Utilizing HPC Systems to Serve Compute-Intensive Tasks from a Data Lake Managing Sensitive Data

Dissertation

zur Erlangung des mathematisch-naturwissenschaftlichen Doktorgrades

"Doctor rerum naturalium"

der Georg-August-Universität Göttingen

im Promotionsprogramm Computer Science (PCS)

der Georg-August University School of Science (GAUSS)

vorgelegt von

Hendrik Nolte

aus Göttingen

Göttingen 2024

Betreuungsausschuss

Prof. Dr. Ramin Yahyapour
Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG), Institut für Informatik, Georg-August-Universität Göttingen

Prof. Dr. Dagmar Krefting
Institut für Medizinische Informatik, Georg-August-Universität Göttingen

Prof. Dr. Martin Uecker
Institut für Diagnostische und Interventionelle Radiologie, Georg-August-Universität Göttingen

Mitglieder der Prüfungskommission

Referent: Prof. Dr. Ramin Yahyapour
Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG), Institut für Informatik, Georg-August-Universität Göttingen

Korreferentin: Prof. Dr. Dagmar Krefting
Institut für Medizinische Informatik, Georg-August-Universität Göttingen

Weitere Mitglieder der Prüfungskommission:

Prof. Dr. Martin Uecker
Institut für Diagnostische und Interventionelle Radiologie, Georg-August-Universität Göttingen

Prof. Dr. Julian Kunkel
Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG), Institut für Informatik, Georg-August-Universität Göttingen

Prof. Dr. Anne-Christin Hauschild
Institut für Medizinische Informatik, Georg-August-Universität Göttingen

Prof. Dr. Christian Riedel
Institut für Diagnostische und Interventionelle Neuroradiologie, Georg-August-Universität Göttingen

Tag der mündlichen Prüfung: 4. September 2024

Abstract

Data lakes have been proposed by James Dixon in 2010 to prevent information loss happening during the extract-transform-load process usually employed to ingest data into a data warehouse. Instead, he proposed a schema-on-read semantics, where the schema is only inferred on the data when reading. This simple, yet powerful idea has been drastically extended to support multiple data sources and use different ecosystems, like Hadoop or public cloud offerings.

Within this thesis, a data lake is designed that integrates sensitive health data and offloads compute-intensive tasks to a High-Performance Computing (HPC) system. The general applicability is demonstrated through a use case where Magnetic Resonance Imaging (MRI) scans are stored, cataloged, and processed. For this, an analysis of the challenges of data-intensive projects on HPC systems is done. Here, the requirement of a secure communication layer between a remote data management system, i.e., the data lake, and the HPC system is identified, alongside the general problem of data placement and lifecycle management.

Following this analysis the communication layer is implemented and provides a Representational State Transfer (REST) interface. This system can be completely set up in user space allowing easy scalability and portability. In addition, it provides a fine-granular, token-based permission system. Among other things, this allows to automatically start uncritical tasks, like the execution of an already configured job, while a highly critical code-ingestion is only possible via an out-of-band confirmation of a second factor. Its generic design enables usage outside of the scope of the envisioned data lake by basically any client system.

In order to implement the actual data lake, the abstraction of digital objects is used. It allows to organize an otherwise flat hierarchical namespace and abstracts the underlying infrastructure. This data lake can offload compute-intensive tasks to an HPC system while providing transparent provenance auditing to ensure maintainability, comprehensibility, and support researchers in working reproducibly. It further provides a modular design, which allows swapping out, or adapting the required backend services, like storage, metadata indexing, and compute.

This modular design is leveraged to guarantee full data sovereignty to the users by providing the capability to store, index, and process encrypted data. For this, a new secure HPC service is developed which provides a secured and isolated partition on a shared HPC system. This secure partition is completely generic and fully implemented in software, therefore it can be deployed on any typical HPC system and does not require any additional procurement. Different encryption techniques are systematically benchmarked with synthetic and real use cases where up to 97% of the baseline performance is achieved. In addition, it can also be used as a standalone service, which is demonstrated by including it in a clinical workflow.

Lastly, scalable and secure metadata indexing is provided by on-demand Elasticsearch clusters which are spawned within the context of a typical HPC job. Two different encryption techniques are discussed and benchmarked. One is based on the

previous secure HPC service, while the other is platform-independent by encrypting all key-value pairs of a document individually on the client side and performing all metadata operations, like search or range queries, on encrypted data on the server side. Therefore, the encryption keys never have to leave the client system at any time. To assess the two different encryption techniques, a throughput-oriented benchmark is developed and used on a synthetic dataset and within the MRI use case.

Acknowledgments

I would like to thank my supervisor Prof. Dr. Ramin Yahyapour who has guided and supported me through the entire process of defining a suitable research topic to the final submission of my thesis. I want to extend this appreciation to my entire Thesis Advisory Committee, Prof. Dr. Dagmar Krefting and Prof. Dr. Martin Uecker, who supported me and provided valuable ideas.

I owe my colleagues from GWDG my greatest gratitude for the supportive working environment without which this work could not have been successful. In particular, I want to recognize the profound influence of Prof. Dr. Julian Kunkel on me and my work. I also want to acknowledge the outstanding roles of Prof. Dr. Philipp Wieder, Dr. Christian Boehme, Dr. Christian Köhler, Dr. Tim Ehlers, and Dr. Sebastian Krey during my studies.

Writing all the code required for this work, quickly iterating through different design choices, and evaluating these would not have been possible without the great help of all the many students I supervised for theses, projects or as student assistants during this time. Their crucial help is highly acknowledged.

Last but not least I want to thank my family for their endless and empowering encouragement throughout the entire time.

Contents

List of Figures

List of Tables

List of Abbreviations

2FA	Two-Factor Authentication
ABE	Attribute-Based Encryption
ACID	Atomicity, Consistency, Isolation, Durability
ACL	Access Control List
API	Application Programming Interface
BART	Berkeley Advanced Reconstruction Toolbox
BIDS	Brain Imaging Data Structure
BI	Business Intelligence
BLOB	Binary Large Objects
BoF	Birds-of-a-Feather
CA	Certificate Authority
CI/CD	Continuous Integration and Integration Development
CLI	Command Line Interface
CPU	Central Processing Unit
CSV	Comma Separated Values
CWFR	Canonical Workflow Framework for Research
DAG	Directed Acyclic Graph
DB	Database
DCPAC	Data Catalog, Provenance, Access Control
DICOM	Digital Imaging and Communications in Medicine
DMP	Data Management Plan
DMS	Data Management System
DOIP	Digital Object Interface Protocol
DO	Digital Object
DSL	Domain Specific Language
DSRP	Design Science Research Process
ECG	Electrocardiography
EDF	European Data Format
EEG	Electroencephalography
EMG	Electromyography
EOG	Electrooculography
ES	Elasticsearch
ETL	Extract-Transform-Load
FAIR	Findability, Accessibility, Interoperability, and Resusability
FDO	FAIR Digital Object
FM	Fabric Manager
FUSE	File System in User Space
FaaS	Function as a Service
GDPR	General Data Protection Regulation
GPFS	General Parallel File System
GPU	Graphics Processing Unit
GUI	Graphical User Interface

HDFS	Hadoop Distributed File System
HIPPA	Health Insurance Portability and Accountability Act
HPC	High-Performance Computing
HTTP	Hypertext Transfer Protocol
IPoIB	Internet Protocol over Infiniband
IP	Internet Protocol
ISC	International Supercomputing
IaaS	Infrastructure as a Service
JIT	Just-In-Time
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
KMS	Key Management System
LDAP	Lightweight Directory Access Protocol
LUKS	Linux Unified Key Setup
MPI	Message Passing Interface
MRI	Magnetic Resonance Imaging
MVP	Minimum Viable Product
NDJSON	Newline Delimited JSON
NFS	Network File System
NIC	Network Interface Cards
NIfTI	Neuroimaging Informatics Technology Initiative
OAuth	Open-Authorization
OLAP	Online Analytical Processing
OPE	Order-Preserving Symmetric Encryption
OPM	Open Provenance Model
ORE	Order Revealing Encryption
PACS	Picture Archiving and Communication System
PCOCC	Private Cloud on a Compute Cluster
PID	Persistent Identifier
PKey	Partition Keys
POSIX	Portable Operating System Interface
PSG	Polysomnography
PXE	Preboot Execution Environment
QoS	Quality of Service
RDMA	Remote Direct Memory Access
REM	Rapid Eye Movement
REST	Representational State Transfer
S3	Simple Storage Service
SCP	Secure Copy
SDK	Software Development Kit
SQL	Structured Query Language
SSH	Secure Shell
SSO	Single Sign-On
SaaS	Software as a Service
TCP	Transmission Control Protocol
TEE	Trusted Execution Environments
TLS	Transport Layer Security
UI	User Interface
UMG	University Medical Center Göttingen
URI	Unique Resource Identifier
URL	Uniform Resource Locator

VM	Virtual Machine
VPN	Virtual Private Network
W3C	World Wide Web Consortium
XML	Extensible Markup Language
XNAT	Extensible Neuroimaging Archive Toolkit
vFabric	Virtual Fabric

Chapter 1

Introduction

In the last decades and even centuries, most scientific progress has been made in a modus operandi, where a basic observation is described by a researcher using a model or theory. Within this newly crafted theory, predictions are made to identify possible experiments that can be used to falsify this theory [160] or to detect anomalies to either improve the existing theory or to provoke a paradigm shift [102]. These experiments are then conducted to collect data, which is afterward analyzed to determine if it agrees with the theory. Historically, many experiments could be performed in laboratory conditions to measure the functional dependency of the required observables in isolation. This motivates the scientific method, an ever-repeating process of designing a hypothesis based on an initial question, in which predictions have to withstand an experimental verification/falsification, ending in further questions, triggering a new execution of this cycle.

However, with the general progress in science, the models in question have become increasingly complex which makes it impossible to isolate the different observables from each other, which are all influencing the system. This makes it increasingly hard if not near impossible to design suitable experiments to test predictions derived by the theory in question.

Meanwhile, there have been huge advances in data acquisition, data storage, and data processing. Nowadays there exists a wealth of sensors that can continuously monitor almost any observable, ranging from weather information like temperature and pressure to personalized medical data like the current heart rate or blood sugar levels. The cost-effectiveness of most of these sensors has led to widespread adoption, while the ability to gather data from almost anywhere is continuously increasing along, which is even driven by other trends like Internet of Things devices. This capability to collect vast amounts of diverse data from heterogeneous sources has coined the term big data, which is primarily defined by the inability to analyze this data set with traditional methods.

Although working with big data has numerous challenges [80, 120], it has the potential to revolutionize research, particularly in complex systems, where the historic scientific methods reached their limits. Here, data science has emerged as a new scientific field that aims to extract knowledge from these datasets and was even titled as the fourth paradigm of science [17], next to theory, experiment, and simulation. This also highlights the outstanding role of compute infrastructure in modern research, considering the high compute demand from scientific simulations. These compute-intensive tasks are commonly served with High-Performance Computing (HPC) clusters. These supercomputers are used to solve extremely compute-intensive problems that are too large to compute on a typical personal computer.

1.1 From Data Warehouses to Data Lakes

To enable knowledge extraction from large datasets, also called data mining, proper data and process management is key. Data management systems are used to store, retrieve, process, and update data, which can then, among other things, be used for scientific hypothesis testing [33]. However, to fully exploit the available wealth of information and to ease the use of data mining techniques, data should be stored cleaned, and integrated [89].

1.1.1 Data Warehouses

This has led to the widespread adoption of data warehouses, which provide access to cleansed and integrated data from different, or ideally all available data sources at a central location [53, 90]. To do so, a data warehouse requires a pre-defined schema before any data can be ingested.

Therefore, a thorough process to design a model is necessary to reduce the chance of misrepresentations later on. Although there was a dispute in science about the different stages of this design process, it was established that first the conceptual modeling is done, where entities, their attributes as well as their relationships are defined. This is followed by logical modeling, and lastly, physical modeling where the previously defined conceptual modeling is further concretized with respect to database-specific datatypes [69, 210]. Lastly, the multidimensional modeling is done, where the previously derived model is split into fact and dimension tables. A fact is hereby the actual observable, a dimension is an additional information describing the observable, i.e., the fact.

This approach introduces a mismatch between the format in which the data are generated at the source, and the format, which is defined in the data warehouse. Therefore, data extracted from a source need to be transformed into the destination format specified by the data warehouse. This problem is generally solved by introducing a dedicated system to perform the required Extract-Transform-Load (ETL) process. However, this leads to an information loss that is happening during the transformation step, where any information that is contained in the raw data but which is not sufficiently modeled in the data warehouse will be discarded. One example of an information loss is an aggregation operation, which decreases the resolution of the raw data along one or multiple dimensions.

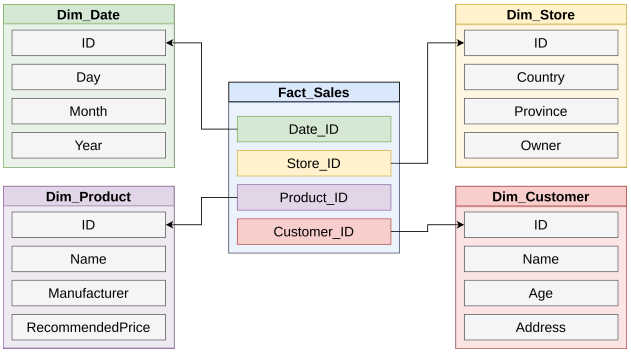


FIGURE 1.1: Star schema of a data warehouse representing a sale in a store. The sale is the actual measured observable, whereas the four dimension tables provide further information about the attributes of the sale.

This information loss has to be particularly considered in a scientific context, where the developed model of the data warehouse will be highly influenced by the specific research question that the researcher is currently investigating. Since it is possible that the initial research question will be altered, or refined over time due

to new insights and a better understanding, this means that the model needs to be adapted as well. However, if the already ingested data is not available in its raw format, as it is common in data warehouses due to the ETL processes, this data cannot be enriched to fit into the updated format. Therefore the reusability of the already gathered data is lost.

1.1.2 Data Lakes

As a solution to this severe drawback, James Dixon proposed the concept of a data lake [56]. Here, the schema-on-write approach of data warehouses and the necessary ETL processes are substituted by a new schema-on-read paradigm. This means that all ingested data is retained in its native format and only when the data, or a subset of it, is requested, the data will be transformed into the required format and is passed to the following process.

Considering the definition of data science as an "open-ended analysis without preconceived hypothesis" [35], a data lake seems like the ideal platform to support this kind of research since the data in it were not subjected to any cleansing, aggregation, or similar pre-processing steps which are usually derived from an underlying scientific theory.

The fundamental idea of data lakes is to retain all raw data in its native format immediately preventing the usage of any lossy compression techniques. This requirement directly leads to a huge increase in data volume which needs to be stored. On the other side, the data must not only be stored but also has to be readily available for subsequent analysis. This results in the necessity for low-cost and scalable storage and tight integration with similarly scalable compute resources to support highly parallelized compute tasks.

1.1.3 Integrating Health Data

The life science domain has the potential to benefit drastically from readily available and integrated health data [180]. For example, by extracting patterns from different patient monitoring systems and using these different input variables, like blood pressure, heart rate, temperature, respiratory rate, peripheral capillary oxygen (SpO2), age, and many more, an early sepsis prediction could be made, which have proven effectiveness in a randomized clinical trial [181]. This example demonstrates that by using data mining techniques meaningful correlations could be identified, or observations could be made, which could have not been obtained by doing an isolated experiment on a living human, or by simulating the entire human body.

Therefore, it is not surprising that the large cloud providers are now increasingly offering special data lake services for personal health record management, like IBM's efforts in using a blockchain [151], or Amazon's *HealthLake*. The reason, why novel developments are required when integrating health data into a data lake is that very high privacy requirements are mandatory by laws such as the Health Insurance Portability and Accountability Act (HIPPA) in the USA or the General Data Protection Regulation (GDPR) by the EU. Therefore, platforms to store, index, and process these sensitive data sets need to be extra hardened to ensure full data sovereignty to the users.

There is an increasing interest by researchers in universities and other public research institutions to process personalized health data, preferably within their local data centers. This has led to a widespread quest by data center operators to provide these highly secured platforms, which is perceived as both, challenging and urgent,

as mentioned in [44]: “At UC Berkeley, this has become a pressing issue“ and it has “affected the campus’ ability to recruit a new faculty member“.

However, just providing a secure platform to do one isolated task, e.g., processing this sensitive data, is not enough. Instead, an overarching and seamless security architecture covering the entire data life cycle is mandatory.

1.2 Goal of this Thesis

The overarching goal of this thesis is to design a data lake that can serve as a central data repository for diverse scientific domains. Its applicability shall be tested within a use case in the life sciences, where data privacy and compute capacity are the main concerns.

> *How can a data lake be designed so that it can serve different scientific domains as the central data repository and processing platform, with a specific focus on sensitive data?*

To answer this primary research question it was split up into different, more detailed, and isolated research questions. These are discussed in the following.

1.2.1 Data Lake Architecture

In the first part, the focus lies on the general development of a generic data lake architecture that can utilize scalable resources for data storage, data processing, and metadata management in a modularized manner. The following research question can be articulated:

> *What data lake architecture can be used to organize a flat-hierarchical data lake with respect to data maturity and offered functionality that is built in a modularized manner using scalable components?*

1.2.2 Scientific Support

Since the intended user group are scientists the developed architecture needs to be focused on these specific needs and assist the researchers in good scientific practice.

> *How can this data lake architecture be made FAIR by design, allowing for the integration of diverse scientific workflows and enforcing overarching data management plans across the lake?*

1.2.3 Secure Processing of Sensitive Data

One scientific domain that is particularly targeted by this data lake are the life sciences. This can specifically involve working with regulated health data. Therefore, the highest security standards are required when processing these data sets.

> *What are the attack vectors to gain access to a user’s data while processing them and how can trusted and secure compute capacity be provided which ensures the full data sovereignty to the users?*

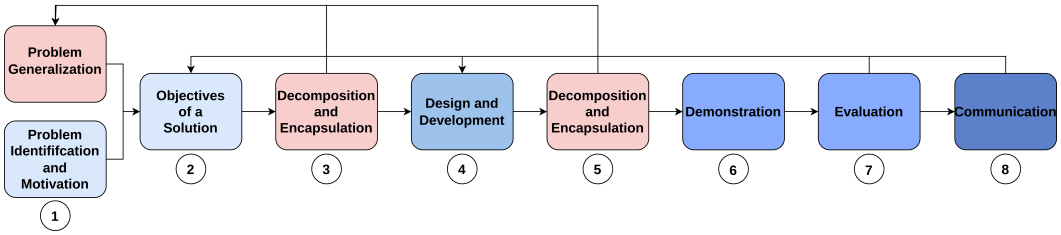


FIGURE 1.2: Used scientific process adapted from the Design Science Research Process [156]. The red steps represent novel adaptions not part of the original DSRP, while all blue shades represent the original DSRP. The different blue shades map to the four different design science process elements of Hevner, i.e. relevance, artifact, evaluation, and communication.

1.2.4 Secure Metadata Handling of Sensitive Data

Since a modularized data lake design is envisioned, one also needs to ensure secure metadata handling separately.

> *What are the attack vectors to gain access to data stored in a database or search engine and how can a secure platform be provided that ensures the full data sovereignty to the users?*

1.3 Research Methodology

The general research methodology used in this thesis is based on the *Design Science Research Process* (DSRP) model, which is formalized by Peffers [156]. This process consists of six steps: a) problem identification and motivation, b) objectives of a solution, c) design and development, d) demonstration, e) evaluation, and f) communication. It extends the *Design Science Research Cycles* described by Hevner [86], consisting of the relevance cycle where the problem or opportunity is described, followed by the rigor cycle where existing knowledge is gathered, and is finished by the design cycle where the actual artifact is designed and evaluated. The DSRP enables researchers to start at any step before the evaluation, as opposed to the methodology established by Hevner which can solely start at the relevance cycle.

Although these are well-established scientific methodologies, they only cover the lifecycle of a single project with a single resulting research artifact. Thus, if the initial problem identification is done too broadly, the strict application of these scientific methods can lead to more integral instead of modular solutions. Therefore, there are efficiency limits when applying these methods to complex problems.

This drawback is accounted for within this thesis by adapting the DSR process as shown in Figure 1.2. Here, an explicit *Decomposition and Encapsulation* step is included after the definition of the *Objectives of a Solution* and again after the *Design and Development* cycle. The goal of these steps is to identify the possibility of restricting further the problem and the derived objectives, i.e., to extract a more specific sub-problem from the original complex problem and solve this using the same adapted DSRP methodology. To be eligible, the expected research artifact of such a sub-problem needs to fulfill two requirements. First, it can be encapsulated, i.e., the complexity of it can be hidden behind a well-defined interface. Second, it needs to solve a general problem, not just a very specific aspect only relevant to this particular

work. To ensure the latter, an extra *Problem Generalization* step precedes the *Objectives of a Solution* step, whereas Peffers [156] suggests starting at the *Objectives of a Solution* or the *Design and Development* step. This deviation from the well-established DSRP shall maximize the reusability and impact of the work.

This eight-step scientific process is applied in this thesis to design a generic data lake to integrate sensitive health data. In the following, only the eight steps for the main process are described, while the identified sub-problems are only mentioned, but are not recursively listed in detail:

1. **Problem Identification and Motivation:** At the beginning, a real-world use case within the life sciences is described, the problem is identified, and the advantages of using a generic data lake for this specific use case are motivated. To clearly identify and define the problem, this use case analysis is accompanied by an extensive literature survey to establish a solid knowledge base.

2. **Objectives of a Solution:** From the requirement engineering based on the use case, as well as from the knowledge about the state-of-the-art, a set of objectives for a suitable solution is derived.

3. **Decomposition and Encapsulation:** A systematic analysis of the required components, their hierarchy, and their interactions are done. If a general and isolated component is found, which is not already available, this is used to restart the process at Step 1 with the *Problem Generalization*.

4. **Design and Development:** The data lake is designed and developed, based on the previously gathered knowledge and for the defined objectives.

5. **Decomposition and Encapsulation:** The design and implementation are checked for isolated sub-components. These can be identified by a well-defined interface towards other components that might also exhibit a hierarchical relationship.

6. **Demonstration:** The applicability of the data lake is demonstrated by using it within the original project.

7. **Evaluation:** Based on the demonstration it is evaluated if the data lake fulfills the prior defined objectives.

8. **Communication:** The communication is done in the form of this thesis, papers, posters, and talks.

1.4 Contributions

By applying this research methodology to the stated research questions the following contributions are made in this thesis:

- Conducting a requirement analysis for the overarching project in which context this thesis is written and proposing an envisioned user workflow based on a condensed set of functional and non-functional requirements

- Identifying gaps in current data lake research based on an extensive literature survey concerning the integration of ecosystem-agnostic compute infrastructure and implementing security measures to manage and process sensitive health data

- Proposing an overarching concept to integrate different user interfaces into data-intensive HPC workflows to enforce a global data governance
- Developing a secure communication layer to orchestrate the control flow of HPC jobs between HPC systems and externally hosted user frontends
- Designing a novel digital object-based data lake architecture to provide the required abstraction of the underlying modularized systems
- Providing a blueprint to integrate new scientific workflows within this data lake which rely on diverse compute infrastructure while still fulfilling the FAIR principles
- Developing a secure partition on a shared HPC system that enables the processing of sensitive data on highly scalable infrastructure
- Proposing two different workflows for providing secure metadata management where one uses client-side encryption while the other offers server-side encryption

1.5 Structure

The remainder of this thesis is structured as follows:

- **Chapter 2:** First, a requirement analysis is done based on an overarching use case. Based on these requirements, a data lake is identified as a suitable data management system, for which further requirements are defined, which are tested with a questionnaire towards their generality.
- **Chapter 3:** General background information about data lakes and their related work showing the current state of the art is discussed. The general applicability of data lakes for the presented use case is confirmed based on use cases provided by the literature. Considering the previously defined set of requirements different shortcomings of the existing systems are identified. One large gap in existing techniques is the integration of diverse compute infrastructure into data lakes, with a complete lack of HPC support.
- **Chapter 4:** This gap is in detail analyzed, investigating the challenges of data-intense projects on HPC systems and conceptually proposing a solution. One specific component that is hereby discovered is a communication layer between a remote data management system and an HPC system.
- **Chapter 5:** This specific component is generically designed, so that it can not only provide the required interface for the envisioned data lake but is generic enough to work with other data management systems as well.
- **Chapter 6:** Based on this work, a modular data lake design is developed according to the requirements defined in Chapter 2. To ensure the data privacy of sensitive data, the data lake design needs to provide the highest security. To provide this, the modular design is used to divide this into smaller subproblems.
- **Chapter 7:** Following this modular design, a secure HPC partition is developed which allows the processing of sensitive data on shared HPC systems.

- **Chapter 8:** Two different methods are developed for secure metadata indexing to provide a secure data catalog. Both of these systems can be used within the context of the proposed data lake, as well as standalone services. The applicability within the project is verified with a novel benchmarking tool.
- **Chapter 9**: A conclusion is drawn and an outlook for further research is provided.

Chapter 2

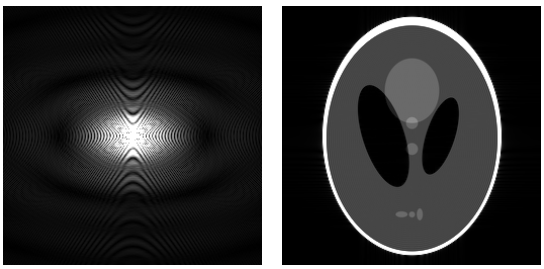
Requirement Analysis and System Specification

To derive the objectives of a solution, a requirement analysis is done. For this, the use case is described in Section 2.1. Then in Section 2.2 a requirement analysis is done specifically for this project with a focus on re-usability of the created scientific artifacts beyond this project. Based on this, in Section 2.3 a requirement specification artifact is modeled as an envisioned user workflow, and a system specification is proposed. Afterward, the gathered objectives are tested for their general importance by a poll in Section 2.4.

This thesis is written in the context of a joint project with researchers from the University Medical Center Göttingen (UMG) within the overarching project "Tapping Into a Resource Hidden Behind MR Images: Learning Quantitative Imaging Biomarkers from Raw Big Data", which involves 4 departments. Within this project, only a limited number of data sets are recorded specifically for research purposes. Therefore, these data sets do not have to be classified as health data corresponding to GDPR. However, the overarching goal is to develop pipelines that can later scale to very large data sets and can also be used for clinical purposes, i.e., for highly sensitive and regulated health data.

2.1 Use Case Description

Unlike other medical imaging techniques, like x-rays, Magnetic Resonance Imaging (MRI) scanners do not record images in the well-known image space, i.e., the space in which the recorded data points directly yield the objects as humans can see them. Instead, an image is recorded in the k-space or reciprocal space. Hereby, the k-space and the image space are connected by a Fourier transformation. This relationship is exemplified in Figure 2.1. In Figure 2.1a, Berkeley Advanced Reconstruction Toolbox (BART)'s [198] Shepp Logan phantom is shown in k-space. In Figure 2.1b, the same k-space image is shown reconstructed by a uniform fast Fourier transformation in image space. This shows, that in order for humans to be able to work with MR images, the actual recorded k-space image needs to be processed, i.e., to be reconstructed into image-space. Typically, not only humans work with these reconstructed images, but



(A) K-Space (B) Image Space

FIGURE 2.1: BART's Shepp Logan phantom.

also analysis tasks, like machine learning methods, try to extract features from these reconstructed images and correlate them to known indicators. However, the reconstruction process is not exact and leads to information loss. Thus, all processing done on reconstructed images is subjected to the already happened information loss and the additional inaccuracies this entails.

Overcoming this information gap between the original k-space and the reconstructed image is the main goal of this project. This is envisioned to be achieved by directly extracting relevant imaging biomarkers from the actually recorded MR signals and including this information back into the reconstruction process. The expectation of these novel techniques are that they will either improve the image quality, for instance when compared to the standard reconstructions an MRI scanner produces for medical uses, or that the recording time of a scan can be reduced while ensuring the same image quality as before. To confirm this expectation, these reconstructed images are then further processed, for instance for multispectral brain tissue segmentation, a volumetric analysis of the brain, or a thickness analysis of the brain. These pipelines can then use both input data, the novel image reconstruction, and the standard reconstruction from the scanners. A simplified view of the processing workflow is shown in Figure 2.2. It is shown, that from the MRI scanner, both the k-space images as well as the reconstructed images are extracted. The k-space images need one more reconstruction step before both reconstructions can be processed with the same processing pipeline. In order to obtain meaningful statistics this depicted workflow should be applied to large data sets. Although only a limited number of MRI scans are planned within the scope of this project, the developed pipeline should be scalable to also work with larger cohorts, like the UK Bio Bank [188].

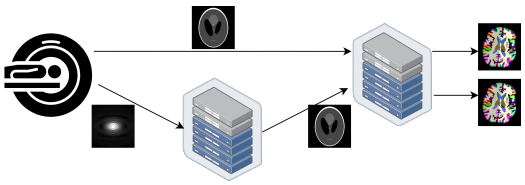


FIGURE 2.2: Simplified view of the MRI processing workflow.

Within this project, the recorded MR images should be archived in a suitable data repository, not only to enable data sharing between all involved researchers within this project, but also to provide access to this data to other scientists as well. This should increase the reusability of the recorded data and ensure its impact beyond the scope of this single project. Therefore, new requirements can come up in the future which are hard to foresee. Thus, both the data repository as well as the processing infrastructure should support new use cases and should not be too narrowly designed for these specific requirements at hand.

2.2 Requirement Analysis of the Use Case

From the original proposal of the described project, as well as many discussions with the involved researchers about their current practice and the envisioned goal, a set of functional and non-functional requirements could be identified. The main, high-level goal is to provide a data and process management system for this project.

Data Repository Doing an MRI scan is very time-consuming and costly. Therefore, the measured raw data has high value and should be stored in a data repository, where the findability, accessibility, and integrity of the data can be ensured. This means specifically that raw data needs to be retained and must not be deleted or

overwritten. In addition, it is important to collect and link the related metadata, particularly for file formats that do not carry any metadata themselves but rely on sidecar files. The metadata should be indexed to yield an overarching data catalog to allow researchers to select the data of interest based on domain-specific, semantic queries. This data repository should not only support raw data but should also integrate resulting artifacts. This is for instance required to be able to compare the different results of the same scan at the end of the pipeline, i.e., the ones which got reconstructed by BART versus those reconstructed by the scanner. This also allows to track and compare differences due to different hyperparameters.

Another requirement for the data repository is that it can be built up over time, without the need to have the full processing pipeline at the beginning. Since doing MRI scans is very time-consuming, they are done over a longer period. In the meantime, the involved researchers can continuously work on novel processing techniques and improve existing ones.

Performance Each of the described processing steps is highly compute intensive and therefore requires scalable and performant compute clusters. In addition, some of the involved researchers already have extensive HPC experience, and highly sophisticated pipelines implemented to efficiently process MR images on these systems. Thus, HPC integration is a key requirement for this project, both from an efficiency and a usability perspective. However, processing large data sets, particularly with iterative methods, as they are often used in machine learning, is not only compute but also data-intensive. Therefore, the available storage tiers need to be efficiently used.

Scalability To serve this big data scenario, it is not only enough to consider the performance of a single node, or a single task, but one needs to ensure scalability as well. This concerns all involved parts, i.e., the processing infrastructure, and the data repository, which combines storage capacity and metadata management. However, not only the capability to scale out is important, but also the cost-effectiveness when doing so.

