002). DOI: 10.1002/cpe.710.

[Cas01] Henri Casanova. "Simgrid: a toolkit for the simulation of application scheduling". In: *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*. 2001, pp. 430–437. DOI: 10.1109/CCGRID.2001.923223.

[CG+14] Henri Casanova, Arnaud Giersch, et al. "Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms". In: *Journal of Parallel and Distributed Computing* 74.10 (June 2014), pp. 2899–2917. DOI: 10.1016/j.jpdc.2014.06.008.

[CLQ08] H. Casanova, A. Legrand, and M. Quinson. "SimGrid: A Generic Framework for Large-Scale Distributed Experiments". In: *Tenth International Conference on Computer Modeling and Simulation (uksim 2008)*. May 2008, pp. 126–131. DOI: 10.1109/UKSIM.2008.28.

[Col19] ATLAS Collaboration. "The Data Ocean project: An ATLAS and Google R&D collaboration". In: *EPJ Web Conf.* 214 (2019), p. 04020. DOI: 10.1051/epjconf/201921404020.

[CR+11] Rodrigo N. Calheiros, Rajiv Ranjan, et al. "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms". In: *Software: Practice and Experience* 41.1 (2011), pp. 23–50. DOI: 10.1002/spe.995.

[EB08] Lyndon Evans and Philip Bryant. "LHC Machine". In: *Journal of Instrumentation* 3.08 (Aug. 2008), S08001–S08001. DOI: 10.1088/1748-0221/3/08/s08001.

[Fra21] Framagit. *SimGrid Git Repository*. Accessed: 2021-03-31. 2021. URL: https://framagit.org/simgrid/simgrid.

[Git21] Github. *CloudSim Git Repository*. Accessed: 2021-03-31. 2021. URL: https://github.com/Cloudslab/cloudsim.

[Goo21] Google. *Google Cloud Platform*. Accessed: 2021-03-24. 2021. URL: https://cloud.google.com/.

[Gro21] PostgreSQL Global Development Group. *PostgreSQL*. Accessed: 2021-12-17. 2021. URL: https://www.postgresql.org/.

[GS20] Domenico Giordano and Evangelia Santorinaiou. "Next Generation of HEP CPU benchmarks". In: *Journal of Physics: Conference Series* 1525 (Apr. 2020), p. 012073. DOI: 10.1088/1742-6596/1525/1/012073.

[Hip21] Richard D Hipp. *SQLite*. Accessed: 2021-12-17. 2021. URL: https://www.sqlite.org/.

[HM+20] Charles R. Harris, K. Jarrod Millman, et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[HM98] Fred Howell and Ross Mcnab. "simjava: A Discrete Event Simulation Library For Java". In: *In International Conference on Web-Based Modeling and Simulation*. 1998, pp. 51–56.

[Hun07] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[K20] Lim K. *DMTN-125: Google Cloud Engagement Results*. Accessed: 2020-11-13. 2020. URL: https://dmtn-125.lsst.io/.

[Kan20] Michiru Kaneda. "External Resources: Clouds and HPCs for the expansion of the ATLAS production system at the Tokyo regional analysis center". In: *EPJ Web Conf.* 245 (2020). Ed. by C. Doglioni et al., p. 07034. DOI: 10.1051/epjconf/202024507034.

[LL+15] A. Lebre, A. Legrand, et al. "Adding Storage Simulation Capacities to the SimGrid Toolkit: Concepts, Models, and API". In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. May 2015, pp. 251–260. DOI: 10.1109/CCGrid.2015.134.

[Loh21] Niels Lohmann. *JSON for Modern C++*. Accessed: 2021-09-30. 2021. URL: https://github.com/nlohmann/json.

[LZ12] Saiqin Long and Yuelong Zhao. "A Toolkit for Modeling and Simulating Cloud Data Storage: An Extension to CloudSim". In: *2012 International Conference on Control Engineering and Communication Technology*. Dec. 2012, pp. 597–600. DOI: 10.1109/ICCECT.2012.160.

[MPT18] Marco Meoni, Raffaele Perego, and Nicola Tonellotto. "Dataset Popularity Prediction for Caching of CMS Big Data". In: *Journal of Grid Computing* 16.2 (June 2018), pp. 211–228. ISSN: 1572-9184. DOI: 10.1007/s10723-018-9436-4.

[OP+11] Simon Ostermann, Kassian Plankensteiner, et al. "GroudSim: An Event-Based Simulation Framework for Computational Grids and Clouds". In: *Euro-Par 2010 Parallel Processing Workshops*. Springer Berlin Heidelberg, July 2011, pp. 305–313. ISBN: 978-3-642-21878-1. DOI: 10.1007/978-3-642-21878-1_38.

[Par] Visual Paradigm. *Visual Paradigm Online*. Last accessed: 2022-10-07. URL: https://online.visual-paradigm.com/.

[PM+14] Sergey Panitkin, Fernando Barreiro Megino, et al. "ATLAS Cloud R&D". In: *Journal of Physics: Conference Series* 513.6 (June 2014), p. 062037. DOI: 10.1088/1742-6596/513/6/062037.

[RRC09] Buyya R., R. Ranjan, and R. N. Calheiros. "Modeling and simulation of scalable Cloud computing environments and the CloudSim toolkit: Challenges and opportunities". In: *2009 International Conference on High Performance Computing Simulation*. 2009, pp. 1–11. DOI: 10.1109/HPCSIM.2009.5192685.

[SC+08] Anthony Sulistio, Uros Cibej, et al. "A toolkit for modelling and simulating Data Grids: An extension to GridSim". In: *Concurrency and Computation: Practice and Experience* 20 (Sept. 2008), pp. 1591–1609. DOI: 10.1002/cpe.1307.

[Sou21] SourceForge. *GridSim Subversion Repository*. Accessed: 2021-03-31. 2021. URL: https://sourceforge.net/projects/gridsim/.

[SP+07] Anthony Sulistio, Gokul Poduval, et al. "On incorporating differentiated levels of network service into GridSim". In: *Future Generation Computer Systems* 23.4 (2007), pp. 606–615. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2006.10.006.

[Tas20] Erdogan Taskesen. *distfit - Probability density fitting*. Version 1.4.0. Accessed: 2022-10-01. 2020. URL: https://erdogant.github.io/distfit.

[TB+15] Ryan P Taylor, Frank Berghaus, et al. "The Evolution of Cloud Computing in ATLAS". In: *Journal of Physics: Conference Series* 664.2 (Dec. 2015), p. 022038. DOI: 10.1088/1742-6596/664/2/022038.

[TZ+12] M. Titov, G. Zaruba, et al. "A probabilistic analysis of data popularity in ATLAS data caching". In: *J. Phys. Conf. Ser.* 396 (2012), p. 032106. DOI: 10.1088/1742-6596/396/3/032106.

[VG+20] Pauli Virtanen, Ralf Gommers, et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[VS+13] Pedro Velho, Lucas Mello Schnorr, et al. "On the Validity of Flow-Level Tcp Network Models for Grid and Cloud Simulations". In: 23.4 (Dec. 2013). ISSN: 1049-3301. DOI: 10.1145/2517448.

[Weg+22] Tobias Wegner et al. "Simulation and evaluation of cloud storage caching for data intensive science". In: *Computing and Software for Big Science* 6 (Dec. 2022). DOI: 10.1007/s41781-021-00076-w.

[Weg21] Tobias Wegner. *Grid And Cloud Simulation*. Accessed: 2021-12-17. 2021. URL: https://github.com/TWAtGH/gacspp/.