Generality Even from the very start of this project, four different departments are involved, and further collaborations can be anticipated. Each of these groups, and the individual researchers within, have different ways of working, different data, and different tools, which entails different needs and foci. This diversity makes it impossible to simply deploy one of the many data management systems available, which are tailor-made with regard to a very specific target audience, and therefore contain different preconceived assumptions about the later usage. Even if a system can be found that fulfills the current objectives, it might not work for future collaborations. Typically, this problem leads over time to a sprawl of isolated data silos, i.e., data management systems that are only accessible and usable for specific use cases and user groups, thus limiting the reusability of their data and the collaborations between different research teams. Overcoming this gap and providing a unified platform to enable all researchers to collaboratively work on the overarching workflow is one main target of this project. Therefore, a generic system is required, which makes no assumptions about the data, or the processes at the design level.

Data Modeling When integrating data into a data repository some metadata modeling is required. Otherwise one can quickly lose the maintainability. One simple

example to highlight the importance of data modeling are the image-space MR images within this project. These can be an artifact created by BART or could be reconstructions made by the MRI scanner, which can be interpreted as some kind of raw data. It is of utmost importance to differentiate between these two data types, although both will be for further processing in the same data format, will show the same subject, and share the same semantic metadata. The only difference will be in the data lineage.

The data modeling step should be quite generic, and should not be done with one single scenario or data flow in mind. This can be further substantiated by the view, that not only the k-space images can be considered as raw data, but also the scanner reconstructions. Therefore, data modeling is required which is flexible enough to map those scenarios into an intuitive schema.

Continuous Integration into Scientific Workflows As already motivated, there are a range of workflows that scientists execute as part of their research. In some simple cases, raw data is extracted from a single source and is then automatically processed by a few tasks. On a high level, this is the case in the motivating example shown in Figure 2.2. However, the above-stated goal is to be generic without any preconceived assumptions. Therefore, the possibility for more difficult workflows, where tasks are not executed in a strict linear order but can be modeled in a directed acyclic graph to allow for task parallelism, should be accounted for from the beginning. In other cases, not all steps are necessarily predefined automated processing tasks but some manual interaction might be required at certain points. Lastly, there might also be some further analysis tasks that are physically performed in a lab. All of these diverse scientific scenarios are possible workflows that need to be able to be integrated into a single data lake to serve as the central data repository. This means a data lake needs to be continuously adaptable to integrate new scientific workflows in order to provide a central data repository with integrated data.

It is very common, but also specifically within this project, that not all steps of a scientific workflow are done by a single researcher. Therefore, data sharing is important to enable and support the envisioned collaborations.

Data Lifecycle Management The successful integration of scientific workflows can further be used to support researchers to follow good scientific practice, which can be done with comprehensive data lifecycle management. This is comprised of actionable information about the duration over which the data has to be stored, or what a sufficient data and metadata quality is within these scientific workflows. One particular focus within the metadata quality is the provenance information. A proper data lineage will allow for retrospective comprehensibility and reproducibility, which is currently a major challenge in science and was even called a "Reproducibility Crisis" [8]. Considering the processing of big data on HPC systems also proper storage allocation across the available storage tiers becomes more important. Data should always be backed up on cheap and scalable mass storage with high durability, while only hot data should reside on fast and expensive filesystems closer to the compute clusters.

FAIR Data One increasingly important aspect of data management is the *FAIRness* [209], i.e., how much the practiced data management adheres to basic principles regarding the Findability, Accessibility, Interoperability, and Reusability (FAIR). These key points are subdivided into 15 high-level criteria. The main goal is to enable

machine-actionability, however, the FAIR principles also ensure sustainable data management which enables the ability of third parties to find and reuse already existing data. These aspects are an increasingly important factor for funding and publishing research, therefore scientists want to be compliant, but also want and need to focus on the actual research. Therefore, strictly following the FAIR principles in a project from the beginning can be perceived as a distraction from the actual work. In addition, these principles are not limited to the data management itself but also extend to the entire infrastructure. Therefore, to fulfill the FAIR principles an entire ecosystem that supports and eases the burden of researchers to individually come up with solutions to this generic problem is required.

Modularity From a very abstract view, the envisioned data repository, i.e., the data and process management system has to integrate three basic services: storage, metadata management, and compute capabilities. Based on these services, others may be necessary, e.g., for governance or version control, or access to more domain-specific services. Since all of these services are subject to change due to technological progress or domain-specific requirements, a modular design is advantageous which will not only allow for necessary substitutions but will also help to achieve the envisioned generality. This will in the end enable researchers to overcome data silos, which are prohibiting cross-silo analysis. A modular design is also specifically required for this specific MRI use case, since small data sets should be processed in a cloud environment, whereas larger data sets should be processed on an HPC system, according to the original project proposal.

Security One particular challenge is the management and the processing of sensitive data. Specifically for MR images, it is not possible to anonymize the data. For example, an MRI scan showing the head of a person shows her/his face and is therefore immediately personalized. Although there exists the possibility to artificially cut out the face, so-called de-facing, this is not possible for k-space data, which also contains this personalized information. For some other processing tasks, de-facing might be unfeasible because it introduces artificial asymmetries that impact the accuracy of the results. Therefore one needs to ensure for these data sets that the full data sovereignty stays with the user during the entire data lifecycle. This means, that no illegitimate users, attackers, or even admins should get access to this kind of data at any time.

2.3 Proposed User Workflow and System Specification

Based on the previously made requirement analysis and the identified key aspects, a suitable system needs to be found that fulfills these defined requirements. Due to the rather diverse user groups and the intent to be able to expand and include even more researchers and use cases, a highly specific, purpose-built data management system can be excluded. Instead, the envisioned target system should be able to integrate all kinds of data in a similar fashion. A data warehouse can be rejected due to the high commitment of storing the raw data. Instead, a data management system is required that can be used to build up a research data repository over time, without the need to have the full processing pipeline in place at the beginning. Thus, a data lake is proposed as a suitable candidate for this intended use case, since it is generally generic and retains raw data in its original format.

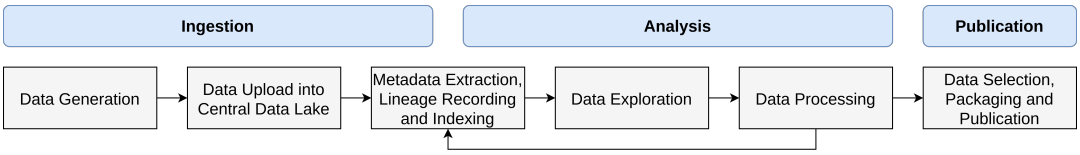


FIGURE 2.3: Proposed user workflow on a data lake.

In order to further derive a suitable data lake design, which fulfills the in Section 2.2 defined objectives, a generic and high-level user workflow is described which models the requirement specification artifact. From this description, further technical and conceptual requirements and features can be derived. On a coarse-grained view, there are three different stages, i.e. *Ingestion*, *Analysis*, and *Publication*. Those three stages can contain a total of six distinct tasks, yielding the proposed, overarching user workflow depicted in Figure 2.3.

The first stage is the data ingestion. Here, within the general philosophy of a data lake, the raw data is uploaded to the data lake. Since a data lake is envisioned that supports multiple data sources and serves as the central data repository, proper metadata management and data cataloging are key to ensuring maintainability [57]. Therefore, at least a minimal set of metadata needs to be extracted from the raw data. The integration of different data sources is required since even from the very beginning the k-space data and the scanner reconstructions can already be considered as two data sources, and it is mentioned by the researchers that more could be required in the future like Electroencephalography (EEG) data. Once the raw data from diverse sources are integrated into this single data lake, further analysis can be done. This can contain a manual data exploration step, where users can explore the available data sets within an overarching data catalog, and can refine their queries to select the desired subset. Once a suitable subset of the available data is found, the actual processing can start. These processing tasks then create again some data products, also called artifacts, which should then be reingested into the data lake. Particularly when ingesting artifacts into the data lake, it is key to properly record the lineage to ensure retrospective comprehensibility and ideally even reproducibility. These newly ingested artifacts can now again be used as input data for consecutive processing tasks, thereby enabling the execution of workflows. Once a final product is derived, researchers usually want to publish their results. Assuming that researchers sometimes have to run the same analysis multiple times with slight deviations, the publication step requires the ability to concisely select the desired result, and probably also the concise workflow used to create this artifact.

In order to enable this envisioned user workflow when working on a data lake, a set of requirements can be derived:

Data Generation: Not only when ingesting data from different sources, but also when ingesting data from a single source, the capability for provenance recording might already be required to support an overarching scientific workflow. For instance, when performing different measurements with the same instrument, it is important to link the resulting raw data against the experimental protocol, e.g. in the form of an externally managed electronic lab notebook, automatically written metadata, or in the form of explicit user annotations.

Data Upload: In order to provide easy access to the data lake from any client, it is important to support common communication protocols, ideally even multiples.

This is necessary, since access and the capability to install required software on a workstation in a lab, or a hospital, can be very limited.

Metadata Indexing: Structured indexing of metadata generally requires dedicated data modeling. The aim of data modeling in the concept of data lakes in general is to ensure the findability, quality, and comprehensibility of the data, especially for unstructured data, within the lake. Since no fixed schema should be enforced, it needs to support continuous updates and a multitude of different data models. It should support domain-specific, user-defined attributes and should, if desired, support the use of an ontology, or thesaurus.

Data Exploration: This step can be used by users to get a better understanding of the available datasets. The particular challenge here is to provide users with a good overview, even in the context of big data. This requires, that users are able to inspect data and data sets in different granularities, or to be more colloquially: users should be able to zoom in and out of the data. For this, views providing overarching information about a data set, for instance by using aggregations, can also be used.

Data Processing: The envisioned data lake should not only provide large and integrated data sets but should also offer scalable compute capacity. These compute resources should not be limited to a single ecosystem that can serve the previously described use case, but the integration should support different systems, like cloud infrastructure for smaller tasks, or HPC systems to support even the most compute-intensive tasks. This will also ensure the capability to support future use cases with different compute requirements. Input data should either be selected by users directly, or indirectly via domain-specific queries. Particularly for experienced users, the HPC system and the provided software should be usable with minor interactions with the data lake, while inexperienced users should be offered a higher degree of automation, particularly with respect to provenance auditing. Proper lineage recording should be guided to enhance good scientific practice as well as the maintainability of the data lake.

Data Publication: Since reproducibility is an increasingly important topic during the publication process at journals or conferences, open data including any code and intermediate results, sometimes even along with a reproducible workflow, are either on a mandatory or a voluntary basis asked for. Depending on the venue and the domain, various specific data formats and accompanying documents are required. Therefore, data, artifact, and workflow selections should provide these datasets in an interoperable format, which can be easily ported to other formats, like research objects [36] or *BagIt* [104]. However, being able to extract data for publication isn't enough, since it is usually required to keep data for at least 10 years archived.

Data Lifecycle: All of the aforementioned steps are part of the overall data lifecycle and therefore all of these steps combined yield the overarching data management. That means, for instance, each of these steps needs to support the FAIR principles.

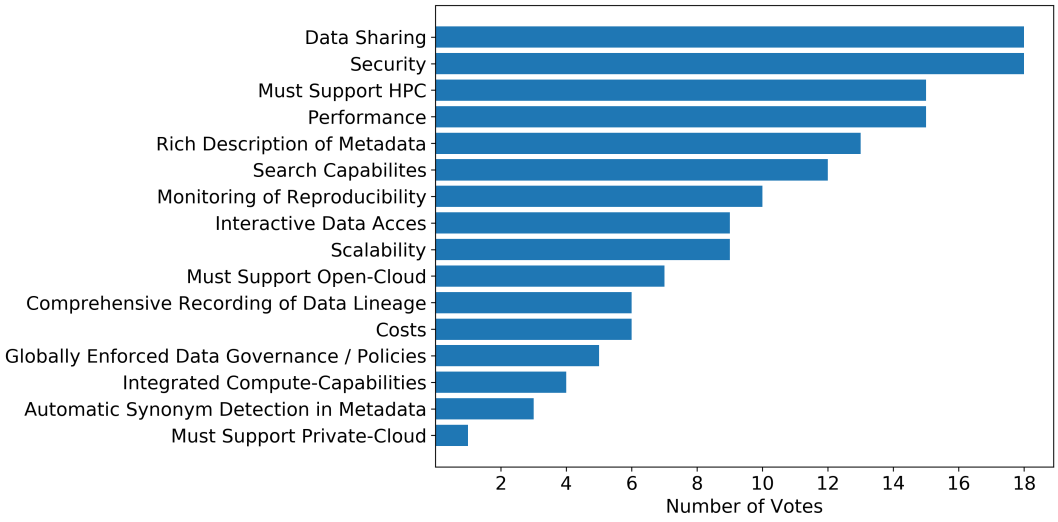


FIGURE 2.4: Result of a poll taken during a data lake BoF at ISC22 with more than 30 participants contributing to the questionnaire.

2.4 External Review of the Objectives

Until this point, the entire analysis with regard to these requirements is driven by this single "Big Data MRI"-project. However, one of the identified objectives is generality, meaning that the developed data lake should be able to extend its scope to cater to new scientific projects. In order to prevent a too narrow view on this topic, the 10 objectives described in Section 2.2 are simplified and concretized with regard to their underlying feature. Additional possible attributes, like the cost, are also added. Then, these features were included in a questionnaire that was given to the audience of the Birds-of-a-Feather (BoF) session "Large-Scale Data Management with Data Lakes" at the International Supercomputing (ISC) in 2022 which counted about 50 participants among which more than 30 actively participated in the questionnaire. The audience can be assumed to be very international and consists of mostly HPC admins and only a few HPC users. Among other things, the audience was asked to rank the importance of the identified data lake features. Each participant had a maximum of 5 votes to distribute over 16 different features, see Figure 2.4. In total 151 votes were counted, ranking data sharing and security as the most important features, while private cloud support was ranked the least important, which was one of the key requirements of the MRI project. Generally, the result is in good agreement with the objectives defined in Section 2.2. Interestingly, the total cost, advanced metadata handling, and cloud support have been deemed less important. In addition, the need for interactive data access has only been identified for hot data within the project but is felt generally more urgent by the audience of the BoF. In general, one can also see, that the capability to model the available data was considered very important, considering the number of votes from *Rich Description of Metadata*, *Search Capabilities*, and *Monitoring of Reproducibility*. From the top 10 voted features only the *Interactive Data Access* has not been explicitly stated as an objective. Considering the high participation and voting rate, one can generally recognize this as an agreement of the participants with the suggested features.

Chapter 3

Background and Related Work

After a data lake is identified in the previous chapter as a suitable platform for data and process management, further analysis of the applicability of these systems for the particular use case is necessary. For this, further background information about data lakes is provided in Section 3.1, followed by an in-depth literature review to determine the current state of the art and establish a solid knowledge base. Different use cases are analyzed in Section 3.2 to compare the defined objectives and intended use case, with established systems. Then, the four identified key properties of data lakes are investigated, i.e., the available architectures in Section 3.3, the different metadata modeling techniques in Section 3.4, the provenance auditing mechanisms in Section 3.5, and lastly the compute integration in Section 3.6. This chapter concludes with an analysis of the different discussed systems in Section 3.7. Parts of this chapter are published in [208].

Data lakes have already been researched for over a decade since their first formal introduction by James Dixon in 2010 [56]. During this time, extensive work has been done on top of the original idea to retain all raw data in its native format. This has led to a vivid evolution of data lakes, including their definition and requirements. For instance, in 2013 *Pivotal* [32] presented a business data lake aimed to substitute the commonly utilized enterprise data warehouses. They proposed to use a data lake that is divided into multiple tiers and receives data from multiple sources. It was also recognized, that in order to provide a unified view and unified data management capabilities, that metadata has to be maintained in a data catalog. This is an extension of the original proposal from Dixon [56] who has originally only argued with a single data source. The importance of proper semantic metadata management was also highlighted a year later by Gartner, saying that "Without descriptive metadata and a mechanism to maintain it, the data lake risks turning into a data swamp" [64]. They also recognized the particular importance of lineage recording when ingesting artifacts. In 2014 this criticism caused James Dixon to revisit his original idea and address these concerns [57]. He defended the single data source, stating that one should have multiple data lakes for multiple data sources. Integrating data from different sources would yield a "Water Garden", not a data lake. However, he agreed that extending the concept to multiple data sources is easily possible, but would require proper metadata management. Dixon also highlights that no assumptions during the ingest about the data must be made.

3.1 A Brief History of Data Lakes

At the beginning, there was a high association between Hadoop and data lakes and was at times even used interchangeably [98]. At first, this connection was probably

formed by the timing of the original announcement after the Hadoop world in 2010 [56]. But Hadoop also offered horizontal scalability not only for storage but also for compute. It also provided an integrated way to define parallel tasks, e.g., to infer a schema on raw data, or for further analytical processing. Access to a Structured Query Language (SQL) like interface on a Hadoop cluster was even more streamlined with the evolution of the entire ecosystem around Hadoop, like the integration of Spark [215], or Hive [195].

Although it was not explicitly stated, this approach implemented a flat namespace, as it did not infer a conceptual hierarchical order on the stored data. This ensured a high degree of data integration since they were homogeneously accessible from a unified interface. The envisioned practice of performing all processing steps on-demand, including some commonly shared preprocessing, has led to a state where some processing tasks were repeated multiple times. Christian Mathis challenged this consent, pointing out the wastefulness of recomputing particularly intermediate results [121]. He suggested that users of data lakes should carefully consider whether recomputing a result or storing an already transformed data set in a more structured format, like a parquet file, using a connector, like Hive, or pushing it into an external database altogether, is more efficient.

This idea was further developed into the *Zone Architecture*. Here, the data lake as the central data repository, is split up into different, separated zones. The basic idea is that data is sorted into these distinct zones based on the pedigree of processing they were subjected to, e.g., if the data is raw, cleansed, or aggregated. Although a multitude of different systems are proposed [70, 118, 153, 165, 178, 219], they all share a similar approach. They all have a *Raw Data Zone*, where unprocessed data is retained in its original format. Pre-processed data is collected in a separate *preprocessed*, *refined*, or *staging* data zone. Based on the intended purpose of the data lake the exact number and usage of zones differs from case to case to adequately accommodate potentially diversely processed data. This means, that each zone might have an independent and unique data governance, like general access rights, or quality controls. Depending on the individual implementation of a data lake, which is built using the zone architecture, these zones can be realized using different systems, both in hardware and software. Therefore, the implementation and maintenance expense for system administrators is increased, and similarly, also the complexity for data scientists in their daily usage since data is scattered across different systems, potentially requiring different access methods due to different software stacks and providing different capacities, both in storage and compute. Thus, this evolution of data lake architectures resulted in more heterogeneous data lakes compared to the simpler Hadoop approach used before.

The following stage of the architectural evolution of data lakes was shaped by large, public cloud providers who offered proprietary but integrated solutions. Basically, every Cloud provider had their own, locked-in offering, including IBM [41], Microsoft Azure [163], AWS or Google. Although each offering consists of its own proprietary software stacks, the general approach is still similar. The central contact point is a scalable, and cost-effective cloud storage, like Amazon Simple Storage Service (S3), which is an object store, offering access via a Representational State Transfer (REST) Application Programming Interface (API). This cloud storage service is then extended by a database, where all objects, i.e., data files, are cataloged. In order to ease the data ingest from diverse sources and data staging into different compute services to perform advanced analytics on the data sets, dedicated data transfer tools are used. Last but not least, a governance service can be used to enforce different policies, like access control lists, homogeneously across all involved

services. Since these different vendor-specific tools offer a high and seamless interoperability within each ecosystem, this has homogenized again the resulting data lake architectures. Additionally, it brought down the cost and led to an increasing adoption, particularly by large companies.

Based on this recap one can see that the architecture, i.e., how a data lake is conceptually built and organized is key in determining its features and user interactions. In addition, metadata modeling was also very early recognized as a mandatory requirement when integrating data from multiple, possibly diverse data sources. One particularly important aspect of metadata modeling is the provenance, which is also related to the overall integration of computing capacity. As motivated by the history of data lakes, the main differences in the development of data lakes were done in these four areas. The last three of these topics also match well with the objectives described in Section 2.2 and the results of the questionnaire discussed in Section 2.4. Therefore these four aspects are in detail examined in the following literature review. Afterward, the results are analyzed towards their applicability of the in Section 2.2 defined objectives.

3.2 Use Cases from Literature

Data lakes were early on adapted by the industry to support their Business Intelligence (BI). An exploratory study among 12 experts on how to implement data lakes in various enterprises found that data lakes are primarily used as a staging area for a data warehouse, as an archive for a data warehouse, as a sandbox for data scientists doing advanced analytics on a local workstation, and as a data source for direct reporting [112]. The main benefits are hereby supposedly the speed at which one can ingest raw data since many people just "dump all the data in there" and the access to raw, unaltered data [112]. In these use cases, common challenges were a lack of data stewardship, data governance particularly for confidential data, data quality, and the expertise of the users since a data lake is "complex to understand". Similarly in [189] a Hadoop-based data lake is presented for BI, which is used as a staging area for a data warehouse. In addition, it is also used for unstructured data which will not be loaded into the data warehouse. Instead, data analysts access the data lake directly for advanced analytics.

In contrast to these BI use cases, Rihan Hai describes a scientific use case from the *Internet of Production*, which serves as a foundation for research and development, instead of supporting decision-making [74]. Here, three different milling scientists are working in different labs with different milling machines which have different sensors and therefore create different output data. Although they all have different research questions and therefore different analysis tasks, they want to be able to compare their data. Therefore they need to be able to find related data sets to the ones they have individually ingested and which they are working on. In addition, they all create per-milling machine diverse data, like pressure and temperature data, or video streams, which need to be integrated to allow for easy ad-hoc access. Retaining raw data is key to enabling continuous support amid changing research questions.

This last example clearly motivates how a data lake can be used to provide a platform for researchers to collaborate, but also shows that it can be used by each scientist as a data repository. The described scenario in [74] can be transposed from milling machines to MRI scanners since there are certain parallels to the use case described in Chapter 2. For instance, there will be also slightly different data which

shall be analyzed and compared to each other, e.g., the novel reconstructions to the standard scanner reconstruction, similar to the different data sets created by the different milling machines. In addition, the inclusion of different MRI scanners in the future might also be possible. The successful use of a data lake in the use case described in [74] can be considered as a confirmation that a data lake is a suitable platform for scientists who want to collaboratively work on joint workflows and data sets.

3.3 Architectures

As of today, a lot of development and analysis was conducted in the area of *data lake architectures*, where the so-called *zone architecture* [153, 165], including the *pond architecture* [88], became the most cited and used. These architectures have already been surveyed [74, 167], and a *functional* architecture is proposed by both of them, while a *maturity* and a *hybrid* architecture are only discussed in [167]. These surveys, however, did not include recent works like the definition of a zone reference model [66].

3.3.1 Definition of the Term Data Lake Architecture

The term data lake architecture is defined to represent the comprehensive design of a data lake, including the infrastructure, data storage, data flow, data modeling, data organization, data processes, metadata management, data security and privacy, and data quality [65]. In this data lake architecture framework, only the data security and privacy, and the data quality are considered to be purely conceptual, whereas the other aspects include a conceptual and a physical, i.e., system specific, dimension. As stated in [117] more generically, a data lake generally has a logical and a physical organization. In this work, the term *data lake architecture* is referred to only with respect to the conceptual organization of a data lake in the highest level of abstraction, since this should make this work more comparable to the existing literature, although there exists a strong dependency on other aspects of a data lake, like the metadata modeling.

3.3.2 Zone Architecture

The general idea to divide a data lake into different zones arises from the necessity to automatically run standardized pre-processing pipelines and avoid to re-run these tasks on every data access [121]. Therefore, one keeps and organizes the resulting pre-processed data, and makes it available to subsequent processing steps, like reporting, *Online Analytical Processing* (OLAP), and particularly advanced analytics. This is achieved by assigning data to different zones based on the degree of processing, and sometimes the intended future use case. Therefore, it is common to have a *raw data zone*, where, according to the original idea of a data lake, data is retained in its native format. Pre-processed data is then usually collected in a dedicated zone for *pre-processed*, or *refined* data, sometimes called *staging zone* [219] or *processing zone* [165]. Data that requires additional governance can be collected in a dedicated zone, called *trusted zone* [219], or *sensitive zone* [70].

The most extensive analysis of the *zone architecture* analyzes five different data lakes [70, 118, 153, 165, 178, 219] with respect to their design differences, specific features, and their individual use cases in order to derive a generic meta-model for a zone, and to specify a zone reference model based on it [66]. It is identified that a

zone is uniquely defined by the characteristics of the data contained in that zone, the intrinsic properties a zone enforces on the data, the user groups that are intended to work in that zone, the modeling approach to organize the corresponding metadata, and the data sources as well as destinations. In the presented zone reference model, Giebler et al. propose to split the zones up in a *raw zone* and a *harmonized zone*, which is use case independent, and a use case-specific *distilled zone*, which serves data to the final *delivery zone*, to support reporting and OLAP tasks, and a *explorative zone* to support advanced analytics. Each zone hosts a *protected* area for data that requires special governance. The actual implementation and the deployed systems can vary in each zone, including the storage, the metadata model, and the metadata management system itself. This entails synchronous, that also the user interface potentially changes with each zone.

3.3.3 Lambda Architecture

The *Lambda Architecture* is proposed to enhance the capability of a data lake to process data streams in near real-time instead of fully ingesting hot data into the data lake and performing batch-processing with a certain time delay [121]. However, retaining all raw data in its native format is the core idea of a data lake. In order to resolve this contradiction, the Lambda Architecture [206] implements two processing streams in parallel. Here, data is being processed in near real-time in the *speed layer*, whereas the *batch layer* ingests data into the data lake and performs some predefined processing steps. There have been numerous implementations proposed for a data lake utilizing the Lambda Architecture, particularly at the early stages of the data lake research [9, 82, 200]. However, the following two particular works are presented which are building on top of public cloud offerings and represent more state-of-the-art systems.

A lambda architecture is used by [135] to build a data lake for smart grid data analytics using Google's cloud computing as Infrastructure as a Service (IaaS). Here, the data is collected by a dedicated *Data Collecting Layer*, in this particular case realized by Apache *Flume*[1]. From there, the data is sent to the core of this specific data lake implementation, a Hadoop Cluster. The master node stores the data on the Hadoop Distributed File System (HDFS) [28], and computes arbitrary, predefined functions using MapReduce [52]. The speed layer is implemented using *Apache Spark* [216]. The serving layer combines the output of the batch and the speed layer and provides a batch view of the relevant data, using e.g., *Hive* [196], *Impala* as shown by [110], and *Spark SQL* [6].

Another work compares three different implementations using the Software as a Service (SaaS) offerings with a focus on serverless delivery of three different public cloud providers, i.e., Google Cloud Platform, Microsoft Azure, and Amazon Web Services Cloud [157]. On AWS the speed layer accepts data via Kinesis Data Streams and processes them using Kinesis Analytics and AWS Lambda. The results are stored in a dedicated *S3-Speed-Bucket*. Similarly, the batch layer uses Kinesis Firehose to ingest data into AWS Lambda, from where it is stored in an *S3-Batch-Bucket*. From here the data is read by AWS Glue and stored in an *S3-Result-Bucket*. The serving layer is realized by Athena which reads the data from both, the *S3-Result-Bucket* and the *S3-Speed-Bucket*. In the Google Cloud implementation, data is ingested by Pub/Sub to the speed and the batch layer, which are both realized using Dataflow. On the batch layer, an additional Datastore is employed to retain the raw incoming

[1] https://flume.apache.org/

datasets. The serving layer uses BigQuery. On Microsoft Azure, data is received and ingested by EventHub. The speed layer uses Stream Analytics and forwards directly into the Serving Layer which is Cosmos DB. The batch layer also uses Stream Analytics to store the raw data into Data Lake Store [163]. From there it is read by Data lake Analytics, which also stores its results in Cosmos DB.

3.3.4 Lakehouse

Lakehouses are a consequence of the general observation that in some cases the raw data from a data lake is used as an input for an ETL process to populate a data warehouse [5]. The first step into a more unified setup is proposed by *Delta lakes* [4], which provides ACID (Atomicity, Consistency, Isolation, Durability) transactions on cloud object storage for table stores. These tables can be accessed from different systems, like Spark, Hive, or Presto [177]. This approach introduces, among other things, the advantage of still separating storage and compute. Lakehouses offer on top of ACID transactions direct access to the storage with traditional database semantics, e.g., SQL, using open file formats like *Apache Parquet* [201] or *ORC* [2]. Therefore, a metadata layer on top of the cloud storage can provide convenient SQL-like access to tables, while compute-intensive, non-SQL code, like machine learning, can directly access the files on the storage devices and thereby get higher performance.

3.3.5 Functional and Maturity-based Architectures

The classification into *functional* and *maturity*-oriented data lake architectures, unlike in the case of the zone, lambda, and lakehouse, do not represent yet another design concept, but rather serve as an improved way for classifying the different architectural approaches. The goal is to allow for a more modular comparison of existing data lake solutions and to better plan the data life-cycle as well as to help match the individual functionality of the architectural pieces, which are building up the data lake, like zones, on the required infrastructure.

Within a *functional*-based architecture classification, the data lake is analyzed towards its operations which are performed on the data while moving through the general data lake workflow. These architectures are represented in layers, where a three-layer design consisting of *ingestion*, *maintenance*, and *exploration* is proposed in [74]. The corresponding functions are then sub-grouped within these layers. A similar definition is provided in [167], where the four main components of a data lake are defined as *ingestion*, *storage*, *processing*, *querying*.

Following the maturity-based architecture classification, the degree of the processing of the data is the central point of consideration. This classification is only helpful in the discrimination and organization of different data sets, however, it completely lacks the consideration of workflows and processing capabilities. Highlighting the advantages of careful data life-cycle planning a *hybrid architecture* is proposed alongside the *functional* and *maturity* based classifications [167]. Within this architecture, the individual components are uniquely identified by the data refinement and the possible functionality, that can be performed on the specific data set.

[2] https://orc.apache.org/

3.4 Metadata Modeling

Proper metadata management is key for preventing a data lake from turning into a data swamp and thus is the most important component to ensure continuous operation and usability [98, 203]. Due to the generally flat hierarchy and the requirement to store any data in its native format, there is always the risk of losing the overall comprehensibility of the data lake. This comprehensibility is lost, if data cannot be found or the relationship to other data sets cannot be retrieved. One of the most severe consequences of this is the inability to define concise queries to select the data one is looking for in a fine-grained manner. As a consequence, numerous metadata models and systems tailor-made for usage in data lakes are proposed. These models and systems originate from different use cases and represent various viewpoints, and therefore differ regarding their feature sets. From this wide variety of available options, a few distinct works have been selected and are discussed in the following sections.

3.4.1 Data Vault

Data modeling in a *data vault* was proposed by Linstedt in the 1990s and published in the 2000s to allow for a more agile metadata evolution, i.e., the continuous development of the metadata schema, in data warehouses compared to star or a snowflake schemata [111]. This ensemble modeling uses traditionally relational database systems and combines the third normal form with the star schema. All data is stored in three different types of tables. *Hubs* describe a business concept and are implemented as lists of unique keys, and can be populated by different data sources. *Links* describe relationships between the aforementioned hubs. *Satellites* contain all attributes which describe the properties of a hub or a link. Evolving the data vault over time mainly implies adding additional satellite tables to links and hubs. Therefore, there is no need to migrate existing tables, which facilitates the continuous addition of metadata over time.

Due to these characteristics of the data vault concept, it is also applied in data lakes. For example, in [139] the definition of a data vault for a specific use case is exemplified and the advantages of porting it to a NoSQL database by comparing benchmark results to a SQL database are discussed. They also demonstrate how new data sources can be added by defining new hubs, links, and particularly satellites. Another work proposes to split the one central, data lake-wide data vault up into three distinct sub-data vaults: the *Raw Vault*, the *Business Vault*, and the *Data Mart*, whereby the latter does not necessarily need to be modeled in a data vault, but could also be a flat table, or a star schema [67]. The authors reported that the agile approach along with the ability to make incremental updates serves well the needs for a data lake implementation. However, they pointed out that it can be hard to enforce business rules across the independent sub-data vaults, that they use, that managing ambiguous keys cannot finally be solved, and that high-frequency data can critically inflate satellites.

3.4.2 *GEMMS*

GEMMS is proposed as a *Generic and Extensible Metadata Management System* with a particular focus on scientific data management and, in this context, specifically for the domain of live sciences [162]. The key component of GEMMS is an abstract entity called *Data Unit*, which consists of raw data and its associated metadata. It is

stated, that the main advantages are, flexibility during ingestion and a user interface that abstracts singular files. These *Data Units* can be annotated with semantic metadata according to a suitable ontology. However, the core is described with *structure metadata*. Mappings are only discussed for semi-structured files, like Comma Separated Values (CSV), Extensible Markup Language (XML), or spreadsheets, however, it seems straightforward to extend this in other use cases.

3.4.3 *MEDAL* and *goldMEDAL*

A graph-based metadata model is presented in [168], where a subset of data, called an object, is represented as a hypernode that contains all information about that particular object, like the version, semantic information, or something called *representations*. *Representations* present the data in a specific way, for instance as a word cloud for textual data. There is at least one representation required per object, which is connected to this object by a *Transformation*. These representations can be *transformed*, which is represented as a directed edge in the hypergraph. This edge contains information about the *transformation*, i.e., a script description or similar. Data versioning is performed at the attribute level of these hyperedges connecting two different representations. Additionally, it is possible to define undetected hyperedges representing the similarity of two objects, provided that the two data sets are comparable.

This approach is revised in [172], where the concept is simplified to only use *data entities*, *processes*, *links*, and *groupings*. *Processes* also generate new *data entities*, dropping the rather complicated idea of *representations*. These concepts are again mapped on a hypergraph. Both models require global metadata, such as *ontologies* or *thesauri*.

3.4.4 *CODAL*

This data lake, and in particular the utilized metadata model called *CODAL* [168] is purpose-built for textual data. It combines a graph model connecting all ingested data sets with a data vault describing an individual data set. One core component is the *xml manifest*, which is divided into three parts: i) *atomic metadata*, ii) *non-atomic metadata*, and iii) a division for *physical relational metadata*. Metadata of the first category can be described as key-value pairs, whereas *non-atomic metadata* only contain references to a specific entity on a different system, they are "stored in a specific format in the filesystem" [168]. Additional information about the link strength which is modeling the *relational metadata* is stored in a dedicated graph database. Here, each node represents one document with a reference to the corresponding *xml manifest*.

3.4.5 Network-based Models

A network-based model, which extends a simple categorization [150] into three distinct types of metadata, i.e., *Business Metadata*, *Operational Metadata*, and *Technical Metadata*, is proposed to improve the data integration of different data sources ingesting heterogeneous and unstructured data into a data lake [54]. Here, the notion of objects, or nodes in the resulting graph, is used as well, which is defined by the corresponding source typology. Based on these objects, links are generated, containing a *structural*, a *similarity* or a *Lemma* [138] relationship. In this approach, a node is not only created for each source but also for each tag used in the *structural relationship* modeling. Lexical similarities are derived if two nodes have a common lemma

in a thesaurus, while string similarities are computed using a suitable metric, which is a *N-Grams* in this particular case [158]. Similar nodes are merged. Due to this merge, synonyms in user queries can be detected and appropriately handled.

3.4.6 *CoreKG*

CoreKG [13] contextualizes the metadata in the data catalog. To this end, four features has been identified to constitute this curation service [15]: *Extraction*, *Enrichment*, *Linking* and *Annotation*. The *Extraction* functionality extracts information from the raw data containing natural language, like the names of persons, locations, or organizations. *Enrichment* first provides synonyms and stems from the extracted features by using lexical knowledge bases like *WordNet* [128]. These extracted and enriched features then need to be linked to external knowledge bases, like *Wikidata* [202]. This enables *CoreKG* to understand, if, for instance, the name of a certain politician is extracted, to link against the corresponding country that politician is active in, i.e., to set it into context. Additionally, users can also annotate the data items.

3.4.7 *GOODS*

GOODS is the internal data lake of Google [77, 78]. It is unique compared to all other systems presented since it gathers all its information in a post-hoc manner. This means, that the individual teams continue working with their specific tools within their established data silos, while *GOODS* extracts metadata about each dataset by crawling through the corresponding processing logs or storage-system catalogs. The central entity of this data lake is a data set, which users or a special data stewardship team can additionally annotate. These datasets are then connected by a *knowledge graph* [183] to represent their relationships. Within these relationships, the *dataset containment* enables to split up data sets, as it allows for *bigtable column families* [34] to be a data lake entity themselves, along the entire *bigtable*. Due to efficient naming conventions for file paths, *GOODS* can build up *logical clusters*, depending on whether they are regularly, e.g., daily, generated, if they are replicated across different compute centers or if they are sharded into smaller data sets. In addition, the data sets are linked by *content similarity* as well. Since the entire data lake contains more than 20 billion data sets with the creation/deletion of 1 billion data sets per day, no pairwise similarity can be performed. Instead, locality-sensitive hash values for individual fields of the data set are generated and compared.

3.4.8 *Constance*

Constance [73] is a data lake service, that extracts explicit and implicit metadata from the ingested data, allows semantic annotations and provides derived metadata matching and enrichment for a continuous improvement of the available metadata, and enables inexperienced users to work with simple keyword-based queries by providing a query rewriting engine [75]. As it is typically done in data lakes, data is ingested in raw format. Then, as much metadata as possible is extracted from it, which makes structured data like *XML* easier to ingest since schema definitions can be directly extracted. In the case of semi-structured data, like JavaScript Object Notation (JSON) or CSV files, a two-step process called the *Structural Metadata Discovery* is necessary. First, it is checked, whether or not metadata is either encoded in the raw file itself, like a self-describing spreadsheet, or if metadata is encoded in the filename or file path. In a second step, relationships are tried to be discovered during

the lifetime of the data lake between the different datasets, for instance, based on the frequencies of join operations. *Semantic Metadata Matching* is provided by a graph model and should use a common ontology. In addition, schemata can be grouped based on their similarity, which is useful in highly heterogeneous data lakes.

3.5 Provenance

One of the generally most important metadata attribute in the context of linked data is provenance [81]. Data provenance or the data lineage hereby contains information about the origin of a dataset, e.g., how, by whom, and when it was created. There has been an effort by the World Wide Web Consortium (W3C) to standardize the representation of provenance information by the use of an *OWL2* ontology, as well as a general data model, among other documents, to complete their specification called *PROV* [16, 130]. Provenance is also becoming increasingly important in science, as it is a natural way to make scientific work more comprehensible and reproducible. This can be exemplified by the adaption of *research objects* [11] and *reusable research objects* [213], focusing even more on precise provenance information and repeatability of computational experiments. Apart from this, provenance is considered key in data lakes, to organize, track, and link data sets across different transformations and thereby ensure the maintainability of it.

3.5.1 *CoreDB* and *CoreKG*

CoreDB [12] and *CoreKG* [13] are data lake services with a main emphasis on a comprehensive REST API, to organize, index, and query data across multiple databases. At the highest level, the main entities of this data lake are data sets, which can be either of type *relational* or of type *NoSQL*. In order to enable simultaneous querying capabilities the *CoreDB* web service is itself in front of all the other employed services. On this layer, queries are translated between *SQL* and *NoSQL*. A particular focus is the lineage tracing of these entities. The recorded provenance is hereby modeled by a directed acyclic graph, where user/roles and entities are nodes while connecting edges represent the interaction. This employed definition is given by the *Temporal Provenance Model* [14] and can answer, when, from where, by whom, and how a data set was created, read, updated, deleted, or queried.

3.5.2 *GOODS*

GOODS metadata model has a particular focus on provenance [77, 78]. In order to build up the provenance graph, production logs are analyzed in a *post-hoc* manner. Then the transitive closure is calculated to determine the linkage between the data sets themselves. Since the data-access events in those logs are extremely high, only a sample is actually calculated and the transient closure is reduced to a limited amount of hops.

3.5.3 *Komadu*-based Provenance Auditing

In [190, 191] a data lake reference architecture is proposed to track data lineage across the lake by utilizing a central provenance collection subsystem. This subsystem enables stream processing of provenance events by providing a suitable *Ingest API*

along with a *Query API*. In order to centrally collect provenance and process it, *Komadu* [192] is used. Hereby, distributed components can send provenance information via *RabbitMQ* and web service channels. These single events are then assembled into a globally directed acyclic provenance graph, which can be visualized as forward or backward provenance graphs. Using this central subsystem, the need for provenance stitching [131] is circumvented.

3.5.4 *JUNEAU*

JUNEAU is build on top of *Jupyter Notebooks*, by replacing its backend and customizing the user interface [218]. It is therefore specifically targeted at data scientists who are already familiar with Jupyter Notebooks. The main constituents of the data lake are tables, or data frames, of which transformations are tracked. Therefore, the notebook itself is considered to be the workflow, and each executed cell within is a task. The provenance information is captured when the code within a cell is transmitted to the used *kernel*. Based on this, the notebook is reformatted into a versioned data flow graph, where procedural code is transformed into a declarative form [91]. Using a modified top-*k* threshold algorithm [61], similar data sets can be found with respect to the individual provenance.

3.5.5 *DCPAC*

In order to manage automotive sensor data, Robert Bosch GmbH has built a data lake [55]. Although the paper mainly focuses on their extensive Data Catalog, Provenance, Access Control (DCPAC) ontology to build their semantic layer, a dedicated data processing mechanism is provided. Data processing is done using containerized applications, which can access data in the data lake, and either create a new data resource from it or curate existing data sets. The semantic data catalog is updated via *Apache Kafka* messages. Hereby, new data items are integrated and their provenance is automatically recorded.

3.5.6 *DataHub*

DataHub [19] combines a dataset version control system, capable of tracking which operations were performed on which dataset and by whom, as well as their dependencies, with a hosted platform on top of it. Hereby, *DataHub* uses *tables* which contain *records* as their primary entities. *Records* consist of a key, along with any number of typed, named attributes. In the case of completely unstructured data, the key can refer to an entire file, in the case of structured or semi-structured files like XML or JSON, the schema can be (partially) modeled into this *record*. These individual *tables* can then be linked to form *data sets* under the specification of the corresponding relationships. The version information of a *table* or *data set* is managed using a *version graph*, i.e., a directed acyclic graph where the nodes are data sets and the edges contain provenance information. In order to query multiple versions at a time, a SQL-based query language called *VQL* is provided, which extends SQL about the knowledge that there are different tables for the different versions of a data set.

Along with DataHub, *ProvDB* [126, 127] is developed. It incorporates a provenance data model [37] which consists of a *Conceptual Data Model* and a *Physical Property Graph Data Model*. The first model considers a data science project as a working directory where all files are either of type *ResultFile*, *DataFile*, or *ScriptFile*. These files can be further annotated by *properties*, i.e., JSON files. This model is then mapped

onto a property graph, where the edges represent the relationship, like a parenthood. Provenance ingestion is possible in three different ways. The first option is to prefix shell commands with *provdb ingest* which then forwards audited information to different specialized *collectors*. Secondly, users can provide annotations. Lastly, there are so-called *File Views*, which allow defining virtual files as a transformation on an existing file. This can be the execution of a script or of an SQL query.

3.6 Compute Integration

Although the first major challenge in building a data lake is the aforementioned metadata management, when scaling towards big amounts of data, (automated) operation and manageability of the data lake become increasingly important. For example, the extraction of metadata upon the ingestion into the data lake requires a scalable and highly automated process, that ideally can be integrated into work- or data flows wherever necessary [121]. As in the case of metadata extraction, it is also sometimes more comfortable to split a complicated analysis into a workflow consisting of different steps. This has the advantage that different parallelization techniques [147, 154] can then be applied to improve the scalability of the implemented analysis. Apart from these challenges, having a data lake is often driven by the wish to perform advanced analytics on big data sets. These tasks are usually quite compute intensive, which makes a tight integration of scalable compute resources necessary. Therefore, integration of compute is one key challenge in the area of data lakes. In the following three different systems are discussed.

3.6.1 *KAYAK*

KAYAK [115, 116] offers so-called *primitives* to analyze newly inserted data in a data lake in an ad-hoc manner. KAYAK itself is a layer on top of a file system and offers a user interface for interactions. The respective *primitives* are defined by a workflow of atomic and consistent tasks and can range from inserting or searching for a data set in the data lake to computing k-means or performing an outlier analysis. Tasks can be executed either by KAYAK itself, or a third-party tool can be triggered, like *Apache Spark* [215] or *Metanome* [152]. Furthermore, tasks can be subdivided into individual steps. By defining a directed acyclic graph, which consists of consecutive dependent primitives, so-called *pipelines* can be constructed. Here, output data is not immediately used as input for a consecutive primitive, but output data is first stored back into the data lake and the corresponding metadata in the data catalog is updated. Users can define a *time-to-action* to specify the maximum time they are willing to wait for a result or preview, or they define a *tolerance*, which specifies the minimal accuracy they demand. A preview is a preliminary result of a step. In order to enable these features, each step has to expose a *confidence* to quantify the uncertainty of the correctness of a preview, and a *cost* function to provide information about the necessary run-time to achieve certain confidence. KAYAK enables the parallel execution of steps by managing dependencies between tasks. These dependencies are modeled as a directed acyclic graph for each primitive. By decomposing these dependency graphs into singular steps, these can be scheduled by a *queue manager*. It enables the asynchronous execution of tasks by utilizing a messaging system to schedule these tasks on a *task executor*, which is typically provided multiple times on a cluster to allow for parallel processing.

3.6.2 *Klimatic*

Klimatic [184] integrates over 10,000 different geospatial data sets from numerous online repositories. It accesses these data sets via Hypertext Transfer Protocol (HTTP) or Globus GridFTP. This is done in a manner that allows capturing of path-based provenance information and can therein identify relevant data sets based on file extensions, like *NetCDF* or CSV. It then pre-processes these heterogeneous data sets to integrate them into this single, homogeneous data lake while ensuring topological, geospatial, and user-defined constraints [27, 45, 60]. The pre-processing is done automatically within a three-phase data ingestion pipeline. The first step consists of crawling and scraping, where *Docker* containers are deployed in a scalable pool. These crawlers retrieve a Uniform Resource Locator (URL) from a *crawling queue* and then process any data found at that URL while adding newly discovered URLs back into the *crawling queue*. Using this approach it is already enough to start with a limited amount of initial repositories, like those of the *National Oceanic and Atmospheric Administration* or the *University Corporation for Atmospheric Research*. After these data sets have been successfully discovered, these are submitted to a *extraction queue*. Elements of this queue are then read by *extractor instances*, and also *Docker* containers which can be elastically deployed. These extract metadata with suitable libraries/-tools, like *UK Gemini 2.2*, and then load the extracted metadata into a *PostgreSQL* database. Using these automated processes, a user is for instance able to query for data in a certain range of latitudes and longitudes, and *Klimatic* will estimate the time needed to extract all data from the different data sets within the specified range, and will then provide the user with an integrated data set using focal operations [179].

3.6.3 Hadoop-based Data Lakes

As motivated in the background provided in Section 3.1, data lakes were much driven by the Hadoop ecosystem. Two of these Hadoop-based data lakes are described in the following [96, 211]. In [96] a data lake system, called *Data Café*, for biomedical data is described. Here, diverse data sources are integrated by the central control layer, which queries diverse data sources, including existing data silos and data warehouses, and stores the results on HDFS, or alternatively on S3 or SQL databases. Data providers can provide additional metadata about the data sources which is stored in Apache Hive. Data lake users can then use Apache Drill [83] to query the data lake. *Data Café* can then use the information about the data sources to find suitable data sets to perform join operations on.

This basic setup is extended to include advanced analytics for electricity usage data [211]. Here, existing data sets are moved via Apache Sqoop to Hive, whereas live data is streamed via Kafka [101] into HBase. Users can search the data lake with Apache Phoenix and Apache Impala [24]. The retrieved data, which is stored in HDFS, can then be processed with SparkMLlib [124].

These two systems are typical representations for Hadoop-based data lakes, which work solely on a flat namespace, which unlike in the zone architecture is not further organized into different subsystems.

3.7 Analysis of Related Work

The related work is discussed with a particular focus on the architecture, metadata modeling, provenance, and compute integration since these are the key aspects identified in Chapter 2. Before the discussed data lakes are mapped on the defined objectives, compare Section 2.2, the related work is compared with regard to common design patterns, and common or individual shortcomings for each category. It can be assumed, that common design patterns can be found in different data lake implementations because they provide convenient solutions to common problems in data lakes. Similarly, there might also be a correlation between common shortcomings and common design patterns.

3.7.1 Analysis of Data Lake Architectures

Most of the presented data lake architectures share the common idea of dividing the overarching data lake into sub-systems, e.g., zones, layers, or classifications, i.e., maturity or functionality. This is done in order to structure the otherwise flat namespace. Data is either put into a sub-system based on the degree of processing it was subjected to, or by the specific functionality a sub-system offers, e.g., offering support for specific workloads.

The advantage of dividing a data lake into sub-systems is that it organizes the data and the interactions with it within the lake. However, this argument holds only true under the assumption of limited and well-defined data sources and workflows, since these sub-systems are purpose-built. Another disadvantage is the technical complexity of these systems. In the specific case of the zone architecture each zone is usually implemented by an individual set of (physical) systems, according to a meta-analysis [66]. This does not only increase the administrative complexity in general but particularly in managing access rights. In addition, these data lakes do not enforce an overarching data catalog by design, limiting the findability and the provenance tracking within the lake for the end users.

Another common feature across all presented architectures compared to the early Hadoop-based data lakes is, that modern data lakes emphasize a separation of storage and compute.

3.7.2 Analysis of Metadata Modeling

There are certain properties that most of the discussed metadata models share, as can be seen in Table 3.1. Almost all of the presented models have a high degree of generality, i.e., they can be used to represent any kind of data. The only exceptions are CODAL and CoreKG which focus on lexical data. Generally one can divide the metadata models into two groups, the entity-based systems, which model each datum, dataset, or data source as an individual entity, and the knowledge-graph, or information-centric systems. One can observe, that all general applicable metadata models are entity-based. All of the systems support continuously evolving metadata, while only two-thirds offer similarity links to support related dataset exploration. InteRESTingly, only one-third of all presented models offer data quality assurance, although the importance is generally recognized.

3.7.3 Analysis of Provenance Auditing

The evaluation of the discussed provenance auditing is done based on four different aspects, i.e., the employed auditing technique, the used communication protocol to

Model	Generality	Entity Based	Similarity Links	Quality Control
Data Vault	+	+	-	0
GEMMS	+	+	-	+
MEDAL	+	-	+	-
goldMEDAL	+	+	+	-
CODAL	-	-	+	-
Network-based Models	+	+	+	+
CoreKG	-	-	-	-
GOODS	+	-	+	0
Constance	+	-	+	+

TABLE 3.1: Comparing the presented metadata models. A + indicates full agreement, a 0 shows only partial agreement, and a - indicates that the corresponding feature is missing.

ingest the lineage information into the data lake, the format, in which it is stored, and the achieved reproducibility. These four aspects are key when an existing technique should be re-used for general-purpose use cases and pave the way for a more generalized system that can incorporate different compute paradigms, specifically for scientists. For provenance auditing a tool is required, which can track all interactions done with/on a compute cluster, e.g., an HPC system or a cloud. This specifically includes multi-step jobs on HPC systems which might include preprocessing on a frontend node, processing done non-interactively on the batch system, and possible postprocessing on the frontend nodes, once the batch job has finished. Similarly, a communication protocol needs to be used, which can be easily used from these different platforms. Here one needs to consider the different network accessibilities, e.g., compute nodes can only communicate within a dedicated HPC network, or cloud platforms might not enable a task to use a network at all. In addition, it is preferred to store provenance information in an established format to ensure interoperability.

In Table 3.2 a summary of the seven discussed data lake implementations is shown. One can see, that most systems do not automatically provide an auditing system and rely on the user or the user's tasks to pro-actively send the lineage information. All systems use a REST API for communication, although Komadu, which is based on RabbitMQ for provenance auditing, uses the Advanced Message Queuing Protocol by default but can be extended to use a REST API with an additional gateway. All systems use some kind of graph to store the provenance information. Usually, this is done via a Directed Acyclic Graph (DAG), however, only DCPAC follows the PROV-O convention, whereas all others implement a custom graph representation. One can also observe, that without an integrated auditing tool, reproducibility can not be achieved by design. Particularly JUNEAU provides a strong capability for reproducibility. That is because it introduces an extra auditing layer between the Jupyter Notebook, where users write their code, and the kernel, where the code is actually being executed. This enables JUNEAU to precisely record what transformations have been done on which input data and store this information along with resulting artifacts.

3.7.4 Analysis of Compute Integration

There are different approaches to providing tight integration of scalable compute resources into the fundamental design of a data lake.

System	Auditing	Protocol	Format	Reproducibility
CoreDB	-	REST API	Temporal Provenance Model	-
CoreKG	-	REST API	Temporal Provenance Model	-
GOODS	post-hoc, log-based	-	Knowledge Graph	-
Komadu	-	RabbitMQ	DAG	-
JUNEAU	Jupyter Notebooks	-	versioned flow graph	+
DCPAC	-	Apache Kafka	PROV-O	-
DataHub	provdb ingest, annotations, file views	-	property graph	+

TABLE 3.2: Comparing the presented lineage tracking systems. The auditing column contains the employed auditing tool, whereas a - indicates that no built-in system is provided. The protocol provides information about the communication protocol which transfers the records to the data lake, and the format states how the information is stored. The reproducibility column gives a qualitative assessment of the achieved default reproducibility.

The discussed JUNEAU system uses Jupyter Notebooks as its primary interface and therefore offers integrated compute capability per default. However, the general applicability is also limited since only tables, so-called data frames, are supported as input and output data.

KAYAK offers a very sophisticated mechanism to implement parallelizable and automatable workflows. However, the decomposition of an abstract workflow into *tasks* which can then be combined to *primitives* and finally can be chained to entire *pipelines*, requires quite some prior knowledge. Although powerful, the decomposition of established scientific software suits, as they are commonly used on HPC systems, into these entities, i.e., tasks, primitives, and pipelines, is not straightforward and would require a substantial amount of work for each job. A similar argument holds true for the usage in cloud environments. Furthermore, although the idea of a *tolerance* and a *time-to-action* is very useful on a data lake, this is only suitable for a subset of methods that are iterative by nature. From the viewpoint of a generic scientific application, this simply adds additional overhead and increases the entry barrier.

Klimatic on the other hand uses Docker containers to perform processing tasks. Although the generic approach to split up a pipeline into distinct stages which are linked by dedicated queues can be adopted to serve other use cases as well, it does not describe how the actual compute infrastructure is integrated. One can therefore assume that this solely works by accessing a local, or remote Docker engine.

Lastly, the Hadoop-based systems are discussed. Considering the capabilities of the Hadoop ecosystem, this achieves a high integration of compute into the data lake design. Using map-reduce methods, one can directly work on the data on HDFS format, but one can also use Spark for advanced analytics like machine learning. However, one is always bound to the Hadoop ecosystem and it is hard if not impossible to integrate other computing paradigms, like HPC or cloud, later if a project needs to extend its reach.

System	Generality	Performance Scalability	FAIR	Scientific Workflows	Data Lifecycle	Security	Modularity
Zone	0	0	-	-	-	0	+
Lambda	0	0	-	-	-	0	+
Lakehouse	-	0	-	-	0	0	0
Functional/Maturity	0	0	-	0	0	0	0
Data Vault	+	0	-	+	-	-	
GEMMS	+	0	-	+	-	-	
MEDAL	+	0	-	+	-	-	
goldMEDAL	+	0	-	+	-	-	
CODAL	-	0	-	-	-	-	
Network-based Models	+	0	-	+	-	-	
CoreKG	-	0	-	-	-	+	0
GOODS	+	0	-	0	-	-	0
Constance	+	0	-	+	-	-	0
CoreDB	-	0	-	-	-	-	-
Komadu	0	0	-	+	0	-	-
JUNEAU	-	-	-	-	0	-	-
DCPAC	+	+	-	0	-	-	-
DataHub	+	0	+	+	0	-	-
KAYAK	0	-	-	+	+	-	-
Klimatic	-	0	-	-	-	-	-
Hadoop-based	+	+	0	0	0	0	-

TABLE 3.3: Comparison of the discussed systems with respect to the in Section 2.2 presented criteria. A + indicates that the system completely fulfills the requirement by design, a - that the design contradicts the fulfillment of the objective, a 0 that it is dependent on the explicit implementation, and empty spaces indicate that no meaningful assertion is possible.

3.7.5 Gap Analysis

Table 3.3 compares the discussed systems to the identified requirements of the envisioned system, as discussed in Section 2.2. One can see that there is currently not a single system that is capable of fulfilling all the objectives. Although most of them are generally applicable, fewer are able to integrate scientific workflows. However, beyond these two characteristics, there is a large decline in the number of systems addressing the depicted challenges. There is only a single system that explicitly supports the FAIR principles, data lifecycle management, and security by design. Only a few others could be extended in specific circumstances, to support these criteria. Additionally, most systems are designed in a monolithic way, which makes the integration of new backend services difficult, which might be required to integrate new scientific workflows.

One remarkable gap is the lack of diverse compute integration into data lakes. The discussed systems in Section 3.6 are all tailormade to a single specific ecosystem, i.e., Jupyter Notebooks for JUNEAU, Docker containers for Klimatic, a purpose-built workflow engine for KAYAK, and the Hadoop ecosystem for all the Hadoop-based data lakes. These purpose-built systems do not provide a generalized compute interface and are solely limited to the individual compute framework, limiting the performance, scalability, and support for new scientific workflows which rely on different systems. This is a particularly outstanding observation compared to the other insufficiently covered requirements of the envisioned data lake, since providing scalable compute resources is generally recognized as a key characteristic of a data lake. There are a number of survey papers published, investigating the current state of the art [40, 48, 74, 136, 137, 167], where none of these survey papers treat computation at all, although [167] explicitly lists "scalability in terms of storage

and processing" as one of six key characteristics of a data lake. However, during their review process, they solely discuss the possibility of doing batch processing with Apache Hadoop and, preferably, Apache Spark, while pointing out that some implementations use Apache Flink or Apache Storm for interactive real-time processing, which is in good agreement with the findings of [48]. In [136] it is claimed that the provisioning of highly scalable computing capacity is one of the key challenges data lakes currently face. However, the presented systems in Section 3.6 do provide scalable compute capacity, however, each one does so in only one fixed and very specific way. Therefore, the actual challenge for data lakes is to integrate different compute infrastructures, like cloud and HPC systems, into one homogenous data lake.

Chapter 4

Integrating Compute and Data Management

Within this chapter, the challenges of data-intense projects on HPC systems are identified in Section 4.1, and a deeper background on storage tiering on HPC systems is provided in Section 4.2, from which an overarching concept for the integration of data management systems into HPC workflows is discussed in Section 4.3. From this analysis, a novel governance-centric interaction paradigm is proposed in Section 4.4. Parts of this chapter are published in [140].

In Chapter 2 the general idea of utilizing a data lake for the presented MRI use case is proposed to fulfill the identified requirements. This is further underpinned by an extensive literature review in Chapter 3. Within a gap analysis based on this review, a general lack of data lakes utilizing HPC systems for compute-intensive tasks is identified. This lack is caused by the general approach to integrate compute resources in a tailormade way to support only a single platform or ecosystem, instead of providing a general compute interface where different platforms can be plugged in. Although it seems generally possible to combine the different presented techniques to design a data lake, that can fulfill the requirements, for instance by using the very generic zone architecture and providing compute capabilities within different dedicated zones, this distributed approach does prevent an overarching data management and governance.

Therefore, this is an ideal point to perform Step 3, i.e., *Decomposition and Encapsulation*, of the presented scientific methodology, see Section 1.3, to identify independent components which can be encapsulated. Since there is no extensive knowledge base that could be established during the literature review, a very basic analysis is done with regard to the different ways users can interact with an HPC system, and what specific challenges are there for data-intensive projects on HPC systems. Based on this analysis, a hierarchical ordering of the identified, involved components is done.

4.1 Challenges of Data-Intense Projects on HPC Systems

When conducting data-driven research on HPC systems, four challenges can be identified:

- **Performance:** One can often see, that data-driven projects work with iterative procedures on small files, which lead to heavy loads on the storage system, particularly on the metadata servers, and can lead to large performance degradation due to storage bottlenecks.

- **Data Management:** Fulfilling the FAIR principles requires the inclusion of external services, like a handle system for PIDs, which are usually not hosted on an HPC system. Therefore, researchers tend to make their data FAIR at the end of a project, which is time-consuming and requires strict bookkeeping. For instance, to make data findable a naming scheme for created files, objects, and paths is mandatory. Then one also has to actually follow it. Sharing data with other researchers often comes as an afterthought. It is a reasonable assumption, that most projects do neither strictly follow the FAIR principles nor their Data Management Plan (DMP) if there was one defined at the beginning of a project. It can be expected that this issue will only be exacerbated by the increasing complexity and heterogeneity of the employed storage systems in the compute continuum.
- **Integration of Compute and Data Handling:** Computing on an HPC system feels a bit archaic. Users have to manually define many system settings, for instance, filenames and paths to define what storage to use. Meanwhile, the complexity of the tiered storage systems in modern HPC systems has drastically increased. There exists no way to define and enforce a data governance, which is homogeneously applicable across all of the disparate storage tiers since they all have their own set of policies. For instance, node-local storage is not backed up, and is only available during the resource allocation, but is very fast, whereas the globally available `HOME` filesystem is backed up but offers only very limited space and performance.
- **Reproducibility:** Being able to understand the lineage of data and how to reproduce certain outputs is important for trust in the scientific results. However, as executions on HPC systems are usually scripts that are invoked manually, on binaries created specifically for the given supercomputer, it is tricky to reproduce results.

For all of these challenges, but particularly for the second one, utilizing a suitable Data Management System (DMS), like a data lake, seems like a good solution. However, since there will always be a gap between a remote DMS and an HPC system, ensuring reliable information within the DMS that originates from an HPC system is an unsolved problem. This lack of reliable information is generally also true for provenance information. This might seem surprising since it is a general requirement for scientific work and there are provenance auditing tools available.

One often used approach is to monitor the system calls on the compute nodes during job execution, like *PASS* [134] does, to create audit trails. Following a similar approach, *LPS* [50] has drastically reduced the runtime overhead, but is not completely transparent due to the use of a dedicated *Library Wrapper*. *ReproZip* [42] also continues the idea of audit trails of system calls to automatically build packages to re-run an experiment.

A different way for lineage recording is provided by *Data Pallets* [113]. Here, all processes run within containers where all write access to the storage devices is intercepted and transparently redirected to data containers. Hereby, all data containers are automatically annotated with reliable provenance recordings.

Although there exist these different solutions, provenance auditing tools are not in widespread use on HPC systems. Considering the general idea of these tools, one simple explanation could be, that users who are already overwhelmed with their data are confronted with even more data, i.e., the recorded provenance. To circumvent this problem, resulting from a too-fine granular solution seems to abstract the

data management problem more coarse-grained to overcome the gap between these node-local and hardware-close tools and the higher level interaction a user wants to have with an HPC system.

4.2 Background on Storage Tiering in HPC

Usually, HPC clusters provide at least two parallel file systems, one `HOME` and one `WORK` filesystem, which provide file access via Portable Operating System Interface I/O (POSIX-IO) semantics, the relaxed Message Passing Interface I/O (MPI-IO) [47] semantics or the close-to-open semantics used by the Network File System (NFS) [155], to name just a few. All of these semantics require dedicated metadata servers to empower parallel file systems, like Lustre [174], or the General Parallel File System (GPFS) [171]. These metadata servers handle all metadata operations using special data structures called inodes to handle these metadata operations. If an inode represents a folder on such a filesystem, it contains a list of all inodes located in this folder. Depending on the actual operation, which should be done, it might be necessary to also read additional information from each inode within a folder, for example, the file permissions, or the ownership. The cost for these metadata operations scales linearly with the number of files stored within a single folder. However, if the list of inodes stored in a single inode becomes too long, indirect inodes have to be used. This behavior can be triggered if those inode lists are inlined within a small data block within the inode itself. This is typically done to avoid lookups on the storage servers holding the actual data of a file, which would otherwise increase the latency of such a metadata operation drastically. These indirect inodes, potentially even consisting of multiple layers, lead to an even worse performance degradation. Therefore, having too many files within a single folder has a huge performance penalty. However, this can often be observed in machine learning projects, e.g., if there a tens of thousands of small images in a single folder whose name encodes the particular target, like a folder called *cats* containing many small images of cats.

Although there is a varying amount of overhead necessary in the different semantics and filesystems, they all share the problem of bottlenecking when exposed to this described small file IO. Current mitigation strategies consist of providing a tiered storage system, where each tier is optimized to handle certain workloads, or meeting a specific cost-to-capacity ratio. This option leads to increased complexity and requires the users to manually move and stage data to the correct tiers to achieve optimal performance while ensuring that cold data is not piling up on fast and expensive storage, which is not backed up.

In addition, novel storage concepts, like object stores, are being integrated into HPC cluster[72], which supports flat namespaces by design. These are already common in cloud environments, with prominent standards like Amazon's S3, or Openstack Swift. However, their REST-based interfaces entail additional overhead, both, on the communication layer, and also on the application layer, since the file handling drastically differs from the well-established POSIX-IO compatible file systems. However, these designs require a lot of knowledge by the users and will increase the overall complexity of the data management in the sense, of what data is stored where and how can it be efficiently accessed.

In addition, the exact composition of these storage tiers is unique for each HPC system and usually evolves over time. This makes it nearly impossible to utilize data management systems, like the previously proposed data lake, to efficiently utilize an

HPC system, when users want to remotely execute an HPC job. That is because the developers, or admins of the data lake would need to continuously adapt the data placement for each function for the connected HPC systems.

Thus, the complex storage architectures of HPC systems are a burden for users working directly on HPC systems, and those only using it via a dedicated data management system. Based on this observation, the need for a generic solution becomes apparent, to further integrate storage and compute without relying on manual configurations by users.

4.3 Overarching Concept of Integrating Data Management Tools into HPC Workflows

There are several challenges that a user typically faces when working on an HPC system due to the complex storage structure. Considering the different namespaces offered by the different storage systems, it is challenging to get a concise overview of all the data that is currently stored. Here, the current strategy is to use systems to provide a unified namespace to users. However, these systems completely lack any awareness of the overarching workflow a user is doing and are therefore struggling to stage data meaningful and transparently record their lineage. That increases the barrier to adhering to the FAIR principles and performing transparent provenance auditing to ensure reproducibility. In addition, the DMS should optimize the usage of a tiered storage system to provide maximal performance during compute and use durable and low-cost storage for cold data. Here, a unified namespace is also not useful, because the data is already cataloged by the DMS.

First, the abstract components and their features are identified and discussed when interacting with a storage and compute infrastructure, such as an HPC system. Then the status quo as an archetype for the standard interaction paradigm and the envisioned user-friendly and data-centric flow are described.

4.3.1 Components

The necessary components when handling data and compute are as follows:

- **Resources:** These are raw storage, compute, and network infrastructures such as compute nodes and object/file systems and their interconnect. They come with their own specification, i.e., what resources they actually provide and their characteristics.
- **Resource Management (Compute):** This layer manages the usage of the resources by assigning compute jobs to available compute resources satisfying the requirements for the respective (parallel) jobs.
- **Resource Management (Storage):** This abstract concept defines where to store certain data and provides the respective space on a storage system.
- **Job Specification:** Defines the scope of a compute job together with its requirements, dependencies, and specification such that it can be executed.
- **Program:** A code that can be executed on the compute infrastructure, e.g., a binary program or script.

- **Software Landscape:** The ecosystem and environment provided by the platform that allows the preparation of programs on the system. This can also include software dependencies.
- **Workflow Specification:** Defines how to execute jobs and their dependencies to achieve the overall data-processing goal.
- **Data Management Plan:** Defines for any data inputs and data products the policies, data handling, and such to enable the FAIR principles while the data sovereignty of the user is preserved.
- **Data Management System:** A generic or domain-specific system organizing and indexing all data in well-defined schemata. Usually, these systems are used interactively and are hosted in a cloud environment. They may also provide process management capabilities.
- **User Interface:** Allows the user to interact with the system, e.g., to manage and interact with some or all of the above components and to upload/download data.
- **Client:** The computer system of the user, from which the user interface(s) are accessed.

The way, one can resolve the previously specified challenges and implement the components depends on the way a user wants to interact with it, the system providing these capabilities, and the data flow involved. For example, either, a user connects to the HPC system, and uses it as the central contact point, or the interface of the DMS is used to manage the data processing on the HPC system remotely.

4.3.2 Interaction Paradigms

On the most extreme scale, one can argue that there are three different kinds of users. For instance, tech-affine people who want to natively work on the HPC system in a traditional command-line approach, users who utilize state-of-the-art compute-centric tools, or those, who ideally only want to work with the interface of their domain-specific DMS. These three interaction paradigms are analyzed in the following.

Traditional Paradigm

In the traditional approach, the user interface is a shell (such as bash) on a login node of the cluster and the client is a Secure Shell (SSH) [212] enabled program that the user runs on their Desktop/Laptop. There exists no data management plan, the user thinks about how to manage data and, therewith, manually performs the resource management for storage, identifies how to map output data to files (influenced by the applications) and directory structures, and utilizes the available parallel file systems. Also, the user manually prepares programs s/he wants to use by downloading the necessary codes on the machine and ensuring it works with the system architecture and software environment that is deployed on the HPC system. The wider software landscape on the HPC system is prepared by data center staff but libraries can be extended by the users in order to create meaningful programs. In most cases, workflows are not explicitly specified but manually invoked. The resource management of the compute resources is provided by tools such as Slurm. The job specifications are (bash) scripts that are invoked - they define the compute requirements.

Such scripts are submitted to Slurm which decides how to map and schedule them on the available compute resources. These steps are basically manually set up, requiring a scientist to think about how the experiment should be conducted and then documented (if at all) in a lab notebook or scripts that do some of the work. Potentially, workflow tools such as Snakemake are utilized to specify and automate dependencies between tasks. This is not only error-prone but any change to the environment requires the user to modify the experimental setup and perform the steps again. This very common interaction with the HPC system can be considered a bit archaic.

Compute-Centric Paradigm

In a Compute-centric approach, a user would connect to the HPC frontend as usual. In the simplest form, a user would delegate the job of maintaining a data catalog and staging the selected input data to a DMS tool. This workflow is depicted in Figure 4.1. Here, to get access to the requested input data, a user would formulate a domain-specific, semantic search query and send this request to the DMS. Usually, a DMS would use a dedicated database or search engine to filter the requested data. The data is loaded into the running code of the user. This could either require a dedicated data transfer to a pre-configured storage target, or the HPC system and the DMS are already connected to the same storage system. When looking at different established systems, like iRODS[1] or Delta Lake [4], the user is typically responsible for lineage recording and enforcing reproducibility. Therefore, these solutions typically only assist users to manage and organize their data but do not free them from the burden of efficient IO and working in agreement with good scientific practice.

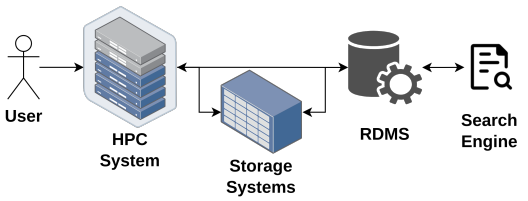


FIGURE 4.1: Sketch of the compute-centric paradigm.

Use-Case-Centric Paradigm

The opposite way to integrate a DMS into an HPC workflow, is to use the DMS as the user frontend, see Figure 4.2. This DMS-provided user interface can be used to query and select input data, define a compute job, and submit this job to an HPC system, without the need to extra log in to the HPC system or transfer data explicitly. One example of such a DMS is *Viking* [100], which offers a user interface on a web application from which users can trigger the execution of different molecular dynamics simulations without logging into the HPC system via SSH. This functionality requires a communication channel, between a remote DMS and an HPC system. Additionally, the DMS needs to be able to work with the individual resource manager, of each HPC system. The advantage of this approach lies in the capability to perform transparent lineage recording and guaranteed reproducibility. That is, because the DMS has complete control over the input data and

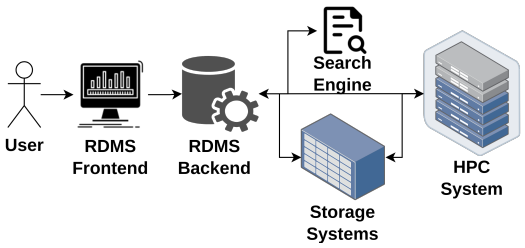


FIGURE 4.2: Sketch of the DMS-centric paradigm.

[1] https://irods.org/

the processes that run on them, using thorough logging methods is enough to ensure reproducibility. This is similar to the idea of JUNEAU presented in Section 3.5.4. Similar to the HPC-centric use case, storage tiering is hard to support.

4.3.3 Data Flow

These two scenarios also differ in their data path, i.e., in the storage systems involved in the data management and data processing, and the data transfers between these systems.

Compute-Centric Paradigm

In the Compute-centric use case, a user accesses data through their respective, native interface, e.g. through the library functions of their respective programming language, or as an input parameter of their program that they want to run. This means, that only data is available which is directly accessible from the HPC system, and data transfers, e.g., for better performance, have to be done manually. In Figure 4.3 a layered diagram is shown which shows the possible data staging strategies. Within the user interface, e.g., ssh or a jupyter notebook, a user can explicitly copy/stage data on non-node-local storage. Within this layer, this has to be done deliberately. Assuming that a workflow engine, like snakemake, is used, data can be staged on non-node-local storage in a more automated and transparent way in the form of a dedicated workflow step. These two options can be considered asynchronous data staging since this will not lead to stalling times on the compute infrastructure. Synchronous data staging happens on the *Job Layer*, where a process first has to access data on a slow storage tier and then stage it. In this case, even a very fast node-local storage tier can be used. Therefore, the *Workflow Engine Layer* is on top of the resource management layer, to enable asynchronous data staging. Afterward, the actual job can continue to process the data. Since the node-local storage is typically only available during the resource reservation, which is managed by the resource manager, e.g., Slurm, a user has to ensure that the overall process running on that node, does not only stage data, but also archives it once it is done. The entire data staging and IO optimization is therefore solely the user's responsibility.

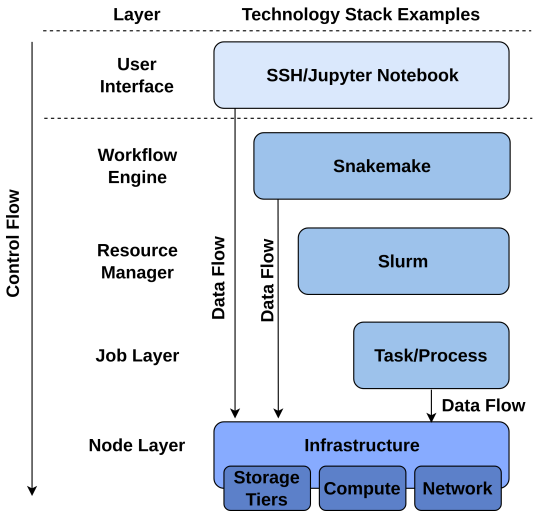


FIGURE 4.3: Layer diagram showing the control and data flow of the compute-centric paradigm.

Use-Case-Centric Paradigm

Within the DMS-centric approach, users are generally not interested in accessing the data directly, e.g., with a suitable library within self-written code. Instead, they are rather interested in having the complexity of running their job abstracted away. For this strategy, a layered diagram is shown Figure 4.4, depicting the data flow. Here, a user accesses the DMS via a browser, or a DMS-specific Graphical User Interface

(GUI). Within this interface, a user triggers the execution of a workflow, or a single analysis step on the selected input data. Often, these DMS are deployed in a cloud environment and have their own storage layer. Depending on whether this storage tier can be integrated into the HPC system, there are different strategies for data access.

Either one can asynchronously or synchronously fetch data from the DMS within a dedicated data mover process and stage it either node-local, in the case of a synchronous data transfer, or non-node-local in the case of an asynchronous data transfer. For this purpose, the data mover process would either be granted access to the DMS storage concerning the user's permissions, or the DMS can provide an endpoint, for instance, a REST endpoint, from which the process can fetch the required data. Since generally there is a communication layer required to access the resources of an HPC system from the outside, this can also be used to asynchronously fetch data from the DMS. Lastly, a dedicated data mover process can also fetch the data synchronously from the DMS on the compute node itself. This means, that the entire data movement and staging strategy is solely in the hands of the admins of the DMS, where the corresponding functionality is implemented and configured.

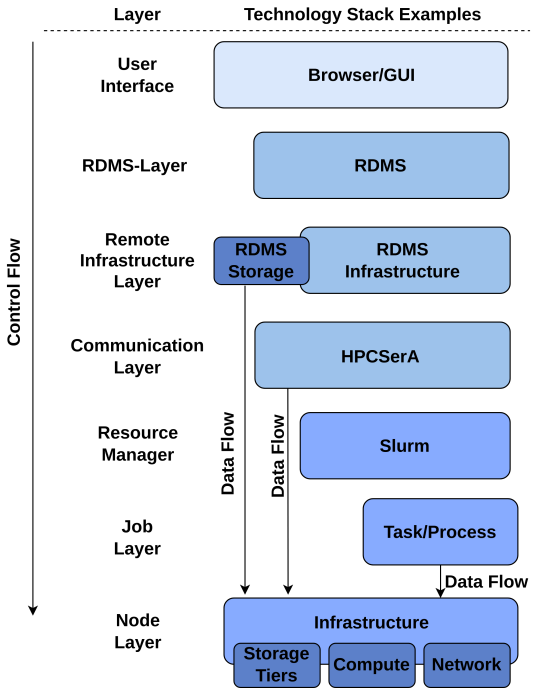


FIGURE 4.4: Layer diagram showing the control and data flow of the DMS-centric paradigm.

4.3.4 Control Flow

Similar to the aforementioned data flow, there also exists a control flow, as can be seen by the left arrow in Figure 4.3 and Figure 4.4. The control flow is initially triggered by the user within the user interface and is from there passed down to the final task running on a node. From this upmost layer, the control path goes down via an optional workflow layer to the resource manager where the tasks get mapped on the actual hardware, in case of the HPC-centric view. In the DMS-centric view, the control flow gets even more abstracted, since the user input recorded by the user interface has to first pass through the DMS layer, where the user request gets initially processed and mapped on the DMS infrastructure. Since this DMS system is completely disjunct from the HPC system, a dedicated communication layer is required to bridge those two systems. On the HPC system, the individual tasks are again mapped to the nodes in the infrastructure layer via the resource manager.

4.3.5 Analysis of the Interaction Paradigms

To summarize the previous discussion about the different user interaction paradigms, Table 4.1 compares the characteristics of the individual components – ignoring the

Characteristics	Traditional	Compute-Centric	**Governance-Centric**	Use Case-Centric
Resources (Compute)	Auto	Auto	Auto	Auto
Resources (Storage)	Manual	Manual	Auto	Auto
Res. Mgmt (Compute)	Semi-Auto	Semi-Auto	Auto	Auto
Res. Mgmt (Storage)	Manual	Manual	Auto	Auto
Job spec	Manual	Semi-Auto	Semi-Auto	Auto
Program	Manual	Manual	Semi-Auto	Auto
Software land	Provided	Provided/User-Container	Provided/User-Container	Provided
Workflow spec	Manual	Semi-Auto	Semi-Auto	Auto
DMP	Manual	Manual	Semi-Auto	Tool-specific
User interface	SSH	SSH+Web	Web+SSH	Web
User interface (Data)	SSH	SSH	Web+SSH	Web
Client	SSH	SSH+Browser	Browser+SSH	Browser
Performance	User-specific	User-specific	++	+
Data management	-	-	++	Tool-specific
Integration	–	0	++	++
Reproducibility	–	+	++	Tool-specific
Flexibility	++	++	+	–

TABLE 4.1: Comparison of different HPC user interaction paradigms.

column Governance-Centric for now. The responsibility for one of the defined components and features are either the user, i.e., a manual process, semi-automatic, thus aiding the user (potentially following a specification), or fully automated. User-specific means it depends on the skill of the user. The resulting differences in the degree of automation can be illustrated best when looking at the interfaces a user can use to interact with the processes and data. Traditionally, only ssh connections are supported, whereas in the compute-centric paradigm, at least some interactions may take place via a web interface. In the use case-specific paradigm typically only a web interface is available that hides the HPC system and all internal processes.

The resource handling requires a lot of manual interaction of the users in the traditional and the compute-centric concept, while it is completely automated in the use case-centric approach based on configurations provided by the admins of the specific DMS, not by the admins of the HPC system.

A similar pattern can be seen in the characteristics of the task-related components. Here, the user experience with the HPC system evolves from a completely manual interaction to a partially automated or guided system in the compute-centric context, where already some low-level programming tools for workflow orchestration, data selection, and containers for dependency management are used. However, the program and data management rely still on manual work and are therefore potentially error-prone. On the other side, the use case-centric system fully automates these steps. Again, all task-related interactions are fully automated by the use case-specific system.

These intrinsic characteristics have different advantages and disadvantages. The traditional HPC usage paradigm relies heavily on manual work by the user to achieve a reasonable performance. Also, the data management, the integration of storage and compute, and therefore the overall reproducibility is very much exposed to user errors. However, this enables the highest level of flexibility. The compute-centric paradigm improves this by utilizing the discussed semi-automated components and hereby improves the integration and reproducibility. The use case-specific systems will most likely have reasonable, but not custom-made, configurations to achieve good performance. Here the interaction with data is challenging as the upload/-download via web frontend limits the performance.

4.3.6 Discussion of Isolated Components

It is shown that in both cases, i.e., the compute-centric and the use-case-centric paradigm, some form of central contact point, i.e., frontend is required, be it a graphical interface of a DMS as in the case of the use-case-centric paradigm, or a system that provides an API that can be used in user programs or with a command line interface. Based on the layer diagram of the two paradigms showing the involved components for the control and data flow, see Figure 4.3 and Figure 4.4, different independent layers, or components are identified. One important component in the use-case-centric paradigm that is identified is the communication layer. This layer needs to provide an interface to the DMS as well as to the HPC system. However, comparing the paradigms, as done in Table 4.1 demonstrates a significant current challenge. Since one of the requirements is to be as generic as possible, it seems reasonable to support both paradigms. Furthermore, when unifying the concepts of the compute-centric and the use case-centric paradigms, the individual advantages of these two worlds should be combined to offer the best possible solution.

4.4 Novel Governance-Centric Architecture

The goal is to expand upon the existing concepts to provide a novel, unified view of processes and data in order to improve the user experience on HPC systems. To this end, a novel *Governance-Centric* interaction paradigm is proposed based on the previous analysis of the current interaction paradigms where different gaps are identified. This unified view can then serve as the foundation to not only integrate HPC systems in a generic data lake architecture but might serve as a blueprint and a deeper understanding of how to integrate diverse computing ecosystems into a data lake, and how to support different interaction paradigms with it. Similarly, this also allows to increase the efficiency of data-intensive projects running on HPC systems that are completely unrelated to the data lake in question. The metrics for the user experience, i.e., performance, data management, integration, reproducibility, and flexibility, all mostly boil down to the question of where the data is located and how they are linked. This has to be tackled simultaneously in two directions: first, an additional integration layer above the resource manager (compare Figure 4.3 and Figure 4.4) is required, as shown in Figure 4.5. This layer has to provide an integrated and unified namespace to the users. Secondly, an information flow, which is directed in the opposite direction as the control flow, is required. Although this can be achieved with available auditing

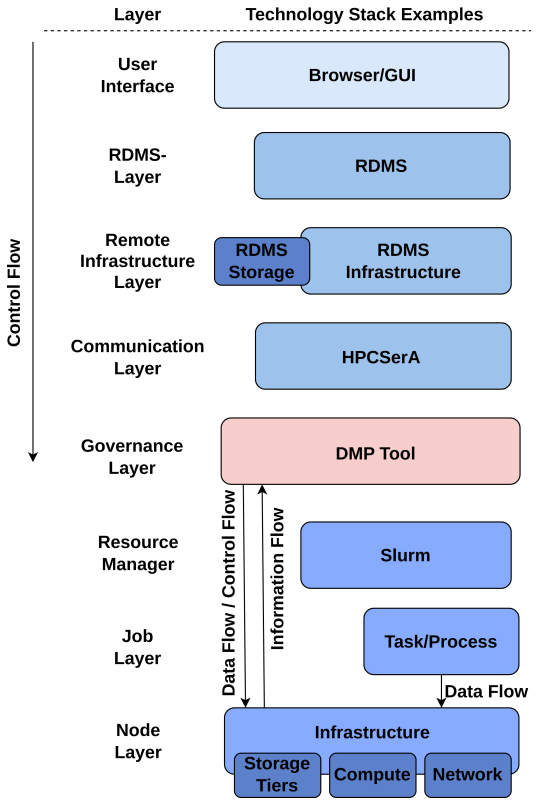


FIGURE 4.5: Sketch of the governance-centric paradigm. Both user groups (compute-centric and DMS-centric) are orchestrating the data flow through the additional *Governance Layer*.

tools, there is no concept for a tool that processes the incoming information and hereby makes it actionable. The advantage of an actionable information flow compared to the existing systems is that the information is utilized to create a desired, predefined state, and not just create yet another piece of data a user has to manage manually.

Therefore, the governance-centric interaction paradigm is proposed that aids the users and automatizes the integration of data and compute. In Table 4.1, the required degree of automation to bridge the gap between the compute-centric and the use-case-centric paradigms is identified. Resource management should be fully automated to achieve the highest degree of integration of data and compute. The task-specific components should be semi-automated to guide the user in managing the data, working reproducibly, and ensuring that a predefined, ideal state is reached while allowing as much flexibility as possible. Similarly flexible should be the interface, to allow interaction from both user groups.

A professional and proper data management requires users to define an *experimental description* at the beginning of a project. This experimental description should consist of a workflow that links data sets and compute tasks as well as a data management plan for the respective input/output data. Then the user has to initially modify their tasks, e.g., job scripts, to allow linking of the tasks and their data products to the workflow and also to generate descriptive metadata for the data sets. Building upon the previous discussion, the goal is to not only use the workflow as an abstract concept that users may informally follow but rather enforce its usage. The implications of the design are that the HPC system can exploit the information to perform many, of the previously manual tasks, automatically. Examples of these tasks that should be automated are, receiving and processing information about input data and artifacts created during the task execution, enforcing archival/deletion policies defined in the DMP, or ingesting results into a DMS along with all required metadata including lineage information. The latter is the required feature that will unify both user groups, i.e., those working in a use-case-centric DMS paradigm and those working in a compute-centric paradigm.

To summarize, the experimental description shall be a user-defined and machine-readable workflow description that contains information about the data flow, the tasks that process these data sets and create artifacts, and further optional information like access policies or the IO intensity of each task. This means that for every task a user wants to schedule via the resource manager, this task has to be linked to a specific workflow step within the experimental description at job submission time. Thus, each and every submission of a job on an HPC system becomes one concrete invocation of the abstract task description within the experimental workflow linked to data in the DMP.

4.4.1 Experimental Description

Specifically in data-driven projects, it is common that there is not a single task, but that the entire processing consists of multiple steps which are concatenated into a workflow. Therefore, a user has to provide a simple graph, compare Figure 4.6, connecting input and output data via tasks as a workflow description. This workflow could represent an MRI pipeline, where Dataset 1 could be a k-space recording and Dataset 2 is the model that is used within Task 1 to reconstruct Dataset 1 into image space. The resulting image can then be manually inspected by a doctor. The reconstruction can also be used for further processing, for instance for a quantized, volumetric analysis of the different brain areas. This process can be repeated with

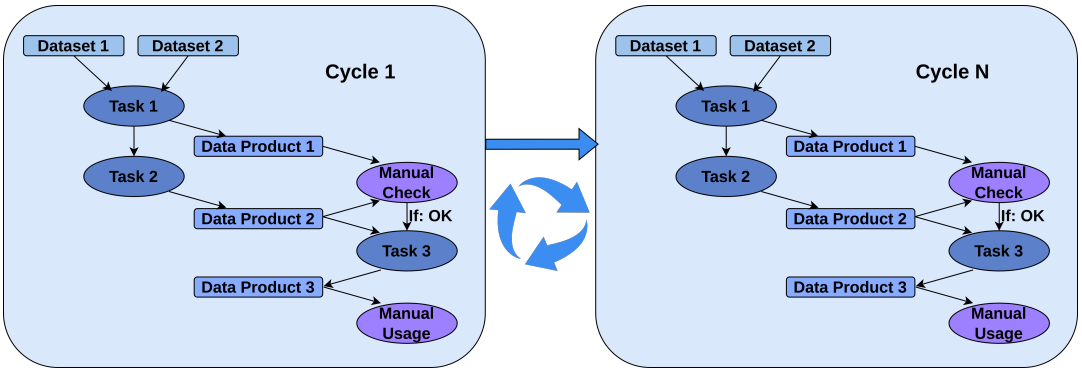


FIGURE 4.6: High-level view of an experimental description represented by a workflow.

different parameters to find the optimal configuration. Manual steps in the workflow are explicitly annotated as they require data to be accessible.

Within this workflow definition, general policies can be defined, e.g. where and when data should be archived, how long artifacts should be kept on hot storage if a manual inspection is required, what accompanying metadata are required, or if input data can be altered. The archiving of data should explicitly support remote DMS as a target, to integrate HPC systems with remote DMS, and similarly, data within a remote DMS should also serve as possible input data. The required data mover tools have therefore to be integrated as dependencies into the DMP. In addition, the users can provide information about the expected IO profile, aiding the proposed DMP tool to find the best storage tier based on heuristics configured by the HPC admins. Based on further metrics, like the available bandwidth of a remote DMS and the HPC system, or the amount of data, the DMP tool can also determine, whether data should be staged synchronously, i.e., during compute time, or asynchronously, i.e., as an additional, dedicated, and dependent step before the compute task starts.

Describing Datasets: The user can and should add further information to the data sets that are getting processed or created. To improve the findability a user should provide domain-specific metadata, or define a task to extract those. The required, and optional domain-specific metadata fields can be defined in the DMP. This can even ensure a homogeneous metadata quality across a larger user group working collaboratively on a joint project. In addition, the data life cycle should be defined, i.e., what is the retention time, or what are the deletion policies. To meet the data governance policies required by the user, additional aspects such as access control must be defined to prevent unwanted data leakage.

4.4.2 Modifying Tasks

Compute jobs on HPC systems are dispatched to the actual resources using resource managers such as Slurm. On this level, a job has to be prepared, annotated, and linked to the step within the workflow description. This can be done with a simple argument provided during job submission. In addition, a user can further restrict and specify the input data. Here, the largest change compared to the traditional HPC interaction paradigm becomes apparent: Instead of working with explicit files, a user rather works with datasets defined by metadata. For instance, a user specifies the input data either based on domain-specific metadata or simply due to the link in the DMP. Therefore, the actual storage location is abstracted from the user. The actual data directory, which a program still needs to specify in API calls, can be

automatically exported via environment variables or generated via support tools in the job script. Before reserving dedicated compute resources, the proposed DMP tool decides to use either synchronous or asynchronous data staging and stages the data respectively.

One key requirement in science is reproducibility. In a first step, this requires at least sufficient provenance information to allow for retrospective comprehensibility of the lineage of the resulting artifacts. One important element to retrospectively comprehend an HPC job is the run script used for batch processing. This can be automatically archived along with the artifacts by the proposed DMP tool. However, these batch scripts, which contain the actual compute job to be run in the form of a shell script, can have multiple ambiguities. One simple example of this would be the execution of an interpreted script, e.g. a *Python* script. Here, one would have a simple line within the batch script which would look similar to: `$ python my_script.py`
The challenge within this call is to track differences between multiple invocations of this script, where the content of *my_script.py* has been changed. For this, three different high-level modi are proposed.

The recommended way is to use a Git repository, where changes in code are properly tracked. In this case the DMP tool checks in the directory of the script for the existence of a git repository and saves the information about the used git commit hash. Then, this information can be stored in the metadata of the created data products. The DMP tool will create a dedicated sidecar file for this metadata in the output directory. The usage of version control systems can, and should, also be part of the required specifications within the DMP.

Alternatively, if no Git repository is set up, the batch script is parsed and untracked dependencies in the user namespace, like a Python script, are tried to be identified and archived along with the artifacts. Since this method is potentially more error-prone compared to a proper version control system, like Git, it is not recommended, but should still offer a better chance for retrospective comprehensibility when compared to other strategies. These dependencies will be listed in the before-mentioned sidecar file, and are archived alongside it. One important distinction to make is the use of containers. In this case, the container image should be archived and linked to the sidecar file. Of course, utilizing provenance-specific tools and translating them to the required standard in the sidecar file, is also an option to explore.

The third option is that users compose the sidecar file by adding code to the batch script, where this provenance information are provided. This can be added to the directory where the output, which should be archived, resides. This file will also override information that was automatically tried to extract in the previous step.

4.4.3 Implications of the Design

This presented design has different positive implications on the user experience that are discussed in the following, based on the characteristics stated in Table 4.1.

Integration First of all, the abstraction of files on storage towards more high-level data sets achieves the tight integration of storage and compute. Abstracting the storage from the data will motivate users to use proper metadata management systems and establish a data catalog, instead of encoding information into file paths.

Performance Since users are only working with datasets and not with a filepath anymore, HPC admins can configure data placement strategies, therefore relieving this burden from the users and optimizing the performance.

Reproducibility Since compute tasks, their input, and resulting data are strongly linked with each other, the lineage of artifacts is much more comprehensible and less subjected to user errors. Utilizing further tools like containers and a version control system will ensure full reproducibility, which is integrated into this paradigm by design because this is just another policy in the defined data governance, which will be enforced for the users.

Enforcing DMP Although the general idea of using data management plans in HPC is far from new, the novel advantage of this particular tool and its design is that it can be enforced. That is because as a response to the control flow provided and triggered by the user, an information flow is received about the changes in the overarching state, which is then acted upon. There are various ways to achieve this goal. A naive approach compatible with existing systems is to use a cronjob that reads in the workflow and task definition files, which a user has provided, and compares the specified, desired state of the storage systems of the user against the actual state at hand. If new output data are detected and the required sidecar file for the necessary metadata is available, the output data is handled as specified. However, if the required sidecar file is not, or only with insufficient content provided the user will be reminded to provide the missing information after a specified grace time. Similarly, if data is detected that can not be matched to the dataset specification in the workflow definition, an error or warning can be raised to the user as such unclassified data shall not exist. Thus, the DMP becomes actionable and hereby ensures a homogeneous system state in sync with the experimental description and user expectations.

Predefined and Encapsulated Jobs Within the experimental description shown in Figure 4.6, tasks are fully defined in advance. This includes the operations that are done on the input data, as well as the software environment in which the processing is done. The execution of these predefined, non-interactive jobs is similar to the invocation of a predefined function, as it is common within cloud infrastructure providing a Function-as-a-Service (FaaS) interface. Thus, the FaaS idiom is an ideal interface to provide homogenous access to heterogeneous and ecosystem-agnostic compute capacity.

Chapter 5

Remote Access to an HPC System

In this chapter, the required communication layer between the DMS layer and the HPC system is designed. For this, the related work is analyzed and gaps are identified in Section 5.1. Then the different design options are discussed and the most suitable is chosen in Section 5.2. Based on this, a new service called HPCSerA is developed, where first the general architecture is presented in Section 5.3, the Function-as-a-Service capabilities are introduced in Section 5.4, and then the security architecture is explained in Section 5.5. Details of the implementation are provided in Section 5.6. Lastly, the general applicability of HPCSerA is demonstrated in Section 5.7. Parts of this chapter are published in [21, 22, 106, 194].

As identified within the *Decomposition and Encapsulation* step in Chapter 4 a communication layer between a DMS and a compute cluster is required. Specifically for compute-intensive tasks, an HPC system should be utilized. Considering the described mechanism in the DMS-centric paradigm Section 4.3.2, that a user should be able to process data via a DMS frontend, this application needs to connect to the HPC system on behalf of the user. Therefore, one needs to establish a communication interface between the DMS frontend and the HPC system, which ideally happens within the user space of the requesting user. Usually, a user would connect to an HPC system via SSH and is then directly working in a terminal, i.e., the traditional interaction paradigm.

An alternative execution model popular with cloud systems is FaaS. As proposed within the governance-centric interaction paradigm, doing large-scale, data-intensive research on HPC systems requires a DMP that consists of predefined functions. Therefore, the FaaS interface is ideally suited to provide homogenous access to heterogeneous and ecosystem-agnostic compute capacity. For this, the communication layer introduced in Section 4.3.2 needs to expose a FaaS interface. Within the FaaS computing model, a platform for the execution of functions is provided, i.e., code can be submitted and configured by the user. Afterward, the user can trigger the execution of the function with parameters via an exposed endpoint, where communication is typically done via HTTP [59]. A runtime system executes the function in an isolated environment, usually a container, and automatically scales up the number of running containers according to the number of incoming requests. This interface and its interaction with it become increasingly popular due to several advantages, including cost-effectiveness, fault tolerance, and ease of use. Providing such an interface for HPC systems would enable an HPC-native drop-in replacement. This enables users to choose for their individual use cases and tasks if they would benefit more from a cloud-based system like *OpenFaaS* [149], or if the full HPC capability and environment are required. Therefore, one has a homogeneous interface that can connect to different compute resources, depending on the compute

intensity. This is one of the main goals stated in Section 2.2 and a main gap in the current state-of-the-art of data lakes as identified in Section 3.7.5.

While, on one hand, there are efforts to ease and open up the use of HPC systems, like providing graphical user interfaces to manage complex and compute-intensive workflows on HPC systems [214], there is, on the other hand, a constant threat by hackers or intruders. Since users typically interact with the host operating system of an HPC system directly, local vulnerabilities can be immediately exploited. Two of the most favored attacks by outsiders are brute-force attacks against a password system [182] and probe-based login attacks [108]. These attacks, of course, become obsolete if attackers can find easier access to user credentials. Therefore, it is of utmost importance to keep access, and access credentials, to HPC systems safe.

In this context, services easing the use of and access to HPC systems should be treated with caution. For example, if access via SSH to an HPC system is only possible using *SSH keys* due to security concerns, these measures are rendered ineffective if users re-establish a password-based authentication mechanism by deploying a RESTful service on the HPC system that is exposed on the internet. Observing these developments, it becomes obvious that there is a requirement to offer a RESTful service to manage data and processes on HPC systems remotely which is comfortable enough in its usage to discourage spontaneously concocted and insecure solutions built by inexperienced users with the main objective of "getting it to work", but which adheres to the highest security standards. In order to prevent those security risks by users, HPC systems are increasingly secured, including Two-Factor Authentication (2FA) for SSH connections [30], which is a problem for automated workflows in general and the offloading of compute-intensive tasks from the data lake in particular, since they need to run without any manual interaction.

To summarize, a number of objectives for the communication layer between the data lake frontend and the HPC system can be defined:

- Communication should be REST-based and follow the FaaS-idiom
- It has to scale to arbitrary amounts of users and function calls
- Communication has to be secure and must not circumvent a second factor

5.1 Related Work

There is without question a general trend towards remote access for HPC systems, for instance in order to use web portals instead of terminals [31]. These applications actually have a long-standing history with the first example of a web page remotely accessing an HPC system via a graphical user interface dating back to 1998 [125].

Newer approaches are the *NEWT* platform [43], which offers a RESTful API in front of an HPC system and is designed to be extensible: It uses a pluggable authentication model, where different mechanisms like Open-Authorization (OAuth), Lightweight Directory Access Protocol (LDAP), or *Shibboleth* can be used. After authentication via the */auth* endpoint, a user gets a cookie which is then used for further access. With this mechanism *NEWT* forwards the security responsibility to external services and does not guarantee a secure deployment on its own. This has the disadvantage that *NEWT* is not intrinsically safe, therefore providers of an HPC system need to trust the provider of a *NEWT* service that it is configured in a secure manner. Additionally, no security taxonomy is provided, which is key when balancing security concerns and usability.

Similarly, *FirecREST* [49] aims to provide a REST API interface for HPC systems. Here, the Identity and Access Management is outsourced as well, in this case to *Keycloak*, which offers different security measures. In order to grant access to the actual HPC resources after successful authentication and authorization, an *SSH certificate* is created and stored at a *FirecREST* microservice. Although this is a sophisticated mechanism, there seem to be a few drawbacks. First of all, the *sshd* server must be accordingly configured to support this workflow. Secondly, it remains unclear how reliable status updates about the jobs can be continuously queried when using short-lived certificates. Lastly, these certificates need to be stored at a remote location, which might conflict with the terms of service of the data center of the user. A similar approach is used by *HEAppE* [193] where the communication is between the API server and the HPC system is done via SSH. To do so, for each project an SSH key is managed by the API server. Users are not supposed to connect to the system via SSH at all. However, in order to upload data via Secure Copy (SCP) users obtain a temporary SSH key. To manage the exposure of a possible data breach of the API server, the developers recommend using one instance of HEAppE per HPC account.

Additionally, HPC systems are often configured to allow logins from a trusted network only, which means that the *FirecREST* microservice can not serve multiple HPC systems at a time.

While the *Slurm Workload Manager* provides a REST interface that exposes the cluster state and in particular allows the submission of batch jobs, the responsible daemon is explicitly designed to not be internet-facing [169] and instead is intended for integration with a trusted client. Clients that shall execute Slurm jobs authenticate the trusted Slurm controller via the *MUNGE* service [58] that relies on a shared secret between client and server. Slurm can be deployed across multiple systems and administrative sites and there are various options for Slurm to support a meta-scheduling scenario or federation. However, if the Slurm controller is compromised, it can dispatch arbitrary jobs to any of the connected compute systems. In addition, decoupling the API implementation from the choice of the job scheduler, as we propose, allows interoperation of multiple sites, possibly using different schedulers.

In conclusion, there is currently no existing service that can fulfill all the requirements, i.e., a scalable and secure, REST-based FaaS interface.

5.2 Analysis of Possible Designs

Before starting to design a new system, a comprehensive comparison between the different options is made in the following. All of the discussed methods can be sorted into two different classes, those that push from a remote system to the HPC system, and those that pull. Pulling requires that there is a process already running on the HPC system, which can perform this task.

5.2.1 Pushing Methods

All discussed pushing methods will be based on SSH, however, with different workflows and server-side configurations. Therefore, in all of the following methods, there is an external system, like an API server, or the discussed data lake frontend, which tries to build up an SSH connection on behalf of the user.

Password-based Authentication

The simplest form of building up an SSH connection is by using passwords. In this scenario, the user would provide the password to the external system, which then has full access to the user space of the HPC system. Since password authentication is the most insecure, some, or even most HPC systems do not support this mechanism.

Key-based Authentication

In this scenario, a user would provide his/her private SSH-key to the external service. Then, this service can connect to the HPC system and acquire a shell within the user space of the user, similar to the before-mentioned password-based authentication. This mechanism can not support newer key authentication schemes like the FIDO2-based security keys. In addition, if the external system gets hacked, the attackers can use these private keys to connect to the HPC system and likely compromise this as well.

Certificate-based Authentication

SSH certificates seem like the ideal solution for this problem. When the user makes the compute request on the external system, a new ephemeral SSH certificate needs to be created for this external system. This can be done using an OAuth flow, that connects to the identity provider which the HPC system uses. Here, a user can acknowledge the request the external system made on his/her behalf. This step can also be combined with a second factor. The admins of the HPC systems can freely configure the expiration date of the issued certificates. By only using short-lived certificates, the attack surface is drastically reduced. In addition, certificates do not need to be written to disk, which further decreases the risk of theft. However, this method requires special infrastructure that needs to be provided for each HPC system. Also, this still gives the external system full access to the user's HPC space, which requires a large trust by the users in this external system.

Force Commands

To tackle the problem that a user has to provide full access to his/her user space so-called force commands can be used. These force commands restrict the usage of an SSH connection to a fixed executable. They can be configured for both, SSH certificates and SSH keys. Although until now, only the trust a user has to put into a remote system is mentioned, there is also another dimension. HPC service providers might require users to keep their SSH access guarded and basically forbid users to grant full access to a third party.

Therefore, when using SSH keys with force commands, it is not enough to simply put the force command within the `authorized_keys` file within the user space to restrict the access of a key. Instead, this needs to be supplemented by a corresponding `Match` statement within the `sshd_config` file. This requires an admin to add the corresponding clause.

When using SSH certificates, each certificate can be tied to a specific force command. This has to happen when the certificate is created using `ssh-keygen`. Certificate creation requires the private key of the *Certificate Authority* (CA). Therefore an extra service would be required, that passes on the force command, an external system wants to execute, to the server holding the CA. Ideally, this is combined with a second-factor authentication where a user can be even prompted with the command

that should be executed. Although this would solve the problem, this requires a lot of infrastructure and would consume large amounts of development and administration time.

5.2.2 Pulling Methods

A different approach is to pull tasks into the user space, instead of pushing into it. The advantage of this idea is, that this can be developed and setup completely in user space, therefore not relying on any processing by the admins. Therefore, from the HPC systems perspective, no additional setup is necessary, since users are only doing, what they are typically allowed to. In both discussed implementations, a process is running on the HPC system in user space, e.g., as a cron job. This task then pulls the required information and data from an external source.

S3-based Communication

The first approach is to use S3 as a middle layer. Within a bucket, each unique HPC job gets a dedicated prefix, e.g., a jobid. To this prefix, all the corresponding input data can be uploaded. In order to enable the necessary control flow, a job specification file has to be provided. That file is a simple JSON file, containing all information required to execute a specific job, for instance, which job to execute, what arguments to pass to it, environment variables that should be exported, a callback URL, or where the input and output data should be stored. Additionally, a status file has to be present for each job, i.e., has to be present within each prefix.

When a new job should be submitted to the HPC system, all these previously described files are created and uploaded, and then the status of the job within the status file is set to `new`. The agent periodically crawls through all available prefixes within the bucket and checks all the status files. Once this agent detects a new function prefix with a state `new` it will read the job specification file, stage the input data accordingly, and submit the job with all optional arguments and environment variables to the batch system. Then, the state of the function is updated to `running`, and the corresponding Slurm job-id is stored. The job status of all `running` jobs is continuously tracked by the agent. After a job is finished, the output data is uploaded back into the S3 bucket into the same prefix, the state is updated to `finished` and an optionally provided callback URL, within the job specification file, is triggered to notify the external system that the job has been finished and that the output data can be fetched from the S3 bucket. This mechanism assumes that the user has access to the S3 bucket and that all functions, that shall be executed, are already available in the user's `HOME` directory, ideally as a containerized image.

This workflow drastically reduces potential damage when access tokens are leaked since an attacker can only execute pre-configured jobs and can not gain full shell access which is required to exploit a local vulnerability. However, this illustrated workflow does not scale arbitrarily, since traversing through all prefixes by the agent is (unnecessarily) time-consuming. In addition, one has to implement any synchronization carefully since the eventual consistency of S3 does not allow syncing processes via file locks.

Method	Security	User Space	Limited Access	Transaction Safe	Scalability	Portability	Implementation Effort
SSH Password	-	+	-	+	+	-	+
SSH Keys	-	+	-	+	+	+	+
SSH Certificates	+	-	-	+	+	0	+
SSH Keys Force Commands	+	-	+	+	+	-	+
SSH Certificates Force Commands	+	+	+	+	+	-	-
S3	+	+	+	0	-	+	+
HTTP API Server	+	+	+	+	+	0	+

TABLE 5.1: Qualitative comparison of the different evaluated communication layers. A + indicates that the requirement is always fulfilled, a 0 indicates that it is not always fulfilled but might be, and a - states that this requirement is never fulfilled.

HTTP-based API Server

The idea of a dedicated HTTP-based API server has already been mentioned within Section 5.1. In this case, it is solely used to support the control flow of a job submission and to solve the shortcomings of the previously discussed S3-based method, although it shall be emphasized that it can generally be used in both, pushing and pulling methods.

Using a dedicated API server, instead of just an S3 bucket, allows in an otherwise equivalent setup, that a job is triggered by calling a specific endpoint on the API server. The job specification file is provided as a payload within this call. The API server has several advantages, like being fully transaction-safe and scaling at much lower costs since no tree traversal is required. In addition to these advantages, they can also be designed to be very secure and user-friendly, as the remainder of this chapter shows.

5.2.3 Evaluation

In Table 5.1 a qualitative comparison of the discussed method is shown, which summarizes the previously discussed results. Due to security concerns, both SSH password and SSH key authentication are not feasible. SSH certificates are due to their limited lifespan and their sole existence within the SSH agent more secure against theft but still allow full shell access to a third party. Using force commands with SSH keys is also undesirable due to the continuous effort, that admins have to add corresponding `Match` clauses within the `sshd_config`. Using SSH certificates combined with force commands seems like the ideal solution, however, implementing the necessary workflow to issue these tokens using the mentioned OAuth flow where a user can sign off on a certain command is extremely complex and costly. Using the discussed pulling methods has the advantage, that no complex development and setup by HPC admins are required. The discussed S3 method, however, lacks transaction safety and scalability. Thus, the method using a dedicated HTTP API server and using an agent that pulls all information into the user space seems like the most promising way to tackle this problem.

5.3 General Architecture

HPCSerA consists in total of three components that enable the access and remote control of an HPC system via a REST API. These three components as well as their

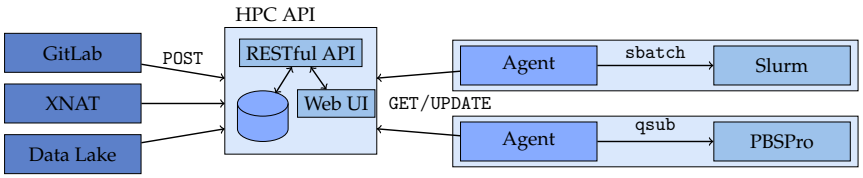


FIGURE 5.1: Components of the proposed architecture, consisting of clients, an API server, an HPC systems.

interactions are depicted in Figure 5.1: The main component is the API server, which at first glance looks like a simple message broker. Clients, shown on the left side in green, can use the REST API of the API server to post a new HPC task. On the opposite side, there is a cronjob running, in the following called agent, which periodically queries the API server for available tasks and pulls them if available. Once pulled, the agent will execute the task and will update the state of the task on the API server accordingly. This simple approach has several advantages:

- If the egress firewall rules allow access to the API server, which would be possible even for HPC systems which do not allow general internet access, the entire setup can be done in user space.
- The agent is independently configurable. This means that the agent does not require a fixed interface, like a certain resource manager, and can be customized to work with any kind of system.
- The agent can only do, what it is configured to do. Therefore, a user can configure what should be exposed. The highest form of exposure would be to allow arbitrary code ingest and execution, like sending a shell script and executing it. A smaller level of exposure would be to just allow the submission of preconfigured batch jobs to the resource manager.
- A user can hook an authorization mechanism into the agent in user space and therefore does not need to completely trust the administrators of the API server. This mistrust allows a large exposure of the agent in a secure manner.

In the following the three components are presented in more detail.

5.3.1 The API Server

As a central component of the *HPCSerA* architecture, the API server handles HTTP connections from the client and agent (described below), maintains the internal state of all jobs and functions, and resolves dependencies between functions. In addition, it provides the necessary maintenance endpoints to allow configuration via a web UI. It communicates with the database for the persistence of the internal state and verification of any authentication tokens. Since every job has to be kept in the database at least until it is completed, the API server is not stateless. All other connections are initiated by other components, therefore the API server is the only part of the architecture that has to allow incoming connections. It is also the responsibility of the API server to ensure separation between jobs of different projects, i.e., these are only visible in response to requests that are authorized for the same project.

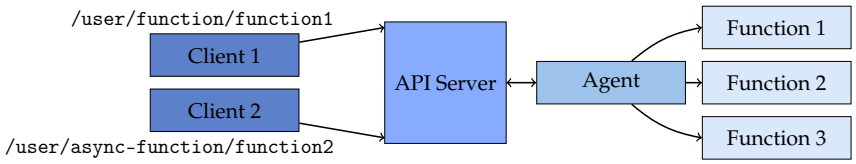


FIGURE 5.2: Basic schema of the FaaS methodology.

5.3.2 The Client

Any application or service that needs HPC resources as a backend implements the *Client* component, which initiates HTTP connections to the API server in order to submit jobs, call functions, and retrieve information on the job or function state.

5.3.3 The Agent

On the HPC system the *Agent* component regularly connects to the API server in order to retrieve jobs that are ready for execution. Depending on the function being called, the batch system might be involved and is regularly queried on the state of each pending or running cluster job. This information is used for further calls to the API server in order to keep the job state up to date. In the case of function calls which depend on each other only via the cluster jobs they need to run, a corresponding set of jobs including the dependency information is being submitted to the batch system.

5.4 Advanced Execution Models

Extending on this general idea, a more formal execution model can be defined. Generally one can observe that the execution model of predefined tasks triggered by a REST call is the aforementioned concept of FaaS.

5.4.1 FaaS for HPC

In FaaS it is typical that a user has a preconfigured task or function which is packaged into a container to be called with varying inputs. These functions are available by user-defined REST endpoints. Since in HPCSerA every user has a dedicated namespace on the API server, this expected behaviour can be replicated on an HPC system using the respective scheduling mechanism for batch jobs.

The basic mechanism for this is shown in Figure 5.2. It can be seen that the user can send REST requests to the API server resembling FaaS requests. For this, each user has their own namespace `/<username>/function/<functionname>`, where custom functions can be registered at their own discretion. It is important to state that the function name must be unique within the namespace of each user and is not being further isolated by additional structures like the notion of projects. Once a client has posted a function call to the API server, it will be available for the agent to be pulled with the corresponding GET request. The agent then actively pulls these function calls and dispatches them by calling a starter script with the same `<functionname>` located in a preconfigured path, e.g., in the user's home directory in `~/hpcsera/functions/<functionname>`. These functions, which are then being executed, can be anything. It can be a Bash script that is being executed on the frontend, it can be a script fetching data from a remote source and staging it on the fast parallel filesystem of the HPC system, or it can be a simple job submission to

the respective batch system, to give just a few examples. Since only executables are being executed, there are no inherent limits to the capabilities of these functions.

5.4.2 Long Running vs. Short Running Functions

Since HPCSerA does not enforce any boundary condition on the user-defined functions, it is important to differentiate between long-running and short-running functions from the beginning. The most important difference from the user's perspective is that long-running functions will be usually executed asynchronously, whereas short-running functions can be executed synchronously as well. The reason is that in this case a Transmission Control Protocol (TCP) connection between the client and the API server can stay established during the entire time. Therefore, the HTTP response will correspond to the output of the function, see Section 5.4.8, e.g., a response code `200` would directly mean that the function ran successfully. There might even be some payload data attached to the response, which can be immediately used by the client. The client process is blocked for the duration of the HTTP request. These functions, however, do not only need to have a short runtime, but also need to have limited resource requirements. In those cases, an oversubscribed queue (commonly used to enable interactive jobs) can be used, which can be created and managed by typical resource managers like Slurm.

In the other case, during an asynchronous function execution the client would get an immediate HTTP response from the API server. Here, the return code `202` would however only mean that the request to execute the function was successfully accepted from the API server. This allows for the established TCP connection between the client and the API server to be terminated. Therefore, the client process would only be blocking for the duration of the initial communication with the API server, but not for the entire time the function needs for processing. However, this leaves the client without the optional output data of the function, which might be required. This can be solved on the client side by providing a callback URL in the HTTP header when the initial REST request is made. The API server or the agent would in that case make a callback to the client once the function has finished. This is possible since the API server offers statefulness for the functions. Since the API server itself is not meant to handle large data transfers, usually S3 will be used for these cases. Therefore, it might be advantageous to implement some event handling using S3 rather than the API server.

About the differences between synchronous and asynchronous jobs which require access to the compute nodes that are managed by a dedicated resource manager like Slurm it can be stated from the HPC perspective that the synchronous case uses `srun`, whereas the asynchronous function call is using `sbatch`.

For HPCSerA a single function configuration is enough to execute the same function synchronously and asynchronously. The client can then choose the mode of execution at runtime and just distinguishes between those modes by using a different Endpoint, i.e., either the `/<username>/function/<functionname>` for the synchronous execution or the `/<username>/async-function/<functionname>` for the asynchronous execution.

5.4.3 Differentiation Between Control Flow and Data Flow

In general, one has to make a distinction between a control flow, and a data flow. The control flow represents the flow, i.e., the transfer of instructions that should be executed. One common example of such a control flow is SSH. On the other hand,

the data flow represents the transfer of data, for which SCP is a common example. As already motivated, HPCSerA is designed to enable only the control flow as a first-class citizen. Therefore, one could compare it to SSH, just that HPCSerA is using HTTP and the FaaS idiom, instead of providing access to a terminal session.

Although the usefulness of sending small payloads along with an HTTP request, for example as input data to a function, is acknowledged and should be offered as previously described, the general separation of concerns should be considered more seriously for larger payloads. Here, some HPC systems will even provide different nodes for these two different tasks. For instance, the usual user frontend nodes for the control flow, where the user can connect via SSH, or instantiating functions using HPCSerA, can be accompanied by dedicated data mover nodes, which are only used for transferring data.

There are different ways to move data to and from HPC systems via HTTP, ranging from deploying an NGINX in user space as used in Open OnDemand [87] for push access, to object storage systems, like the aforementioned S3, for a pull access. Considering this wealth of options, one can justify neglecting the data flow in a first attempt to build this system.

5.4.4 Remotely Building Complex HPC Jobs

Offering a FaaS infrastructure based on HPC which enables long-running, data-intensive, and highly parallelized functions is a useful addition for those users who are already in the FaaS ecosystem. There is, however, also the HPC-native user group, where people would like to be able to access and use an HPC system as before, just with a RESTful interface. In order to combine these two scenarios, a closer look at the typical HPC usage is necessary. The usual workflow for users working on HPC systems can consist of several steps:

- The environment and binaries for the computation are prepared. This is mostly done interactively on the frontend.
- Input data for IO-intensive applications is staged on a fast parallel filesystem prior to the job submission.
- Last changes to the batch script are done and the job is submitted to the batch system.
- After job completion the results can be inspected and possibly backed up.

Since there are no restrictions on the capabilities of the functions, one can recreate the workflow described above under two conditions:

- The execution of a function can depend on conditions.
- Code ingest needs to be supported.

The first condition is derived from the requirement that a job can only be submitted to the batch system once its environment is built and its input data is staged. There can also occur other examples and more complex conditions. Since HPCSerA is not in any way supposed to replace a workflow engine it is also not supposed to handle complex conditions on its own. Therefore, one can only add the condition to a function that it should start only after one or more other functions have (successfully) finished. The logic to determine whether a function call has been successful or not has to be within the function itself.

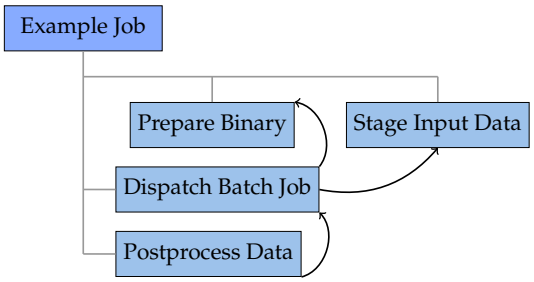


FIGURE 5.3: Sketch of a Job data structure in which four different functions are organized.

In order to build complete end-to-end HPC jobs with this mechanism these function calls need to be embedded in a suitable data structure.

In Figure 5.3 it is shown that all function calls are organized within a data structure called Job. A user has to first create a new Job, which gets a unique `JobID` assigned by the API server. Afterwards, a user can call functions within the context of a Job. These function calls then get a Function-ID associated to them. Conditions can be assigned to these function calls, i.e., other functions within this Job structure have to be (successfully) finished before this function can start. This mechanism allows building up a typical, multi-step HPC job as described above, by calling consecutively the exposed FaaS REST API. Independent function calls will be executed concurrently. In the example in Figure 5.3, this applies to both the *Prepare Binary* and *Stage Input Data* functions which have no dependencies. *Dispatch Batch Job*, on the other hand, can only be run once both previous function calls are completed. Finally, *Postprocess Data* is run once all other functions are completed.

Alternatively, one can define a single HPC job in HPCSerA using a single YAML file. In this case all functions need to be known when submitting the job request to the API server. If not specified, the functions are executed in the order they appear in the YAML file, and an implicit dependency on the previous function is assumed. In the consecutive buildup where independent functions are called in the context of a job, additional function calls can be issued to the same Job-ID at a later time.

5.4.5 Virtual Function Calls

Since all dependencies that HPCSerA is supposed to resolve should only cover the exit status of a function, i.e., with or without error, a mechanism is needed to map more complicated conditions onto this boolean. One example of such a more complicated condition would be that a function should only start after some special resource, like a certain block device, is provisioned.

To cover these cases, one can define *Virtual Functions*. These functions differ from normal functions in that they do not need to be pulled by the agent and then need to be executed. Instead, these functions are only existing on the API server. There they expose a REST endpoint, where from an external source the state of the Virtual Function can be changed. This means that some external program can make a REST call to that endpoint to set the state of the function to (successfully) finished. This is an alternative, similar to the call-back URLs provided by clients when triggering an asynchronous function. When an external REST call is used, the necessity to execute functions which do busy waiting to check if a certain condition is met can be avoided. However, functions that do busy waiting can also be used in a straightforward manner, as long as the necessary logic is implemented to differentiate between

the waiting state since the condition is not yet satisfied and the failed state where the condition will be never satisfied. In the latter case, the function should be terminated with the corresponding unsuccessful state. If a proper failure condition can not be formalized, when a condition fails and will not be met in the future, a final wall-clock time should be specified after which the function is terminated and the state of the function is unsuccessful.

5.4.6 Function Configuration

There are two different ways to configure and deploy a function. The first option is to connect to the HPC system via SSH and prepare the executable which is called when the function is triggered. This executable can be a binary, or a shell script, for instance. In case a binary should be executed within a certain environment, one can wrap the call of the binary within a shell script. If more complex environments are required, the executable can be packaged together with its dependencies into a container, e.g., Singularity/Apptainer [105] can be used. Once the function is configured within such an SSH session, it can be called afterward by the agent, therefore it is also immediately available for the client on the API server.

The alternative is to configure a new function via the API server, i.e., completely within the context of HPCSerA. For this, all necessary files can be zipped or tarred and sent along with the configuration request, which is available on a dedicated REST endpoint. The agent will then accept the archive, unpack it into a temporary directory, and will execute a preparation executable. This preparation executable can be as simple as just copying the function executable into the function directory of the agent. More complicated examples may include some code that needs to be compiled. For large files, like a Singularity container, it is recommended to upload those via REST via the dedicated data flow mechanism,e.g., to an S3 Bucket. Then just passing a preparation executable to the agent, which fetches this large file from the remote bucket and places it in the necessary path on the HPC system, is enough to configure such a function.

5.4.7 Passing Arguments to Functions

Some or even most functions will require that some arguments are passed to them when calling them. These can be passed to the API server of HPCSerA either as URL query parameters or as a JSON file. In the first approach, an arbitrary list of key-value pairs can be passed with the calling REST endpoint, e.g., `/<username>/function/<functionname>?k=val`. This call would forward the key `k` with the value `val` to the function in two possible ways: Either the agent would export an environment variable `<PREFIX>_k` with the value `val` (where `<PREFIX>` can be set by the user) before calling the executable corresponding to the called function or alternatively, these key-value pairs can be formatted into a single command line string which is appended to the binary call, as it is common when executing an executable on a Linux shell.

In case a function requires more extensive arguments, this previously discussed method is not handy anymore. Instead, one can use a JSON file which is sent along with the REST request to trigger the function. This JSON file is then simply forwarded from the API server to the agent which accepts the JSON file and stores it locally. The file path can then be passed as an argument to the function. Neither the API server nor the agent will in any way process the content of the JSON file. If a

function requires this kind of complex input data, the logic needs to be implemented by the function itself or a wrapper script.

5.4.8 Returning Output Data to the Client

When a function call is completed, the method of returning its results depends on the mode of execution and the volume of the produced data:

- For synchronously executed functions, see Section 5.4.2, the results are available by the time of completion and can be included in the HTTP response. If applicable, the results can be completely included in the form of a JSON structure produced by the function, e.g., for scripts that query the status of the system, such as custom calls of the batch system or storage Command Line Interface (CLI) tools. Binary data, such as base64-encoded Binary Large Objects (BLOB) can be included, although it is recommended, especially in the case of a high volume of produced data, that the JSON structure merely contains information about the location of the output data, for example, a file system path on shared storage or the URL of an S3 bucket.
- If the function call is asynchronous, only information about the data structures created on the API server can be relayed, in particular the `JobID`, or `FunctionID`. This has to be kept on the client side and used for later status requests.

5.4.9 Error Handling

Since the functions have a state, which is managed by the API server, and for instance distinguish between a successful and an unsuccessful exit, the user-defined functions are ideally able to distinguish between those states. However, some error in the code execution itself is not the only error that can occur. It could also happen that during the function execution the process is unexpectedly killed, or the host is suddenly turned off, for instance, due to a power outage. When the agent is able to detect those interruptions, where a function stopped processing without sending a proper exit code, it assumes that a crash unrelated to that particular function has happened and will trigger the function execution again. In order to support this behavior, a function should be idempotent, i.e., it should be possible to execute a function multiple times with the same input parameters and it will always produce the same output.

5.4.10 External Job Dependencies

With a slight modification in the data structure describing a job and the included functions, the proposed architecture can support medium-term storage of campaign data as well: The set of functions within a single YAML definition file is originally designed to be run in order, but a more generic solution is given by implementing a DAG of dependencies. Hereby, each function can define one or more dependencies on another function, which can either exist in HPCSerA or represent an external event via a virtual function. The latter is marked as done via an external source, for example when campaign storage or a data source is ready to be used as job input data. This workflow is typically used for research projects and can include dependencies between compute jobs, storage provisioning, and data migration. However, the conventional linear chain of functions is still included as the special case where

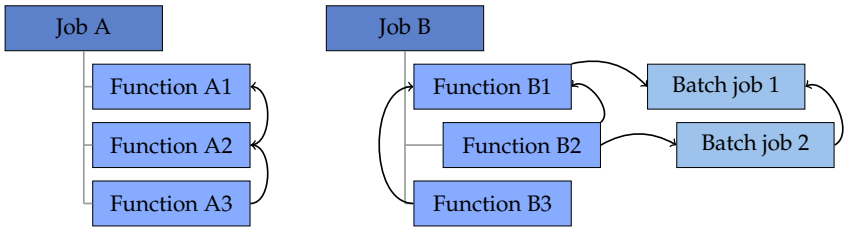


FIGURE 5.4: Jobs with implicitly defined dependencies and a custom dependency DAG.

each step depends on its predecessor. As shown in the first half of Figure 5.4 this variant is implemented via dependencies between functions. However, in the more general case, as depicted in the second half, multiple functions could depend on the same prerequisite, in this case `B1`. If a subset of the function calls is implemented via batch jobs and all other dependencies pointing outside of the set are already fulfilled, the corresponding subgraph can be submitted in one step, thereby delegating further resolution of the remaining dependencies to the batch system.

5.5 Security Architecture

The general idea for user authentication on the API server is based on static *bearer tokens*. The separation of access tokens by the user who created them and the services (clients and HPC agents) to which they are deployed, already enables revoking trust by the API server in a setup with multiple services and multiple backend HPC systems. However, during operation, there is global access to the entire state for all parties involved. In order to segment trust between groups of services and HPC backends, fine-granular permission management is required. Therefore, each token can be assigned one or multiple scopes, i.e., permissions, that restrict the combination of HTTP endpoints and verbs for which it can be used. The token's lifetime is implied by the granted scope but can be adjusted by an admin.

User control over each individual task and job that is allowed to be run or submitted, respectively, is enforced by introducing an intermediate authentication step that requires user interaction via an external application. This could be run on a mobile device or hardware token, like the ones being used for 2FA or integrated into the web-based user interface used for token and device management for fast iterations on the workflow configuration. Metadata about the action to be authorized is included in the user prompt in order to allow an informed decision. However, the measure is restricted to this most critical step of the process, while non-critical endpoints, such as retrieving the state of pending jobs, can continue to respond immediately. For submitting a new job, the necessity of individual user confirmation is also determined by whether new code is ingested or an already existing job is merely triggered to run on new input data.

From the user's perspective, setting up the workflow would start with logging into the web interface and creating tokens for each service to be connected to the API, and configuring them in each client and agent, respectively. In order to acquire a minimal working setup, at least one token for the client service and one for the agent communicating to the batch system on the HPC backend system would be required. OAuth-compatible clients could initiate this step externally, thereby sidestepping the need for the user to manually transfer the token to each client configuration. As

soon as each client has acquired the credentials either way, HPC jobs can be relayed between each service and the HPC agent.

While the OAuth 2.0 terminology [79] allows a distinction between an authorization server, which is responsible for granting authorization and creating access tokens, and a resource server, which represents control over the entities exposed by the API, in our case the tasks and batch jobs to be run, both roles are assumed by our architecture, so the design can be as simple as possible and deployed in a single step. However, since the endpoints for acquiring access tokens and the original endpoints that require these access tokens are distinct, a separation into microservices (which again need to be authenticated against each other) would also be compatible with the presented design.

The steps necessary for code execution are illustrated in Figure 5.5. As a preliminary, it is assumed that the HPC agent is set up and configured with the REST service as an endpoint. The arrows indicate the interactions and the initiator. The individual steps are as follows:

1. The workflow starts with a user logging into the web interface. The Single Sign-On (SSO) authentication used for this purpose has to be trusted since forging the user's identity could allow an attacker to subsequently authorize a malicious client to ingest arbitrary jobs.

2. The user can create tokens for the REST service in the web UI.

3. The tokens are stored in the Token Database (DB), along with the granted scope, project tag, and token lifetime.

4. The retrieved tokens can then be used by a client, e.g., to run some code on the HPC system or have an automatic process in place, provided the code is already present on the system, rendering manual authentication unnecessary.

5. The request is forwarded to the REST Service, which verifies the information in the Token DB. On success, the code to execute is forwarded to the HPC agent.

6. If the client chooses to use the OAuth flow instead in order to avoid manual token creation, the authorization request is forwarded to the Auth app instead.

7. The user can choose to confirm or deny the authorization request. In the former case, the generated token is stored (compare step 3) in the Token DB. Again, further requests can then in general proceed via step 5 without further user interaction.

8. Like any other client, the HPC agent uses a predefined token or alternatively initiates the OAuth flow in order to get access to the submitted jobs.

9. For the most critical task of ingesting code into the HPC system but also optionally for other tasks such as executing code on the HPC frontend or submitting batch jobs, the agent can be configured to get consent from the user by using the Auth app for authentication. This request is accompanied by metadata about the job to be executed, such as a hash of the job script, allowing an informed decision by the user. This step also avoids the need for trust in a shared infrastructure, since the authentication part can be hosted by each site individually.

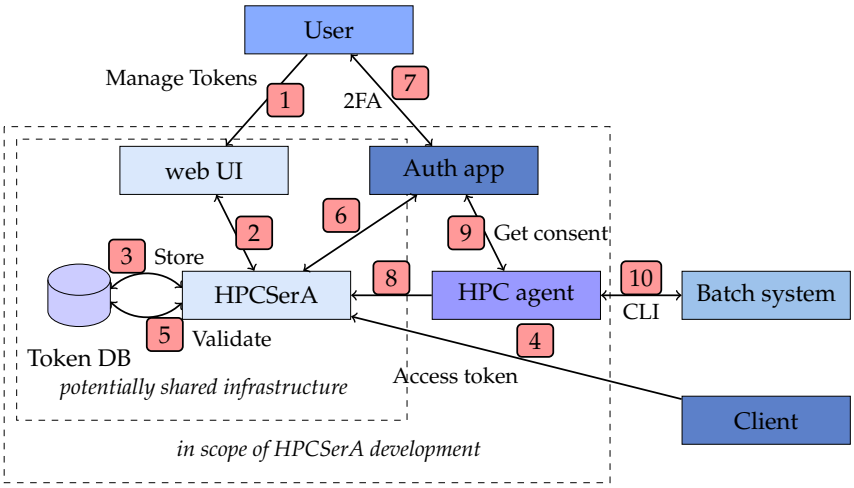


FIGURE 5.5: A sketch of the proposed token-based authorization flow.

10. Once the user has confirmed the execution, the HPC agent executes the code, e.g., by submitting it via the batch system. In this case, information about the internal job status is reported back to HPCSerA.

It can be assumed that the HPC agent is secure as otherwise the system and user account it runs on are compromised and, hence, could execute arbitrary code via the batch system anyway. The web UI, HPC agent, HPCSerA Service, and client are all independent components. For example, a compromised REST Service could try to provide arbitrary code to the HPC agent anytime or manipulate the user's instructions submitted via the client. However, as the user will be presented with the code via the authenticator app and can verify it similarly to 2FA, the risk is minimized.

There are two approaches for deploying HPCSerA across multiple clusters and administrative domains:

Replication Each center could deploy the whole HPCSerA infrastructure, compare Figure 5.5, independently, maximizing security and trust. By adjusting the endpoint URL, a user could connect via the identical client to either the REST service at one or another data center – this is identical to the URL endpoints in S3. Although the user now has two independent web UIs for confirming code execution on the respective data center, the authenticator and the identity manager behind it could be shared. An additional advantage of this setup would be that the versions of HPCSerA deployed at each center could differ.

Shared Infrastructure The maximum shared configuration would be that for each HPC system, a user has to deploy a dedicated HPC agent on a frontend node, but all the other components are only deployed once. As the HPC agents register themselves with the REST service, the users can decide at which center they would like to execute any submitted code. While using a single web UI for many centers and cloud deployments maximizes usability, it requires the highest level of trust in the core infrastructure: If two of these components are compromised, arbitrary code can be executed on a large number of systems. However, authentication for access to the web UI via the user's existing account from their HPC center can be implemented as SSO using OpenID Connect Federation.

Role Number	Role	Description
1	GET_FunctionStatus	Client can retrieve information about a submitted function
2	UPDATE_FunctionStatus	Used by client/agent to update the function status
3	GET_Function	Endpoint used by the agent to retrieve function information
4	POST_Code	Client to ingest new code to the HPC sytsem
5	GET_Code	Agent pulls new code. Might be necessary to run new function
6	POST_Function	Client triggers parameterized function
7	UPDATE_Function	Client updates already triggered function
8	DELETE_Function	Client deletes already triggered function

TABLE 5.2: Definition of the eight roles. Operations marked in red have to be considered security critical from the admin point of view, whereas the orange marked operations from a user point of view.

5.6 Implementation

In the following, more details about the technologies chosen for the implementation are provided with a focus on the current scope definition and the authentication/authorization scheme employed. Generally, the OpenAPI 3.0 [148] specification, which is a language-agnostic API-first standard used for documenting and describing an API along with its endpoints, operations, request- and response-definitions as well as their security schemes and scopes for each endpoint in *YAML* format, was used to define the RESTful API. This API is backed by a *FLASK*-based web application written in *Python*. The token database is in a SQL-compatible format, thus SQLite can be used for development and, e.g., PostgreSQL for the production deployment. The database schema contains only the user (`user_id`) and project (`project_id`) that the token belongs to as well as the individual permission-level (`token_scope`).

5.6.1 Definition of Access Roles

In order to give granular permissions for accessing each of the endpoints, OpenAPI 3.0 allows to define multiple security schemes providing different scopes to define a token matching to the security level of each of the endpoints. Eight different roles have been identified, which are listed and described in Table 5.2.

These roles are entirely orthogonal, which means they can be combined as necessary. If, for instance, on one HPC system only parameterized functions need to be submitted, the agent can be provided with a token that has only the permissions of roles 2 and 3, thus lacking role 5, which is required to fetch new files. Similarly, if a token is provided to a client that is not 100% trustworthy, one can choose to only provide a token with the role 6, i.e., to only allow to trigger an already registered function. Important to understand is the difference in mistrust between the roles 3, 4, and 5. The security mistrust in role 4 comes from the admins , who want to ensure that a code ingestion is indeed done by the legitimate user. Therefore, in order to allow code ingestion, the possession of a token with the corresponding permission is not enough, the user has to confirm the code ingestion via a second factor. The mistrust in roles 3 and 5 comes, however, from the user, who wants to ensure that only functions s/he confirmed are being executed. This is, again, completely orthogonal, to the enforced second factor in role 4 and can be optionally used by the user. This fine-grained differentiation between the different security implications of the discussed endpoints minimizes user interference while providing a high level of trust.

5.6.2 Providing Tokens via Decoupled OAuth

The introduction of OAuth-compatible API endpoints has several advantages: Access tokens can be created on demand in a workflow initiated by a client or HPC agent. In addition, while there is a default API client provided, a standard-compliant API enables users to easily develop drop-in replacements.

It is important to note here that the usual OAuth authorization code flow, where a client gets redirected to the corresponding login page to authorize the client is modified. This "redirect approach" has two problems:

- The client is a weak link, where the Transport Layer Security (TLS) encryption is terminated and therefore becomes susceptible to attacks and manipulation.
- It does not support a headless application, like the HPC agent, which is not able to properly forward the redirect to the user.

Due to these shortcomings, a modified OAuth flow is developed to enable the usage of headless apps and improve security. This modified version decouples the user confirmation from the client, which means that the client is not being redirected but that the confirmation request is being sent out-of-band, e.g., via the web UI or via notification on a smartphone.

Starting with the case that the script does not already come equipped with a token, analogous to the usual OAuth flow, the generation of a token is requested. Since the use case is initially built as an instance of machine-to-machine interaction, i.e., headless, the issue of a lack of user interface is encountered; the usual OAuth flow - implemented in the browser - would redirect the user to an authorization server where the user could actively provide their username and password to the authorization server. The authorization server would then return a code, in the case of the authorization code flow, in the redirect URI, which would be posted in a backchannel, along with a client secret assigned at the time of registering the client to attain an access token.

In order to circumvent this headless app problem a synchronous push notification system analogous to the Google prompt where a notification is pushed to a user's device awaiting a confirmation to proceed is used. In the Minimum Viable Product (MVP), it is implemented in the SSO-secured web UI in order to have a more integrated interface. Eventually, the final product will see an Android and iOS app that receives such notifications. This flow then grants the permission to execute a security-critical operation, compare table 5.2.

This confirmation via push notification cannot solely rely on time synchronicity since it would be susceptible to an attacker requesting tokens and/or 2FA confirmation for carrying out a security-critical operation in the same approximate time frame. Therefore, a sender constraint has to be implemented. This is done in a similar way to the original authorization code flow: The access code is signed with a client secret which is configured with the API server prior to the execution of this workflow, and then sent to the API server. The API server verifies the secret and only then sends the actual token. This secret is implemented using public-private key pairs, where the public key is uploaded to the API server in the initial setup to register a new client or agent.

Alternatively, in the case that a token is supplied along with the software or script that is submitting a job to the HPCSerA API, the permissions are validated against a token database. In the case that the token provided contains permissions for accessing a sensitive endpoint, the second-factor check is triggered through the web UI, and the notification/confirmation process is once again undergone. It is important

to note that this is not a hindrance since already-running functions and non-sensitive endpoints proceed without user intervention.

5.6.3 Mapping of Roles to Functions

In order to provide the user with a FaaS interface that is capable of handling automated machine-to-machine communication of headless apps the previously defined roles need to be mapped on the FaaS endpoints. The most important differentiation is still between the `POST_Function` role and the `POST_Code` role. The latter is required when a user wants to configure a new function via the API server. Here, the user can upload new code either directly as an archive, or via an external storage target. Therefore, the configuration of a new function corresponds to the `POST_Code` role. The client making that request needs to have these elevated access rights.

On the other hand, simply triggering the execution of an already configured function, for instance on new input data, corresponds to the `POST_Function` role. As shown in Table 5.2, this role is not security-sensitive. Thus, it can be used by a client without any manual interaction as long as the client has a token with the corresponding role. Therefore, HPCSerA can support automated FaaS functionality for its clients.

The agent side is not considered critical for the admins, but optionally critical for the users if they distrust the API server. Thus, they want to implement their own 2FA mechanism here. To support the previously discussed endpoints, the client needs to either use the `GET_Function` role to receive the request to execute a function or the `GET_Code` role to pull some new code and to configure a new function. The agent would then also require the `GET_FunctionStatus` role and the `UPDATE_FunctionStatus` role to manage the state of the functions, which is maintained on the API server. Via the `UPDATE_FunctionStatus` role the output data of a function can be send to the requesting client. The client would then also need the `UPDATE_FunctionStatus` role in order to be allowed to receive the output data.

5.6.4 Assigning Roles to Clients and Agents

The fine-grained distinction between those different roles, discussed in Section 5.6.3, is an important part of providing the highest level of security while enabling a high degree of automation. This means especially that users should only use tokens with the minimum roles attached to them. For instance, if a user configures functions always manually within an SSH session, the token of the agent should not have the `GET_Code` role enabled. Users define the roles of the tokens in the current setup within the web UI. Here, users can either check the needed roles, create a token, and copy it out of the web UI or they can, upon request from a client or agent via the presented detached OAuth flow, choose which roles should be associated with the token that will be created. Once a token has been created, it can also be revoked if it is not needed anymore or a potential breach is assumed.

5.7 Demonstration with Different Use Cases

The purpose of including the extra *Decomposition and Encapsulation* step with the following *Problem Generalization* step in the employed scientific method is to maximize the overall impact of the research artifacts. To prove the general applicability and its importance, the usage of HPCSerA in different scenarios is discussed in the following.

5.7.1 Implementing Workflows with an External Workflow Engine

In order to implement workflows with an external workflow engine to automate the orchestration, HPCserA can be used. Due to multiple repetitions and time dependencies, manual interactions severely limit the functionality and practicability of a workflow, thus HPC jobs should be fully automated without user interaction. Therefore, the use of pre-configured functions is encouraged. There are various levels where dependencies between jobs can be managed. The following descriptions and examples refer to Figure 5.6:

1. The preferred way to do the dependency resolution required to perform automated workflows is by completely handing over the involved logic to a workflow engine. In this case, workflow tasks are submitted as individual functions via HPCSerA. If there is a dependency between two jobs that require a batch job to finish, on completion of the first cluster job the agent updates the job state on the API server from which the workflow engine eventually obtains the new state. In the example, this is the requirement to proceed from Task I to the dependent Task II. Only then can the second job be submitted to the API and is finally retrieved by the agent and submitted to the batch system. In conclusion, this variant is the easiest to implement but involves a high amount of latency for resolving job dependencies.

2. For jobs that are submitted with multiple Function IDs, the API server will handle dependencies by only providing function calls to the HPC agent for which all function calls on which they depend have been successfully completed. Compared to the previous scenario, once the agent has marked the last batch job of Job A (A2 in the example) as completed, the function status of A2 on the API server is updated and the next one (function A3) can be immediately retrieved and run. While the dependency chain has to be implemented by building more complicated calls to the REST interface, there is no back-and-forth communication with the client contributing to the latency.

3. In view of Section 5.4.10 the most low-latency resolution of job dependencies occurs when multiple Function-IDs which contain batch jobs are presented by the API server to the agent. In this case, the completed first batch job (A1) directly leads to the scheduling of the second batch job (A2) by the batch system without interference from any HPCSerA components. Here, the resource scheduler of the HPC system takes over the dependency resolution at runtime. To do so, the agent has to forward these dependencies at submission time.

5.7.2 GitLab CI/CD

Since the *GitLab* Runner can be configured to run arbitrary code without including secrets in the repository, thanks to GitLab's project Continuous Integration and Integration Development (CI/CD) variables [68], the required tokens can be made available to the CI/CD job so it can in turn access the API endpoints required to transmit the current repository state to an HPC system where the code can be tested using the HPC software environment or even multiple compute nodes.

A new commit might of course introduce arbitrary code to the HPC environment, therefore it is advisable to enforce the extra authentication step when code from a new commit is submitted to the HPC system. The corresponding hash, available by default via the `GIT_COMMIT_SHA` variable, would be a helpful piece of information to display to the user when asking to authorize the request.

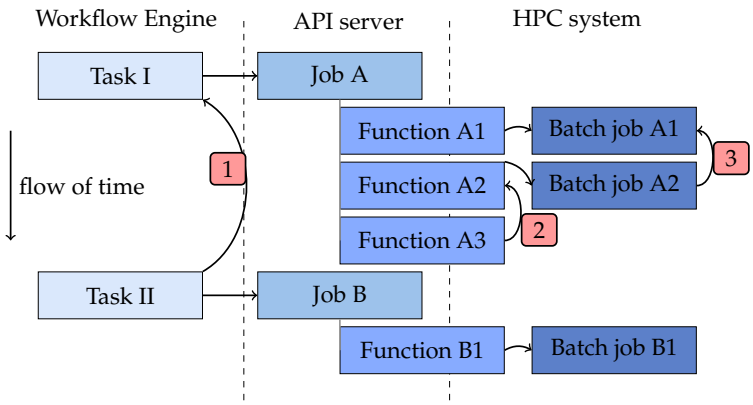


FIGURE 5.6: Overview of the levels at which function dependencies can be resolved.

5.7.3 XNAT

The Extensible Neuroimaging Archive Toolkit (XNAT) [119] is a research data management system for neuroimaging and related data. It organizes data within a hierarchical structure, starting with *Projects*, which consists of *Subjects* and *Experiments*. Individual measurements, e.g., an *MRI Scan*, are associated with a Subject via an *Image Session*. XNAT offers a we UI and a REST API, through which users can interact with the system. For instance, users can look at MRI scans in their browser using the web interface, or they can start an analysis task using a drop-down menu on the available data. These tasks are executed by pre-configured Docker containers. The corresponding Docker daemon can either run locally or on a dedicated server, which allows for some compute capacity to be provided. However, for large-scale and compute-intensive jobs for which an HPC system is favorable, no simple solution is available.

To provide one possibility to enable job offloading from the XNAT web interface to an HPC system, a curl-based client is built within a Docker container and registered within the XNAT *Container Plugin*. The container can be started using the *Container Plugin*. The user can then provide the function name s/he wants to execute on the HPC system within a pop-up window. Then the Docker container is started as usual using the *Container Plugin*, whose whole purpose is to send the corresponding REST request to HPCSerA. These requests are processed and the corresponding functions are started. Important is, that to enforce the access control lists of XNAT, data is always accessed by the HPC functions through REST API of XNAT. Similarly, resulting artifacts have to be ingested into XNAT via the REST API by the function to make the data available within XNAT.

5.8 Summary

In conclusion, within this chapter, a concept for a secure communication layer between a remote user interface, such as a data management system, and an HPC system is proposed. This concept enables a RESTful FaaS interface for HPC systems. Since these FaaS-based interfaces are increasingly common in different compute ecosystems, this interface is capable of providing ecosystem-agnostic computing infrastructure. Based on this idea, a general architecture is designed consisting of a stateful API server and a stateless agent, which is deployed on the HPC system. Leveraging the FaaS idiom within this HPC interface enables the definition of a

fine-grained role-based access model, which allows the execution of pre-configured functions on arbitrary input data without requiring the interactive confirmation of a second factor by the user. The remarkable aspect is that it is shown that this is possible without relaxing the generally applicable access restrictions, even if the HPC center requires a second factor for SSH access. This key characteristic enables the automatic execution of pre-configured pipelines on HPC systems without any user interaction. Based on this concept, a novel service called HPCSerA is developed and extensively tested within different use cases. In all of the presented use cases, HPCSerA is able to orchestrate the control flow between a remotely deployed client system and an HPC system. To summarize, HPCSerA can be used to provide a generic communication layer between a remote DMS and an HPC system and as such can be used to design a data lake architecture that uses a FaaS idiom to offload computations to cloud and HPC infrastructure depending on the compute intensity.

Chapter 6

Data Lake Design

Within this chapter, a general data lake architecture is designed, which can fulfill the requirements of the described use case as discussed in Section 2.2. For this, the advantages and disadvantages of the abstractions currently used in data lake designs are discussed in Section 6.1. Based on this, a novel Digital Object-based architecture is presented in Section 6.2. This design is extended to incorporate FAIR Digital Objects in Section 6.3. The applicability of the Digital Object-based data lake design is demonstrated in Section 6.4 by the usage within the "Big Data MRI"-project. Parts of this chapter are published in [21, 144].

Within this chapter, a novel data lake is designed to serve as an institution's central data repository. For this, it needs to integrate heterogeneous data from diverse sources, which requires a generic data modeling technique. This data lake should be specifically focused on researchers with data and compute-intensive projects, which require the integration of scalable storage systems and similar powerful compute infrastructure. However, this compute infrastructure should not be provided by a single ecosystem but should support diverse systems. This capability is a general lack in state-of-the-art data lakes as it is identified in Section 3.7.5. To provide scalable compute infrastructure, cloud and HPC systems are of particular interest since these are by far the most widely used compute clusters. Based on a detailed analysis in Chapter 4, a unified interaction paradigm, called the governance-centric interaction paradigm, shall be used to overcome the gap between the DMS-centric and the compute-centric interaction paradigms. In the end, both user groups should be supported. To provide the required communication layer between the data lake and the HPC system, as it is identified in Section 4.3.3, HPCSerA, which is presented in Chapter 5, is used. Since HPCSerA also implements a REST-based FaaS interface, this concept can be used to achieve the required ecosystem-agnostic interface, since it can also be used to connect to cloud resources. However, the integration of diverse compute infrastructure is only part of two of the seven in Section 2.2 identified objectives. Thus, the proposed architecture is discussed more broadly i the following.

6.1 Choosing the Right Abstraction

In all of the presented data lake systems in Chapter 3, abstractions have been used to structure the overall system. These abstractions are employed on different levels and by different stakeholders. For instance, the zone architecture consists of individually linked subsystems [66]. Here, these sub-systems are the individual pieces that are being abstracted in order to infer a hierarchy in an otherwise flat namespace. There are several disadvantages entailed when abstracting systems.

1. Admins providing such a data lake service have to do the abstraction step for the users by setting up and further managing dedicated systems. Therefore,

even later on during the production phase of such a data lake, the actual end-users can not do everything within user space but rely on active collaborations with the system administrators.

2. Depending on the granularity in which the hierarchy needs to be employed, multiple instances of the same zone need to be deployed. For example, if a differentiation between the k-space data and the scanner reconstructions, see Figure 2.2, is required on the zone level, two individual systems need to be deployed. That is, because according to [66] each zone should be identified by its interfaces to other zones, the modeling approach, user groups, data characteristics, including granularity, schema, syntax, and semantics, and the properties of a zone, e.g., protected, or governed. Thus integrating new datasets with new characteristics entails the introduction of zones, and since zones are abstractions of systems, the provisioning of new systems is required.

3. Since each zone is, to a large extent, an encapsulated system that is integrated via well-defined interfaces into the overarching data lake, it is difficult to obtain a global view. This is also the reason why data governance is defined and enforced on a zone level, not on a data lake level. This lack of global oversight is particularly disadvantageous for scientific use cases, where a globally enforced provenance record is required to adhere to good scientific practice. That is because the zone architecture lacks the possibility to guarantee a globally consistent state since data governance can not be globally enforced. This also extends without any limitations to a very fine-grained and distributed access management, which can only be done on a per-zone level.

4. The definition of a single zone is a mixture of the data contained in it, the user group it serves, the technical properties it has, and the interfaces it offers. Although it defines each zone uniquely and in a structured and reproducible way, this mixture can be confusing for users, who have rather a data-centric view, i.e., they just want to get access to some data.

On the other hand, the zone architecture has some undeniable advantages, which explains its success:

1. It manages to break down a large and highly complicated system with very diverse requirements, i.e., the overarching data lake, into small, and individually manageable pieces, i.e., the zones. Defining these individual zones can be done in a standardized and structured way. When combining these individual zones, developers only have to care about matching the interfaces of the zones between which data and optionally an information exchange should happen.

2. Working with these independent zones means that changes in each zone can be done completely decoupled from any other zone of the overarching system, as long as the interface stays compatible. For instance, within each zone, fine-granular data governance can be enforced, or a very specific user-facing interface can be provided without any consideration about possible consequences for other zones.

3. By sorting data into different zones according to their characteristics, e.g., their degree of processing, schema, syntax, or semantics, a hierarchy is inferred on the data lake. This hierarchy ensures the manageability of the data lake and the findability or explorability of data within the overarching system.

To summarize, one needs an abstraction that allows one to modularize a data lake into smaller, independent subsystems. Additionally, these abstractions should infer a hierarchy on the data lake, so that users and admins can sort and distinguish on a higher-level data and functionalities. However, this abstraction should ideally not be done on a per-system level and should not mix data association and technical capabilities. In addition, it should enable a global view of the available data within the entire data lake, and should not only offer a restricted view on a single subsystem.

6.2 Novel DO-based Data Lake Design

In order to address these shortcomings of the prevalent zone architecture, a new entity needs to be found, which can hide the complexity and technical details. Considering that a data lake is primarily all about data, these seem to be a good choice for a central building block.

6.2.1 Using the Abstraction of Objects

Using the abstraction of objects in computer science has a long and rich history of representing information in a more human-comprehensible way, ranging from the object-oriented programming paradigm to storage systems and digital objects [94]. For this novel data lake architecture, the notion of Digital Objects (DO) is reused to provide an abstraction of the data and forms well-defined, encapsulated entities which each can be interacted with independently of other constituents of the data lake. A DO in this sense consists in its core of the actual data, which one wants to store as a single entity. This raw data is described by semantic metadata with key-value pairs. Since each object, and therewith each atomic data item, is unique, each object requires a Unique Resource Identifier (URI) in order to resolve a specific entity. Each object must have a type. A type is uniquely defined by a name, and a schema, i.e., a set of mandatory and optional keys, and a target space, to which the corresponding values of the keys can resolve to, for instance, strings within a certain enumeration, which has to be part of an ontology or thesaurus, or integers within a certain range. These defined datatypes should support the concept of inheritance to reduce the manual, and in the case of updates, error-prone maintenance tasks, and further allow a more fine-grained, hierarchical organization, when compared to only relying on pre-defined, and generalized datatypes. For instance, raw data that is being ingested into the data lake should have a data type that inherits from a predefined *RawData* type. This infers the meaning, that the data comes from an outside source, ideally directly from a data source. On the other hand, artifacts, i.e., data that is created by processing input data from the data lake, should be modeled with a data type that inherits from a *ProcessedData* type. This automatically infers that these data types are derived by other input data from within the data lake, and therefore should have a matching retrospective provenance linking to their input data. Inheriting from an existing data type should always infer that the defined key-value pairs with the corresponding target space are transferred. Derived data types can therefore only further restrict or concretize an existing data type, but not generalize it again.

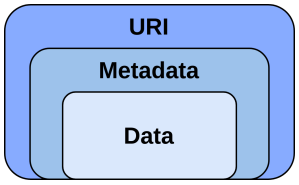


FIGURE 6.1: Sketch of a Digital Object.

6.2.2 Architecture Overview

The aforementioned DO's, which consist of data, metadata, and a URI, are rather a conceptual idea, than existing, physical data structures. Instead, these DO's have to be generated, in order to provide the described abstraction to the users. Therefore, the core component of the proposed data lake architecture is a central web application that generates the described DO's and their functionalities on the fly upon user request. For this, the data lake frontend then communicates with the required backend services, which are used for storing the core data, and the associated metadata, and possibly enabling data processing, compare Figure 6.1. This architecture can be seen in Figure 6.2. Here it is emphasized, that users do not communicate with the required backend services themselves, but use the provided frontend instead. This ensures, that all operations on the data lake use the same orchestration functions, which guarantees that the data lake is always in a consistent state. This architecture always allows admins to swap out existing back-end services without requiring any adoptions by the users for existing workflows and pipelines. This data lake frontend also provides a central and global view of the entire data lake. This data lake frontend is also one of the in Section 4.3.6 identified and required components.

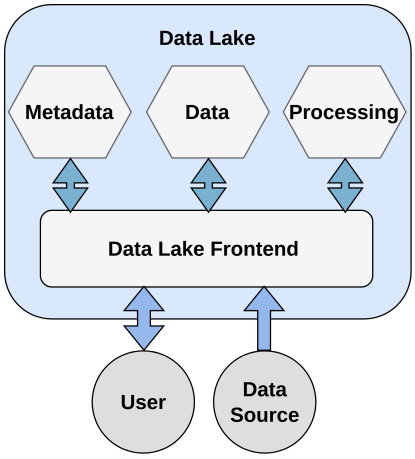


FIGURE 6.2: Sketch of the data lake frontend and backends.

Depending on the exact functionalities such a data lake should offer, there might be multiple back-end services necessary. Such an approach introduces a combinatorial problem for the overarching, data lake-wide governance. That is because each of these services will have their own mechanism to enforce certain policies, like Access Control Lists (ACL's). Ensuring a compliant state on the data lake would therefore mean that one has to ensure a consistent, homogeneous, and compliant state on all employed back-end services. This is, of course, a difficult, error-prone, and labor-intensive task and can be drastically eased if the governance is only enforced on a single system. This single system is the described frontend in this proposed architecture.

Even if some stakeholders decide to retain the full data sovereignty, for instance for sensitive data, and want to manage access control with use case specific workflows, the number of systems is still reduced compared to the zone architecture, since no additional distinction with regard to data, user, and interfaces needs to be made. In this case, it is sufficient to simply provide the required information on how to get access to the data, or metadata within the DO itself.

6.2.3 Ingestion

Although it was at first heavily debated, it is now widely accepted that a data lake, as the central data repository of an institution, should accept any data, i.e., any type and format, from any source [117]. Therefore, the ingestion layer of a data lake has to be highly flexible. This has in particular implications in two parts. First, the data transmission should use common protocols like HTTP. Here, it can be even more

favorable if a data lake is not only a passive data sink but can actively pull data from other systems in order to stay up-to-date. This can also reduce maintenance overhead since a lot of different and possibly heterogeneous data pipelines can be managed from a single system and interface. Second, the storage and metadata systems need to be able to cope, or at least easily adapt to, any kind of input data in order to be able to automatically extract descriptive metadata to update the central data catalog.

In order to be able to upload a new file containing raw data, the user first has to configure a new data type, containing the key-value pairs with their respective mappings. Then, data can be uploaded to the data lake, or pulled by the data lake from a data source, as such a previously defined data type. That means that every uploaded dataset becomes a concrete and unique instance of the abstract data type. The accompanying metadata to create the described DO can be delivered in different ways. Either a sidecar file is provided, whether manually written or automatically generated, or a metadata extraction function needs to be registered on the data lake prior to the ingestion of data. This function is then executed on a copy of the previously uploaded data and has to return the extracted metadata mappings of the corresponding data to the data lake frontend. Here, the provided metadata is checked in both cases against the provided data type definition, that all mandatory defined keys have a value set, and that all provided values are within the pre-defined target space. This ensures the data quality in the data lake.

6.2.4 Data Processing

After the raw data has been ingested into the data lake, the next step is to transform and/or analyze these data sets. The resulting artifacts should then also be re-ingested along with their provenance information. For processing the notion of functions is used. This intuitively enables the usage of FaaS-based compute infrastructure which can provide homogeneous access to different compute ecosystems like clouds or HPC systems, as explained in Chapter 5. Hereby, a function is a, usually pre-configured, piece of software which takes some input data, and computes some artifacts, i.e., some output data. Due to the compute intensity of most analytics tasks, outsourcing to a remote system is preferred over, e.g., local computation. There are in general two ways, in which such a processing task on a data lake can be performed.

The first one is inspired by JUNEAU, which performs the provenance auditing by capturing the code prior to its execution, by being a proxy between the users' input and the executing kernel. To reuse this idea when offloading compute-intensive tasks to another system, the remote compute services have to expose a suitable API for the data lake frontend to use, for instance, the REST API of HPCSerA. Then a user can describe the desired tasks unambiguously in a *Job Manifest*, which states the input data, the function to execute, as well as any run time arguments that get passed along. Depending on the concrete implementation of the function, its version is also tracked, e.g., in the form of a git commit hash. If a container is used to provide the necessary dependencies on the remote compute host, not only the recipe file can be tracked via git, but also the used container image can be uploaded into the data lake. This would then also be a DO, in this case, derived from a base type *Container*. Integrating the very image used to derive a certain result allows for better reproducibility, since the versions of the provided dependencies, e.g., libraries, can change over time, if an explicit rebuild of the image is triggered later on. In order

to offload the computation of a job to an HPC system HPCSerA, see Chapter 5, provides the necessary communication layer to execute a function and enables the data lake frontend to pass any required argument to the HPC system. A similar approach to offload tasks to a cloud system can be used with, e.g., OpenFaas [149]. Alternatively, by integrating different interfaces into the data lake frontend, also other techniques can be used like directly connecting to a remote Docker Engine.

The second option is to access the remote compute system, have the input data in the data lake staged or accessed from there, and perform the computation outside of the scope of the data lake's frontend. This approach has the challenge that users are now solely responsible to re-ingesting any artifacts along with their correct provenance information and further metadata. If this is done as a manual process it can be error-prone and therefore lead to an inconsistent, and unusable state of the data lake. However, this is a completely fine practice, if proper data management is done by the user. To support this, and bridge the gap to the first option, the governance-centric interaction paradigm is conceptualized in Section 4.4 which can be realized with the proposed DMP tool.

6.2.5 Publishing Analysis Tasks

Particularly the first presented option for performing analysis tasks on the data lake has the advantage that it can be easily shared among users. For this, developers can provide their software suite as described in Section 6.2.4, along with a generalized job manifest. The job manifest is generalized in that sense, that for instance, the input data is variable, and some task-specific configurations might be requested from the user, or are automatically derived by the metadata attributes of the DO's. Each task should explicitly check if the provided input data, in the form of a DO, is suitable for this specific job. Hereby one achieves an analysis service for new data which can then be used by other data lake users by uploading suitable data into the data lake and starting this pre-configured analysis task on it. For this process, the access policies of the uploaded data can still be completely private. The analysis service is accessed by its own unique resource identifier which can be resolved by the data lake frontend. Although using such published analysis services will be fairly easy for other users, since most of the configuration has been done by the actual developers, it should be able to provide, upon request, a minimum set of documentation of the available options. Here, particularly the specification of suitable input data is of utmost importance.

6.2.6 Linking Tasks to Workflows

In order to obtain an automated workflow these single tasks need to be chained together. Since the essential element of a data lake is the data itself, it seems reasonable to focus on data-centric workflows which are often represented by a DAG. Having all the necessary task definitions in place, i.e., the generalized job manifests as described in Section Section 6.2.5, the independent tasks simply need to be linked by their respective input and output data. This can be done in a DMP which describes the overarching data lifecycle. These are already used within the governance-centric interaction paradigm discussed in Section 4.4.1. There, tasks should be either generalized job manifests or published analysis tasks to offer a higher degree of automation. The data lake functions again as an additional abstraction layer between the workflow definition and the actual workflow implementation. As discussed in

Section 4.4, the DMP can bridge the gap between the data lake frontend as a synchronization layer, and remote systems like an HPC system. If external workflow engines, like CWL [3] or Apache Airflow, are required suitable adapters are necessary for that specific tool. These mappings also support portability outside of the context of the data lake to re-run and verify a workflow locally. However, the intermediary workflow manifest concepts make the data lake highly adaptable as it enables tailor-made solutions for individual projects while still enforcing well-defined and homogeneous provenance information and artifact handling.

6.2.7 Constantly Evolving Metadata

The originally envisioned schema-on-read semantics of a data lake need to be in practice accompanied by some metadata modeling if diverse sources and processing tasks are integrated into a homogeneous system. This initial contradiction can be resolved by allowing frequent changes on the employed schemata, and also on the individual entities inside the data catalog [98, 121]. These frequent changes are enabled in this proposed data lake architecture by allowing users and analysis tasks to add, or update the metadata of an DO. This process will again trigger the schema matching control to ensure the data quality within the lake. Using version control for the schema and storing the schema version information of the used template as part of the technical metadata of each DO allows for continuously evolving metadata, even in the case of changing attributes or changing mappings. One attribute, which can get frequently updated is the forward provenance, i.e., a link on DO's within the data lake pointing to a job manifest where these DO's were used as input data to derive some artifacts. These job manifests then link to their respective output data. On the other hand, the backward provenance is also maintained on the artifacts by linking to the same job manifest used to derive them. This process builds up a provenance-centered graph linking the DO's within the lake, where the nodes are DO's representing either input or output data, and the connecting edges are the job manifests, i.e., a *process* within the notion of the Open Provenance Model (OPM) [133]. The exact format, in which the provenance information is stored, e.g., OPM, PROV-O, or others, can be adjusted on the data lake frontend. However, during the project only the PROV standard is implemented.

In addition to the metadata attributes of an individual DO, also relationships between data types, apart from their lineage, can be continuously modeled. These relationships will be on the level of the data types, or schemata, not on the level of individual DO's.

6.3 Extending the Design Using FAIR Digital Objects

Although this DO-based data lake architecture has a lot of advantages compared to other existing systems, one until now unaddressed concern is the interoperability and the implementation effort particularly for the data lake frontend to create these abstract DO's. Therefore it is advantageous to re-use existing infrastructure for this concept. In this scenario, where a research data management infrastructure should be build, the concept of *FAIR Digital Objects*, which have been recently promoted, seem promising [51, 173, 175]. In the following it is investigated, how well these two concepts are aligned.

6.3.1 What Are FAIR Digital Objects?

Put simply, FAIR DO's (FDO) are DO's which are specifically designed to fulfill the 15 high-level FAIR principles [209]. One key component of FDOs is the globally uniquely resolvable and Persistent Identifier (PID) which is the reference used to resolve a single FDO. Therefore, one needs a PID service, a handle system, which resolves a PID to a PID record. This concept is utilized by FDOs by requiring a set of necessary information in these PID records so that it qualifies as an FDO record. An FDO record is a data structure of a fixed type, which is machine-readable, interpretable, and particularly actionable. This is achieved by providing not only the data as a bit sequence but also providing typed metadata attributes. Mandatory, optional, and recommended attributes, i.e., typed key-value pairs, are defined in a FDO profile. Each type has to be registered.

The requirement that each FDO has to have a registered type is derived from the goal of machine actionability. By relying on defined types, machines can automatically look up possible operations on these data types, or resolve the relations between types. Representing an FDO in a sketch similar to Figure 6.1 yields an almost identical representation, as can be seen in Figure 6.3. The largest difference is that FDOs have operations defined by their registered types. In the previously presented data lake, these functions, or operations are an attribute of the data lake. Defining functions on the level of an FDO has the disadvantage that publishing a new function as described in Section 6.2.5 requires access to the data type definition, which likely not every user will have. However, the advantage is that each object has information about how it can be processed.

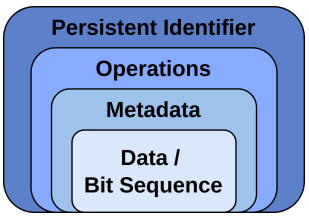


FIGURE 6.3: Sketch of an FAIR Digital Object.

Another difference is, that instead of a local URI, a FDO has a PID. Therefore, the immediate frontend when working with FDOs is the PID resolver which returns the FDO record, instead of the aforementioned data lake frontend. Only allowing to resolve PIDs is however not enough, since a semantic search over indexed data must be possible in a data lake. Generally, the definition of FDOs allows that the metadata entry in an FDO record links to an external service. This external service then has to a data catalog in the case of a data lake, which can also be accessed independently to query and explore the available data set without a concrete PID. Similarly, the data, i.e., the actual bit sequence can also be located on an external service, and only a reference has to be put into the FDO record. The communication between clients and an FDO-based data lake would follow the Digital Object Interface Protocol (DOIP) defined by DONA.

To summarize, one can build a data lake, as previously proposed with FDO by extending the PID resolver with the discussed additional data lake functionalities. This is generally possible and in no contradiction to the fundamental concept and was even foreseen, with the idea of "value-added reference services" [94].

6.3.2 Canonical Workflow Framework for Research

One particular challenge when moving from the more generic DOs to FDOs is the processing. In the DO-based architectures, functions were defined on the data lake frontend, whereas in the FDO-based architecture, at least some functions have to be registered in the FDO profile and are part of each object. However, in data-intensive

research projects often not only a single function is being executed, but entire workflows consisting of many dependent and independent function calls. Therefore it is important, that only some abstract functions like those required to access the actual data should be modeled at this level. All other, more advanced processing tasks should be, as described in Section 6.2.4, modeled as their own FDO. Here, the data at the core can be, for instance, the container image, and the functions defined on that FDO can describe how to execute it on some specified input data. This enables the publishing process of tasks, as described in Section 6.2.5, possible in a machine-actionable way.

To make these workflows and in particular the data they create more FAIR, the Canonical Workflow Framework for Research (CWFR) is proposed by the FDO-CWFR working group in 2020 led by Peter Wittenburg [1]. CWFR is supposed to be an additional layer on top of existing workflow technologies and provides the highest level patterns of recurring canonical steps. To reuse these canonical steps a library of canonical steps is provided, which can link each canonical step to the library of more specifically tuned packages. The described concept of CWFR further suggests storing all information in a single FDO and making each individual state of the workflow resolvable by its own PID. In particular, each of these PIDs resolves again to an individual FDO.

The previously proposed FDO-based data lake architecture is a concrete concept to implement the abstract idea of CWFR. The described recurring patterns are the user workflows. The canonical steps are represented by the generalized job manifests, which can be fine-tuned with respect to some input parameters to create a package repository of canonical steps. The overarching composite FDO representing the entire workflow and making each state individually resolvable via PIDs is achieved via the provenance-centered graph connecting input and output data.

6.4 Example Deployment for the MRI Use Case

To further emphasize how the previously described data lake architecture can be used, an example implementation based on the previously described MRI use case, see Chapter 2, is discussed in the following.

6.4.1 Data Modeling

The first step one has to do is to model the raw data that one wants to ingest into the data lake. In this particular case, the raw k-space MRI data shall be ingested. To do so, first the corresponding data type has to be registered. A new data type can be defined within a JSON document, stating the typename, in this case, `mri_kspace_data`, and assigning an arbitrary amount of key-value pairs to it. In this example, 59 different key-value pairs with domain-specific, semantic relevance are defined. The

```
{
...
"SystemVendor" : "Text()",
"SystemFieldStrength" : "Float(1.5,3,7)",
"NumberReceiverChannels" : "Integer()",
"Lamor_frequency": "Float()",
"BodyPart" : "Text()",
"PatientName": "Keyword()",
...
}
```

FIGURE 6.4: Short excerpt from the `mri_kspace_data.json` used to define the new k-space data type.

[1] https://osf.io/9e3vc

defined keys range from the system vendor, to the Lamor frequency, the bodypart and patient specific information like the name, weight, age, or sex, see Figure 6.4. In this particular case, only a relaxed quality control for the values is enforced based on the type. In this case, an empty constructor maps onto the entire valid value space. As an example, the `SystemFieldStrength` has additional arguments passed in its constructor, which will restrict this key to only the specified values. Using the REST interface of the data lake frontend, one can upload this new data type definition. Since in this overarching (F)DO-based concept each data type has to be unique, one can decide, whether the definition should be possible from user space, or of this requires a more privileged access. However, there are no technical limitations inferred by this architecture.

6.4.2 Data Ingestion and Metadata Extraction

Once this new `mri_kspace_data` data type is registered on the frontend, one can start to upload to it. Upon upload, the data lake frontend will try to use the data model and the uploaded data to create a concrete instance of the abstract data type. For this, it needs to populate the defined metadata keys with the corresponding information about the ingested data. This can be done by providing a sidecar file containing the metadata. Within this project, this step is also outsourced to the data lake. For the case of k-space data, which format is exported from the scanners in vendor-specific formats, the process of extracting all related metadata can be fully automated. In the case of Siemens scanners, the k-space data is stored in a `.dat` file. This file format has an extensive, utf-8 encoded header, which can be read out. For this particular project, a Python script that was developed by Martin Schilling is used for this. This script was packaged within a container image, which is uploaded to the data lake. Each container has a unique URI, with which it can be resolved. Further, each container image has a version, which enables updates of the metadata extraction tools while ensuring full resolution of the provenance.

To make this image executable, the execution of the container has to be configured on the data lake frontend. Here, the exact execution command, and the mount paths, where the input and output data reside, have to be provided. Once this is done, data can be uploaded to the data lake frontend and the container can be automatically executed on it. Once the metadata is extracted and the data of the DO is saved, the metadata is indexed in a document-based database to represent this single entity to have it searchable in a data catalog. The original k-space data is saved on a highly durable storage tier.

All of the previously mentioned steps can be done from a client without any complex configuration or dependency handling by only using the REST interface of the data lake frontend. However, it is very important to strictly enforce the separation of concerns. The data lake frontend should only orchestrate, abstract, and govern, not itself be involved in executing tasks other than that. Therefore, the frontend needs to have a backend service where this computation can be reliably done. In a naive first approach for this particular case, it is done by connecting to a remote Docker Engine. This can leave some entanglement between these two systems due to their blocking behavior, which further can lead to connection timeouts when the filesize, and hereby the data that has to be copied, or sent via UNIX sockets, increases. Therefore, a stricter separation with a well-defined interface is much more desirable, which allows for more fine-grained control.

6.4.3 Querying

Having ingested the raw MRI data, it is possible to select a single scan or a subset of the available scans by querying the data catalog, or by providing the URIs of the data of interest. Queries are defined by stating the attribute of the DO(s) of interest, i.e., the key of the metadata, and its value. For example with respect to Figure 6.4, one can define the simple query `attribute:SystemFieldStrength, value:3`. This query will select all the available DOs that have a key `SystemFieldStrength` set to the value of `3`. Important to note is, that this query is per default executed on all DO types, thus not only DOs of type `mri_kspace_data` can be selected, but also others, which also poses this attribute. By defining the type of the DO in question, the results can be further restricted. Multiple key-value queries can be defined and concatenated by an *and* operation. Concatenations based on *or* operations can mapped by sending several independent queries and merging the resulting data sets.

6.4.4 Processing

Once a suitable query is constructed to select the precise data set one is looking for, this data set can be processed. For this, the notion of a job manifest is used, which is a JSON document, which unambiguously describes the job. To do so, different options have to be provided by the user:

- A list of input data, or a query for the data catalog which resolves to the desired input data.
- Ideally, a container image is specified that contains all the dependencies a job needs to run. If this container image is ingested to the data lake it is also represented as a DO within the data lake. The URI of the container DO can be used to link resulting artifacts to it, significantly enhancing the quality of the provenance data.
- Similarly, code that should be built dynamically for each job execution should be version controlled, e.g., by using `git`.
- Environment variables which shall be exported into the job context.
- A single or multiple compute commands that should be executed.
- Specific resource requirements, like the expected compute time, the number of CPUs and Graphics Processing Units (GPUs), or conditions like access to the internet.
- The output data directory can be specified as well as the datatype of the DO as which the artifacts should be ingested into the data lake.
- Further comments can be added to aid comprehensibility.

Using the REST API of the data lake frontend, such a job manifest can be submitted to be executed. For this, the entire job is split into three distinct phases: pre-processing, processing, and post-processing. Since these processing tasks are generally assumed to be compute-intensive and/or data-intensive, the computations are done on an HPC system.

Within the pre-processing phase, the input data is staged. For this, a list of URIs is created based on the user-provided query. A data staging script then loops over this list and pulls all stated input files. If no other storage target is defined, the input files

are stored on a Scratch, or Work filesystem. These filesystems are typically accessible from the high-speed interconnect, which offers a high bandwidth and low latency, but do not offer any backup, and are only accessible for a limited amount of time. Since all input data is already stored on a durable storage system, these limitations are neglectable.

For one particular reconstruction pipeline, a fine-grained data staging mechanism is chosen. Here, the kspace data are stored in a specific folder structure given by `/kspace/hum_$NUMBER/$DATE/`. The `hum_$NUMBER` is a pseudonym representing the patient name. This folder structure allows to store all scans taken at different dates of one patient within a single path in a structured way. This exemplifies that input data does not need to be staged within a flat folder structure but can be customized.

Once the input data is staged, including the container image, the required software can be dynamically built, if specified in the job manifest. In the different processing tasks involving the reconstruction of k-space MR images, this is typically done by using two distinct git repositories. The first one contained BART, which is for each case typically built using a distinct branch. The second repository is then usually a use case-specific one, containing some driver scripts. These repositories are cloned and bind-mounted into the container where these repositories can be built. The container image then provides the dependencies like `libfftw3`, or `libopenblas`, which require only infrequent updates. On the other hand, the code that is actively being developed can be updated with every new job execution. This offers large time reductions when compared to rebuilding an entire container image for each job. Additionally, saving these images would require large amounts of storage, and would therefore be unfeasible. The git commit hashes are updated within the original job manifest.

Once the pre-processing is finished, the job can be submitted by evaluating the compute commands the users stated in the job manifest. For this, the user-specified environment variables are passed to the context of the job, like the `TOOLBOX_PATH`, which provides the location of the `bart` executable.

One example of such a job is developed by Moritz Blumenthal, where he reconstructs the staged k-space images within a fixed folder structure. It uses first the `bart ecalib` command to auto-calibrate for the different coil sensitivities using the *ESPIRiT* method [199]. Then, using the `bart pics` command, a parallel image (SENSE) reconstruction using an L1 wavelet regularization with a speedup factor of 2 is performed [25, 114, 161]. Afterward, the images are formatted to the Neuroimaging Informatics Technology Initiative (NIfTI) file format to be available for the next processing steps. These are stored within a similar folder tree, prefixed by `reco/$VERSION/` instead of `kspace`. Each image reconstruction is hereby a single batch job, and the corresponding batch script is stored for each image. Further metadata information can also be provided within the filesystem by a sidecar file, as it is done for the provenance information that is stored as PROV document in JSON format. All information is also indexed within the data lake, allowing for efficient searches to retrieve specific file paths.

Once the job is finished, the post-processing is started. Here, the resulting artifacts are ingested into the data lake. For that, the directories, which were bind-mounted into the container during the job execution, are checked for new files. This default behavior can be overwritten by explicitly stating the directories or files that should be ingested into the data lake within the job manifest. Upon ingestion, these artifacts are modeled as DOs and are linked to the job manifest to store their lineage.

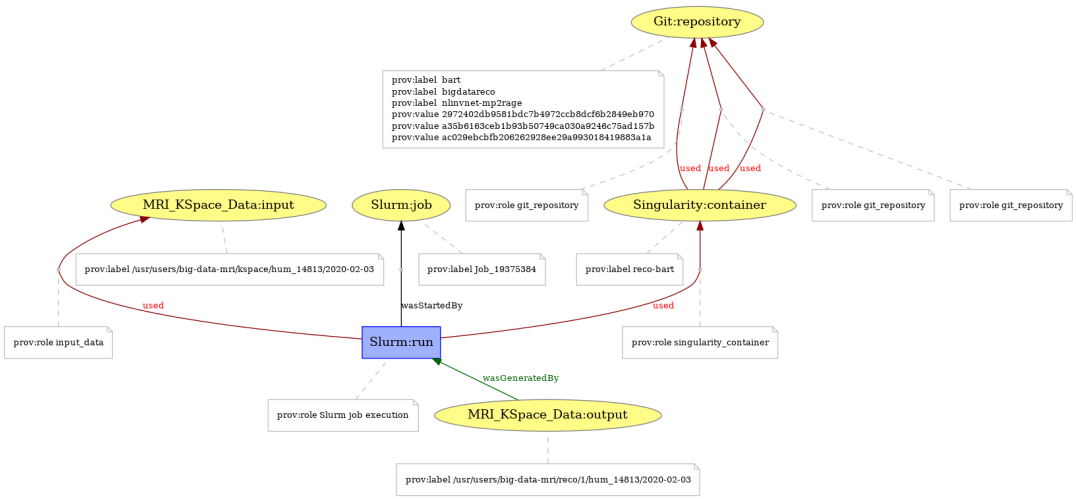


FIGURE 6.5: Image of the DOT code serialization of the PROV document of one of BART k-space reconstruction jobs.

In addition, their provenance information is automatically attached as a *PROV* document. One example of this is shown in Figure 6.5, where a DOT serialization of one of the PROV documents is plotted as a graph. Here, one can see, what the paths are for the input and output data, what the Slurm job ID is, that a Singularity container is used, and that three git repositories are bind-mounted into the container along with their corresponding git commit hashes. These PROV docs are created automatically for every HPC task and are stored as a sidecar file within every output directory and are indexed for each processed data object type under the *provenance* attribute. Therefore, searches with respect to certain job specifications, like what data has been processed with this particular git hash, are possible.

In this example, there are three calls to HPCSerA required: the first one to trigger the execution of the preprocessing, the second for triggering the actual data processing, and the third one to trigger the postprocessing. That is because each of these phases is a single function. The pre- and postprocessing functions are quite generic and can be reused across different use cases. The processing step is the actual step where user and job-specific code is executed. To execute this automatically with HPCSerA, this code, and its environment, e.g., a specific container image, has to be provided beforehand, for instance by uploading it as a DO into the data lake.

6.4.5 Interacting with the Data Lake from the HPC system

The k-space data and the reconstructed images are not only represented by DOs within the data lake and would be therefore solely accessible via the data lake frontend but are also stored on an HPC system within a specific folder structure. This is done since some of the researchers are used to work with and organize their data in a specific way. These researchers and their processes require that their datasets are organized according to the Brain Imaging Data Structure (BIDS) [71]. An example of such a folder structure is shown in Figure 6.6, a shortened version provided by the authors of the format[2]. The proposed DMP tool, see Section 4.4, can in this case be used to ensure that the data set is indeed always stored according

[2] `https://bids.neuroimaging.io/`, Accessed 17.12.2023

to the BIDS standard, like the overall folder structure, the individual filenames, that all subjects are also listed within the top-level `participants.tsv` file, or that for all derivatives the provenance information is provided by the `GeneratedBy` metadata field within the corresponding sidecar file. An example of such a sidecar file is shown in Figure 6.6 where the `sub-01_task-rest_bold.json` contains the metadata of the `sub-01_task-rest_bold.nii.gz` scan. The provenance recording is not only retrospectively controlled by the DMP tool, but is also supported beforehand since explicit linking of every job invocation with a task within an experimental description is required. This not only allows researchers to work with their dataset as usual, for instance using the BIDS Manager Pipeline [93], but also offers additional quality control.

```
dataset/
    participants.tsv
    sub-01/
        anat/
            sub-01-T1w.nii.gz
        func/
            sub-01_task-rest_bold.nii.gz
            sub-01_task-rest_bold.json
    sub-02/
    sub-03/
```

FIGURE 6.6: Example folder structure according to the BIDS standard.

One of the advantages is that the DMP tool, as it is introduced in Section 4.4, would also be capable of monitoring the folder structure for new subjects, and derivatives and automatically triggering an ingestion into the data lake using pre-configured DOs. This enables the intuitive integration of these neuroimaging datasets and workflows with completely different data to allow other researchers to search for correlations between neuroimaging data and, e.g., information about the subjects' physical activity or diet. These novel tasks can then be performed on integrated data that is not bound to the BIDS format. Therefore, the overall availability of these datasets is increased for researchers working outside the scope of the BIDS community. When ingesting these datasets into the data lake, all entities get indexed within the overarching data catalog. This also allows for efficient search for specific data, e.g., those created by a certain version or hyper-parameter combination. In the example mentioned in Section 6.4.4 this is just indicated by a number at the top level of each dataset, which then requires a dedicated sidecar file for each directory stating the exact provenance information for this workflow.

6.5 Evaluation of the Proposed Architecture

Based on the usage of this proposed data lake within the "Big Data MRI"-project an evaluation with regard to the in Section 2.2 defined objectives is made in the following:

Data Modeling: By modeling data as an abstract, typed DO based on pre-configured schemata, one obtains an entity-based metadata model, which is generic and allows for fine-grained quality control. These DO can hierarchically organize an otherwise flat hierarchical data lake based on their types. In addition, these DOs offer a simple yet powerful mental model for humans to interact with the constituents of the data lake to easily comprehend their function and content.

Generality: In this DO-based architecture, the data itself is used to organize and infer a hierarchy based on its intrinsic value. This allows for the integration of much more diverse data in a single system since no assumptions about the data and the

envisioned tasks have to be made a priori. This is also reflected, in that every datum can be modeled as an object, including processing tasks. This is fostered by the capability to continuously evolve the DOs and their definitions homogeneously over time, and not be limited by presumptions made at the beginning which later turn out to be incorrect or incomplete.

Modularity: The DO-based data lake design is highly modular, as it offers a homogenous interface for a potentially, but not necessarily, heterogeneous backend. This allows the complexity of multiple storage systems, databases, and compute systems to be abstracted away for the users. This, in turn, enables admins and data engineers to include different systems in an efficient way, since only the interface between a single component and the data lake frontend needs to be handled, but not every component has to work with every other component. This is further emphasized when extending to FDOs, since the FDO records only require a machine-actionable instruction on how to retrieve the data and metadata, but do not require the actual information to be centrally accessible.

Performance: Due to the integration of diverse compute infrastructure, including HPC systems, computing tasks can be processed very efficiently. Enabling different backends not only for compute but also for storage and metadata indexing enables users to optimize for specific scientific workflows.

Scalability: Due to its modular design, the DO-based data lake is highly scalable, since the underlying systems, providing the required functionalities like storage and compute, are not bound to a single specific implementation. Instead, they can be chosen to offer the aspired scalability. In this sense, dispatching tasks to an HPC system or cloud infrastructure is highly scalable by design, since these systems are built for this specific purpose. Similarly, highly scalable storage systems can be used, to retain all data.

FAIR Data: Extending the DO-based architecture to use FDOs will provide FAIRness by design. That is because the employed DOIP is specifically designed for the purpose of fulfilling the 15 FAIR principles.

Continuous Integration into Scientific Workflows: Due to the generally applicable metadata modeling, as well as the modularized design which allows to exchange or add specific back-end components, the proposed data lake can continuously integrate new scientific workflows.

Data Lifecycle Management: The overarching data lifecycle management is done within the concept of the governance-centric interaction paradigm discussed in Chapter 4. This allows users to define an envisioned state and continuously ensure its consistency with the defined DMP, while still enabling the use of different interfaces.

Security: Generally, the DMP is capable of ensuring that permissions are set correctly, i.e., ensuring that a user does not accidentally provide access to private data. However, this requires that the permissions are always correctly enforced. In the

case of a privilege escalation, there is currently no mechanism to prevent unauthorized access to sensitive data or metadata. Therefore, additional measures are required to isolate the underlying services from these attack scenarios.

6.6 Summary

In conclusion, a generic and modular data lake design is presented. It eases the administration by providing a more suitable abstraction in the form of DOs compared to the prevalent zones. The organization of this potentially flat namespace is done via unique data types, the definition of which can then be used for quality assurance upon data ingest. The proposed data lake provides scalable and ecosystem-agnostic compute resources based on a private cloud and an HPC system by hiding their peculiarities behind homogeneous FaaS-based interfaces. When executing these functions on the data lake, transparent provenance auditing is done via the provided middle layer. The corresponding input data for these functions can be selected using semantically meaningful queries and are automatically staged if necessary. The proposed data lake is successfully utilized within the MRI-based use case, by automatically extracting all available metadata from the `dat` files and indexing it in a data catalog. Once the raw data is available, one can select the data using a list of identifiers, or a query to reconstruct the kspace images using `BART` on the HPC system.

The concluding evaluation shows that except for the full data sovereignty, i.e., security, all the identified requirements are fulfilled. Due to the modular design, data privacy can be consecutively included, by securing the individual components of the data lake. These are, on a high level, storage, compute, and data cataloging. Since data storage is an implicit part of its processing, it is sufficient to focus on the processing and metadata handling to secure the data lake. Since metadata handling, i.e., indexing, updating, and searching through a data catalog, can be considered to be a specialized form of processing, a generic platform for processing sensitive data is developed first in Chapter 7. Afterward, the obtained knowledge about the secure processing of sensitive data is applied in Chapter 8 to develop a secure data catalog service.

Chapter 7

SecureHPC: A Workflow Providing a Secure HPC Partition

Within this chapter, a secure partition on a shared HPC system is developed which enables the processing of sensitive data. For this, first, the relevant related work is explored in Section 7.1, followed by a general introduction to the architecture of HPC systems in Section 7.2. Then the general design of the workflow is discussed in Section 7.3, with which access to the special secure partition can be obtained. Afterward, implementation details of this workflow and the partition are provided in Section 7.4, followed by a dedicated presentation of multi-node support in Section 7.5. Based on this design a security analysis is done in Section 7.6. Then extensive benchmarking is done to determine the additional cost associated with the extra security in Section 7.7. Lastly, the applicability and real-world performance are demonstrated in three different use cases in Section 7.8. Parts of this chapter are published in [145, 146] and are submitted to [141].

Until this point, the in Chapter 6 proposed data lake is only capable of managing, i.e., storing, indexing, and processing, insensitive data. Due to the specification of the DOs, it is possible to store sensitive data by only uploading it encrypted [94]. Leveraging the modular design of the proposed data lake, it is possible to consider the processing of sensitive data as an isolated service, which can also be used outside the scope of the data lake. Once available, it can be used as a drop-in replacement for the current "normal" HPC service. For this, it needs to support the previously utilized FaaS method so that it can be exposed to the proposed data lake using the in Chapter 5 developed interface, called HPCSerA. Processing highly regulated data, see GDPR or HIPAA, requires a special setup since HPC systems are typically built for the highest performance, not the highest security. That is because different users share the available resources and can run their compute jobs simultaneously on shared or an exclusive subset of the available nodes. Due to the optimization for performance, it is very common, that users interact directly with the operating system of the host. Therefore, users are trusted to some extent and, thus, any local vulnerability can be immediately exploited by users or bots that gain control of user credentials. Taking into account that there are continuously new attacks discovered that lack a reliable solution over a sustained period of time [92], sensitive data should only be transferred, stored, and processed with care in public data centers. Some industry and government data centers, such as for weapon research, limit access and employ strict policies regarding system access, even to the point where sensitive data is physically disconnected if not needed. However, restricting system access does not resolve the problem with the data access, since administrators basically have full access. Therefore a novel service is required where even in the case

of a privilege escalation leading to a compromised cluster, the data integrity should be guaranteed. This general need for secure compute capabilities and the resulting requirements for refinement of existing security concepts, particularly for HPC systems, is acknowledged in the literature. Christopher et al. describe in [44] that "At UC Berkeley, this has become a pressing issue" and it has "affected the campus' ability to recruit a new faculty member".

7.1 Related Work

In BioMedIT [46], a distributed network is described, where virtualization is used to create completely isolated compute environments that are exclusively reserved for a particular project, in a private cloud. However, the use of virtualization for isolation purposes is only effective, if it is ensured that other users can not access the host system directly. Therefore, this approach requires dedicated hardware and software, which drastically increases the cost of hosting such a system.

A similar approach is described in [170], where *Private Cloud on a Compute Cluster (PCOCC)*[1] is used to deploy a private virtual cluster. The outlined Slurm integration allows for direct integration into an existing HPC system. However, a dedicated Lustre file system is needed and it remains unclear how this virtualized cluster is secured against a compromised host.

Systems like *SELinux* [185] or *AppArmor* [10] are using the *Linux Security Modules* to enforce mandatory access control with the general goal to prevent zero-day-exploits. However, once an attacker finds a vulnerability these mechanisms become useless. Here, intrusion detection systems, like *auditd*, could be used to detect such a system [95]. In contrast, the goal of the envisioned secure HPC partition is to ensure data privacy under the assumption that a root exploit is already successfully done. Therefore, these two approaches work completely independently of each other.

In order to limit the actions a malicious root can take on a compromised system, one can use kerberized filesystems [129]. This would then, however, limit the functionality of non-interactive batch jobs, and/or would require the additional usage of e.g. *Yubikeys* by all its users.

One possible way to isolate a single task on a multi-tenant node is the use of *Trusted Execution Environments* (TEEs). Here, access to sensitive data or code, which is loaded into memory, is secured from access from the host kernel. There exist several different solutions, including commercial solutions like Intel's SGX [122] and open-source solutions like *Keystone* [107] which are based on basic primitives provided by the respective hardware. In order to utilize those so-called *enclaves*, changes to the source code of the corresponding application are necessary. To mitigate this issue, solutions like *Graphene* [197] enable users to run unchanged code within an enclave. Similarly, *SCONE* [7] is developed to support Linux secure containers for Docker. These solutions for TEEs are very interesting to secure and isolate a running process, including its data, from malicious access but are in itself not sufficient to provide an end-to-end workflow to securely upload, store, and process sensitive data on an untrusted, shared system. In addition, there is currently no sufficient solution to provide access to an accelerator, like a GPU.

Containers are processes that are executed on the host operating system and are pseudo-isolated by *namespaces* and *cgroups*. This allows the provisioning of a private root file system in order to execute software in a portable environment. As with any

[1] https://github.com/cea-hpc/pcocc

other process, containers are executed with the rights of the user, which can be extended with a *setuid*-bit. Containers solely isolate the processes running within their context from direct access to the host, not vice versa. Therefore, even an encrypted container is without the usage of a TEE decrypted in kernel space and can be fully read by the host.

Secure storing of sensitive data on shared, untrusted storage on an HPC system is explored in [186]. Here, *Ceph Object Gateways* [207] are deployed on single-tenant compute nodes alongside an *S3FS* which bind-mounts the corresponding S3-Bucket as an POSIX compatible directory onto the host. The host-based configuration then performs automatic encryption/decryption of data that is written/read to/from this specific directory. While this is an important part of secure data processing, it is only a part of a holistic solution that is required to provide a secure end-to-end workflow. For instance, features such as secure transfer of the necessary keys for accessing the S3-Bucket and for performing the decryption/encryption are important. Additionally, some data center policies may require a strict separation between the HPC and the storage networks.

Other ways to securely store sensitive data on a typical filesystem include *Linux Unified Key Setup*[2] (LUKS), *eCryptFS*[3], and *GoCryptFS*[4]. LUKS, which is based on `dm_crypt`, provides encryption for a block device and allows for multiple passphrases to be used. On the other hand, eCryptFS and GoCryptFS offer filesystem-level encryption which is based on a File System in User Space (FUSE) filesystem. To allow for efficient file access while still using a block cipher, they are splitting each file up into independent blocks, or *extends* as it is called in the case of eCryptFS. The advantage is, that random access to a file is faster since only each involved block needs to be de- or encrypted, instead of the entire file.

In contrast to the related work, a blueprint for a holistic end-to-end processing pipeline suited for sensitive data to be used on a shared, untrusted HPC system is required. This is complemented by a detailed discussion about the security implications as well as extensive benchmarking to assess the general applicability.

7.2 General Usage of HPC Systems

This section describes the architecture of HPC systems, as well as the general usage of HPC systems. Based on this introduction, a security risk analysis is performed to determine different attack vectors that can be used to compromise the data privacy of sensitive data during transit, at rest, or during compute.

7.2.1 Architecture of HPC Systems

Generally, HPC systems are composed of different node types. They serve different purposes and have, therefore, different security policies applied to them. In the following, an overview of typical node types is provided and their interactions are explained. This will further serve as the basis for the nodes that are deemed secure, even in the case of a privilege escalation of a user.

The general architecture of an HPC system is illustrated in Figure 7.1. HPC systems are commonly guarded by a perimeter firewall, requiring users to connect via a Virtual Private Network (VPN) or a jump host. Afterward, they can log in via SSH

[2] https://gitlab.com/cryptsetup/cryptsetup/
[3] https://www.ecryptfs.org/
[4] https://github.com/rfjakob/gocryptfs

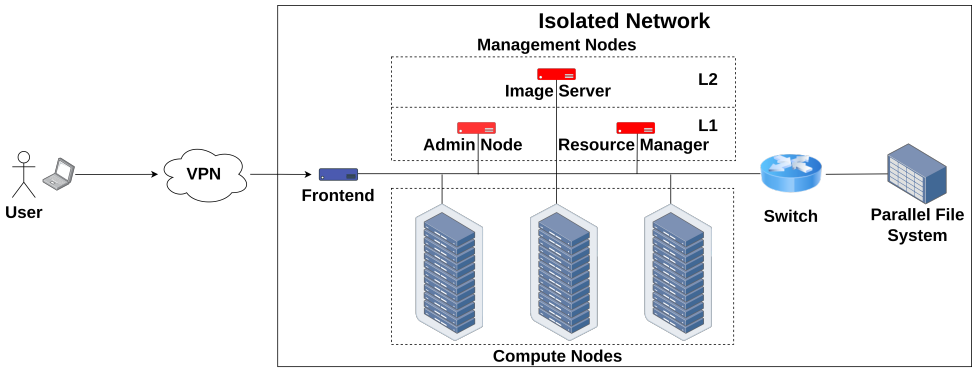


FIGURE 7.1: A simplified sketch of an HPC system.

on a frontend node. Frontend nodes are shared by all users and are used to build software, move data, or submit compute jobs to the batch system. Access to compute resources is granted by a resource manager, like Slurm, which schedules user jobs in such a way, that the general utilization of the system is maximized. The batch system dispatches jobs to the compute nodes. Although an interactive compute job is generally possible, the majority of the available compute time is consumed by non-interactive jobs, i.e. they run completely without any user interaction. The frontend as well as the compute nodes share at least one parallel file system, like *Lustre* [29] or *BeeGFS* [85].

The management nodes are comprised of several different nodes that are solely reserved for admins. Hence, they share the basic requirement, that they need to be protected from any user access. Typically, there is a specific admin node, which is used just for login. Very important is the so-called image server, i.e., the node which is used to provision the golden images to all the nodes, including the frontend and the compute nodes. If an attacker would gain access to this server, the images could be compromised and distributed to the nodes. In order to increase the security of this node, it is placed in the Level 2 security zone, where access is highly limited and requires further activation and authorization. In order to decrease the attack surface from the image server in the Level 2 security zone, compared to the admin nodes which are being operated in the Level 1 zones, the image server does not allow access from any other system located in a lower security level, i.e. the Level 1 admin nodes and the user nodes. In addition, there are no user-accessible daemons listening on any ports. Write access to the images is strictly limited to this image server. The read access for the compute nodes via the Preboot Execution Environment (PXE), necessary for the actual deployment of the images to the compute and frontend nodes, is only done by an admin node located in the Level 1 layer. In this way, a possible exploit in the cluster manager, i.e., the system responsible for the distribution of the golden images, cannot lead to a compromised image.

Another important node is the one, that the resource manager is running on. This server is responsible for enforcing the correct assignment and access of the jobs and users to the compute nodes. The last pieces of infrastructure are the networks that connect the different nodes.

The provisioning of software is usually done via an environment module system, like *Lmod* [123] or *Spack* [63], and is cluster-wide available, which allows replicating the desired working environment by loading the appropriate modules on any node of the cluster.

7.2.2 Typical User Workflow

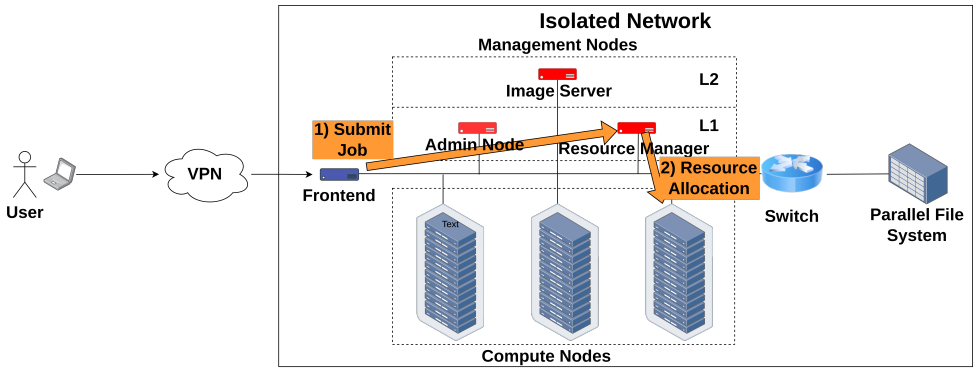


FIGURE 7.2: A schematic sketch of the typical workflow for users to submit a job on an HPC system.

The typical workflow to execute a job on an HPC system is depicted in Figure 7.2. A user logs in and can write or submit a batch script. This is typically similar to a shell script, where the desired resources and the commands to be executed are specified. The resource manager checks, whether or not the specified resources are eligible for the *uid* the request is coming from, i.e., if the user is authorized to use the specified resources. If the request is permissible, the resource manager schedules the job in an appropriate time slot for execution with the overarching goal of maximizing overall system utilization, although other strategies can also be specified. The input and output data on a parallel file system can be accessed from all nodes. The necessary communication between the storage nodes and the compute nodes or, in the case of multi-node jobs that are communicating via MPI, ideally takes place via a high-performance interconnect like *Omni-Path* [23] or *Infiniband* [159].

7.2.3 Possible Attack Scenarios

Based on the HPC system architecture and user workflow, potential security risks can be identified that are discussed in the following. It is assumed that a privilege escalation, i.e., a user gaining *root* privileges, can happen on any system accessible to users, in particular the frontend and the compute nodes. Because the nodes and storage systems users have direct access to are solely protected by the permission system of the Linux kernel, the trusted code base is very large and has a large attack surface which presumably yields unknown vulnerabilities that have been exploited in the past [38]. For the following security analysis, it is therefore assumed, that an attacker can gain *root* access or can impersonate an arbitrary user on one of these nodes.

Data Stored on a Shared File System

Starting from the node the attacker gained *root* privileges, *root* can get access to any file that is stored on one of this node or a shared file system mounted on this node. This direct access can be made a little bit more uncomfortable, for instance, by an NFS *root-squash* prohibiting direct root access. Such issues can be circumvented by an attacker by changing the *uid*. Therefore, all data should be considered as compromised.

Data Stored on a Compute Node

Additionally, after the job has started, the user is also able to log in on these nodes, for instance via SSH. The access is hereby granted by the resource manager solely based on the *uid*. Thus, a *root* user has immediate access to all nodes allocated to any user and therefore has access to the local data and processes. Furthermore, a *root* user can submit jobs to the batch system with an arbitrary *uid*, thus gaining access to compute nodes reserved by the resource manager for a specific user group.

Manipulation of the Provided Software

A malicious *root* user can also tamper with the provided software stack or with the individual system image of the node the attacker is currently on. Such a compromised system can then continuously leak data.

Manipulation of the high-speed interconnect

When the processing power for an application needs to be scaled up, the application's data is typically divided between multiple processor cores and nodes and is executed as parallel tasks, often called *ranks*. The most common way for these individual processes to communicate intra- and internode is the usage of MPI libraries which transfer information and data between the ranks. For small numbers of ranks, this can be on a single node, for larger numbers, multiple nodes are required. In these cases the capabilities of the interconnect become important if a lot of information needs to be exchanged between the different nodes. To get the best scalability, that is the efficiency with which additional cores deliver additional performance, high-speed interconnects like OPA or Infiniband are used in HPC systems to provide better latency, message rates, and bandwidth than a conventional Ethernet would.

The MPI data is injected into the high-speed interconnect unencrypted. Encrypting the MPI data, even with hardware, would add latency. Even a minor increase in latency can have a significant impact on scalability, and so encrypting MPI data is undesirable. In practice, the unencrypted MPI data consists of numerical data exchanged between the ranks of the job as it performs the calculations required by the application. As such, a bad actor managing to access this data is unlikely to be able to derive any value from it. Every node in the job knows the identities of the other nodes, so for a bad actor to partake in the MPI data exchanges, it would need to spoof the identity of a valid node. Therefore, the risk of data leakage, even in an unsecured network, is fairly low.

However, it is always possible that if an unsecured node hosting a bad actor can send data to a secured node, then it may be able to exploit some previously unknown weakness within the secured systems. In addition, these Network Interface Cards (NIC) allow for Remote Direct Memory Access (RDMA). This means another computer in the same network can access the memory of a node without the host being aware of it. Thus, this access cannot be prevented via a local firewall.

These high-speed interconnects are switched networks that rely on a *subnet manager*, also known as a *Fabric Manager (FM)*, for configuration, including the creation of the used routing tables. An attacker can try to imitate the FM on hijacked nodes and bombard the switches with malicious configurations. In addition, an attacker could try to spoof its source node and ingest packages in order to maliciously manipulate the execution of the job on the secure node.

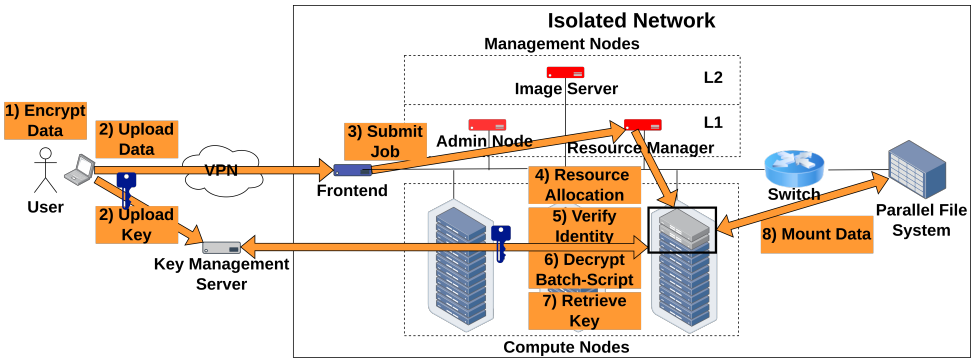


FIGURE 7.3: A schematic sketch of the secure workflow on an HPC system, which is divided into eight distinct steps.

7.3 General Design of the Secure Workflow

As stated before, all systems to which users have direct access could be considered to be compromised and insecure as unknowingly malicious software by a user may have gained administrator permissions. Also, an administrator should not be considered completely trustworthy and permissions should be limited as much as possible. In order to design a secure workflow, data and software need to be protected on such an exposed system and a mechanism is necessary to trust selected nodes, on which the actual computations can be securely done. Based on the discussed security problems identified in Section 7.2.3 a secure workflow is designed which mitigates these problems. This secure workflow is presented in the following.

7.3.1 Assumption

In order to provide trust in an otherwise untrusted system, this trust needs to be derived from a secure source system. Therefore, it will be assumed that i) the *image server* of the HPC system as well as ii) the local system of the user, for instance, the respective workstation or laptop, is secure. These assumptions are reasonable because on the one hand the *image server*, as shown in Figure 7.1, is located within the 2nd security level of a cybersecurity onion of the already highly guarded admin nodes, which deploy only limited services. On the other hand, the local system of the user is the system where the data resides unencrypted at the beginning of the workflow. Therefore, it has to be secure since otherwise the data would be leaked without any involvement of the secure workflow.

7.3.2 Overview

As discussed in Section 7.2.3 there are different attack scenarios that a secure HPC partition has to protect against. These scenarios can be divided into the protection of the data in transit, at rest, and during compute. In order to secure data during transit and at rest, encryption is typically used. To secure data during compute, one needs an ideally air-gaped, but at least reasonably isolated system or node. Based on these two simple ideas a generic secure workflow is developed as depicted in Figure 7.3.

Unlike the case of the typical submission workflow, presented in Figure 7.2, it is not possible to upload data, containers, and batch scripts directly into the shared file system, since this would leave the workflow vulnerable to attacks. The solution is

to encrypt everything using state-of-the-art encryption. Therefore, the first step is to encrypt the data as well as the container on the local system of the user. Afterward, in Step 2 the encrypted data is then uploaded onto the shared storage of the HPC system. The corresponding key is uploaded onto a key management system. This communication channel is completely independent of the HPC system and can therefore be considered out-of-band. Analogously should be proceeded with the containers.

In order to be able to retrieve the keys from the key management system, a valid access token needs to be provided in the batch script. To prevent this token from being leaked to an attacker on the frontend, the batch script needs to be encrypted as well. This can happen completely independent of the used resource manager by implementing this mechanism on the secure node, on which the job should run. For this, a public-private key pair is created on the system image of the secure node. The public key is then distributed to the users and can be used to encrypt the batch script while the private key has to be highly guarded. Since it is only available on the secure node and on the image server, this mechanism depends on the safety of the latter. This encrypted batch script can be submitted to the resource manager just like any other unencrypted script. For this, a corresponding `decrypt_and_execute` command needs to be implemented on the secure node, which takes as input the encrypted shell script.

The resource allocation made by the resource manager in Step 4 is only dependent on the used *uid* of the user on the HPC system. This is not secure enough, therefore the authenticity of the batch script needs to be checked. In this proposed reference workflow, this problem is solved by the user signing the batch script after it was encrypted with a private key. Here, the corresponding public key has to be made available to the secure node before a job can be submitted.

During Steps 5 and 6 the counterpart of the previously described Step 1 is done. This means, that first the signature of the batch script is checked to verify that this batch script was legitimately submitted. Here, the stored public key of the user who has made the request is used to match the signature. If that verification is successful, the batch script is decrypted, yielding a shell script that can be executed.

Within this script, an access token for the key management is provided. This token is used in Step 7 to retrieve the keys needed to decrypt the data and container.

In Step 8 the keys are used to decrypt the data and container image from the shared file system on the secure node. Now, the intended job of the user can be executed within the container. Using a container at this stage probably is the easiest way to maintain a heterogeneous software stack that is required to support the diverse processing steps. As mentioned, the additional advantage here is, that the container image itself can be encrypted and thus the integrity can be ensured, since tampering is only possible if the key is known. In addition, even in the case that some potentially sensitive information might be contained with this image it will not be leaked to an attacker. Furthermore, mounting all unsafe file systems per default read-only into the container prevents an accidental data leak, for instance by files that are temporarily written by the program without the knowledge of the user.

7.3.3 Securing an OPA Fabric

Broadly, network security of the high-speed interconnect is applied in two areas: (1) Partitioning the job's nodes from the other nodes in the cluster, and (2) Securing the FM from malicious interference.

Partitioning: This prevents packets from an unsecured node from reaching a secured node, and packets from a secure node from leaking to an unsecured node.

Omni-Path, in common with Infiniband, uses a system of Partition Keys (PKeys). PKeys are a mechanism used to support soft partitions, which enable the creation of multiple, overlapping communication domains. Using PKeys, secured nodes can be run in a separate partition from the other nodes of the HPC cluster. Within the configuration of the FM, the partitions that are required and the nodes that should be members of those partitions can be specified.

The FM sends the PKeys over the fabric to the switches and the adapters in the nodes. The FM updates the PKeys whenever there is a change in the FM's PKey configuration. If the switch receives a packet for a node that does not have the required PKey, it will block it. This is a very secure system, implemented in the hardware of the switches and adapters, and can only be controlled by the FM. The adapter is logically split into a fabric side and a PCIe side; the PKeys are managed from the fabric side, so there is no way that malicious code on the node can change the PKey configuration of the adapter. Therefore, by using PKeys, it is ensured that no unsecured node can send packets to a secured node.

Because a node can be a member of more than one partition, if it has been given more than one PKey, a user needs to specify which PKey to use when running a service that accesses the high-speed interconnect. For example, to launch an MPI job on PKey `0x0012`, the user sets the `PSM2_PKEY` environment variable to `0x0012`.

For the IP network interfaces typically used by storage systems, a network interface device is configured for each PKey that needs to be accessed. From within Linux, these devices behave in the same way as Ethernet devices and each will be configured on a different IP subnet. The network interface running with the default PKey appears as the familiar ib0, and an additional interface on PKey `0x0022` would appear as the device `ib0.8022`[5].

Securing the FM: The FM provides the PKeys to the switches and adapters in the fabric. Therefore, it is the core component that ensures the legitimate operation and isolation of the different networks. Thus, it is of utmost importance to prevent it from becoming compromised or that some malicious interference with the FM can take place.

It is possible that a bad actor on the bare metal of an unsecured node could bring up a competing FM. It would be difficult to do this with predictable results, but the activity could be disruptive. The FM's traffic runs in the Admin vFabric with its own PKey, and in a secure configuration, only the FM node(s), identified by the hardware GUID of its adapter card, are full members of this vFabric. Thus, any FM-like activity from a node that has not been authorized will be blocked.

Therefore, it is important that the FM node (and its standby) are secured, meaning that only the most privileged admin users are able to log in to these nodes.

7.3.4 Isolating Compute Nodes

In order to be able to provide secure compute capacity, an isolated node or subcluster is exclusively available to an individual or group for an arbitrary amount of time. These nodes have restrictive firewall settings and running administrative services are slimmed down. Therefore, no login capability, e.g., via SSH, is available. Furthermore, the nodes need to reliably boot a trustworthy operating system and must enforce that only authorized users can access the node. This is done in two steps:

[5] The 0x8000 bit is always set in the device name.

i) Users can only submit jobs that are running in a non-interactive mode, therefore they are required to submit a batch script to the resource manager, and ii) this batch script needs to be signed with a secret on the local system of the user. This secret is a private key, which must never become compromised. The matching public key is available on the secure node to verify the authenticity of the submitted job. Since this secret will never be uploaded to the HPC system, an attacker can't get access to it and can't submit jobs to a secure node from a false *uid*.

Software that is usually provided via environment modules needs to be installed in the system image of the secure node since using these shared modules would mean importing an untrusted codebase into the secure node. Alternatively, the filesystem where the software modules reside needs to be exported read-only to all exposed nodes, in particular, login and compute nodes, and only be writable from a specially secured admin node.

7.3.5 Key Management System

The usage of the Key Management System (KMS) should also follow certain best practices. Although the depicted security measures should be sufficient, it is favorable to use a one-time token mechanism to retrieve the keys from the KMS. If an attacker gets their hands on a token, with which the keys can be retrieved, the legitimate request from the user will fail, because the token was already used. Thus, it is immediately obvious that a security incident took place. The token should be short-lived as well, to limit the availability of the keys in the case that a job crashes before the token can be used.

Additionally, a reverse proxy is placed in front of the KMS. Here, the received API calls can be filtered based on the source Internet Protocol (IP) address and the used HTTP Verb. The goal is to allow an upload of keys from anywhere, however, limit the legitimate requests to retrieve keys via HTTP solely to the secure nodes.

7.4 Implementation of the Secure Workflow

The design and implementation are done with a minimal number of assumptions so that it can be considered as a blueprint for other systems with different requirements as well. Based on the general design presented in Section 7.3, implementation was done on an HPC system at GWDG.

7.4.1 Key Management System

Vault[6] is used for the KMS. It allows for the distribution of personal tokens to individual users. With those, users can generate tokens with limited permissions and a short, configurable lifetime. A *response wrapping* is used on these tokens in order to enable single-use tokens to access the deposited keys. In addition, the root token can be reliably deactivated, preventing the root user from spying on the user keys.
In front of *Vault NGINX*[7] with the `ngx_http_geo_module` is deployed as a reverse proxy. It performs IP-address filtering based on the HTTP verb in order to allow an upload of a key from external systems but restricts the request for the key retrieval to the known secure nodes.

[6] https://www.vaultproject.io/
[7] https://www.nginx.com

7.4.2 Data and Software Management

Since most HPC applications expect a POSIX-IO compatible file system, LUKS can be used to encrypt the data. These LUKS containers can be mounted, if the decryption key is available, thus providing the expected interface while transparently encrypting everything written to that mount. Encrypted containers are provided via Singularity. Similar to the LUKS data containers, these encrypted Singularity images are decrypted in kernel space as well. This means they reside decrypted in the RAM of the host, thus swapping needs to be deactivated on these secure nodes, to prevent that sensitive data is written unencrypted onto a non-volatile storage medium like a local SSD. By bind mounting only the LUKS data containers into the Singularity container, it is ensured that only encrypted write access is possible from the container onto the file system. Since the encrypted Singularity images are also used as a signature, i.e., used to ensure that the image has not been tampered with, the secured nodes need to be configured to only allow the execution of encrypted images.

7.4.3 Isolating a Secure OPA vFabric

The OPA FM uses a concept of VirtualFabrics (vFabrics) that can be used to create partitioned groups of nodes and/or apply a Quality of Service (QoS) to different classes of traffic. In order to isolate nodes and networks from each other, only partitioning is required. Here, vFabrics use the previously described PKeys as the underlying mechanism.

Nodes are assigned a membership of partitions within the configuration of the FM. Nodes can be Full Members or Limited Members. Full Members can communicate with all members, but Limited Members can only communicate with Full Members. This feature is useful when nodes in different partitions need access to the same shared resource. The following description is simplified to remove some of the site-specific configurations at GWDG, see Table 7.1.

- 3 `DeviceGroups` are created in the FM configuration: General, Secure, and Storage. These DeviceGroups hold the identities of the relevant nodes.
- 3 vFabric partitions are created: General, Secure, and Storage which contain the corresponding DeviceGroups as Full Members.
- Additionally, the General and Secure DeviceGroups are defined to be Limited Members of the Storage vFabric.

In this way, the nodes within the DeviceGroups General and Secure can both access the Storage nodes, but the nodes in General and Secure cannot access each other. The Storage nodes must be secured in a similar way as the FM node(s), as a bad actor on these nodes could get access to the nodes in the Secure vFabric.

The FM responsible for defining and enforcing these policies has to be located within the security onion of the admin nodes. Additionally, the FM needs to be configured to quarantine nodes if they try to spoof their identities, for instance in order to reach into a secure vFabric.

vFabric	Full Members	Limited Members
General	General	
Secure	Secure	
Storage	Storage	General, Secure

TABLE 7.1: Partitions and their Members.

Note: Currently, the OPA FM is not able to implement the Full/Limited functionality in the GWDG environment, and so a workaround is required using only Full

Membership. The workaround provides the same protections but with less flexibility and will be removed when the functionality is available.

7.4.4 Isolating a Secure Node

In order to isolate a secure node, the system image is hardened. To prevent an attacker from logging into that node, a restrictive firewall configuration is implemented using *iptables*. In addition, suitable services for accessing these nodes, like SSH, are turned off. For all services that need to be listening on a specific port, like the *Slurmd*, only the IP address of the known counterpart, like the node where the *Slurmctld* is running on, is reachable. In order to ensure these settings, a node needs to directly boot into these restrictive configurations and the image server, as well as the network which is used for the PXE boot, need to be trusted. Therefore it is mandatory, that an attacker can not reach the management nodes, and particularly not the level 2 layer of the employed security onion.

7.4.5 Submitting a Batch Job

In order to use encryption for the batch script, a 4096-bit *RSA* [166] key pair is created in the system image, and the public key is shared with the user. Since the scheduler might, like Slurm does, require a valid bash script to be submitted, a small workaround needs to be used. The actual command, one wants to execute, is encrypted in a gpg[8] message and passed to a helper function located on the secure node, yielding a valid and effectively encrypted bash script. Such a batch script is shown with a shortened PGP message in the following listing:

```
#!/bin/bash
/usr/bin/decrypt_and_execute <<EOF
-----BEGIN PGP MESSAGE-----
hQIMA8ErHWKpRkmoAQ/+LyPQJORoQwC5UjqNqXPcebqVYXfqgdt1rPo
[...]
=cRKs
-----END PGP MESSAGE-----
EOF
```

As previously discussed, the authenticity of the compute request needs to be ensured on the secure node. In order to do that, a detached signature of the batch script is created on the local system of the user and is uploaded into the same directory as the batch script on the HPC system. When submitting the batch script to Slurm via the `sbatch` command, the batch script is parsed and sent to the slurmctld. From there it is copied to a secure node using a communication channel between the slurmd on the secure node and the slurmctld. After this is done, the slurmd starts executing the job. However, before the just received and locally stored bash script is invoked, the *Slurmd Prolog* is executed. Since this is the first piece of code that gets triggered to be executed on the secure node, here the detached signature the user provided is compared to the local copy of the batch script. Only when this detached signature matches, the user-provided code starts to be executed. Upon failure, one can choose to either cancel the job or quarantine the node.

After the batch script is decrypted, the resulting bash-script is executed by the `decrypt_and_execute` method. Here, the provided token is used to get the keys from

[8] https://gnupg.org/

Vault. These keys are then briefly stored in a *tmpfs* to mount the LUKS data containers and to execute the Singularity container. Since only the LUKS data containers have a writable bind mount within the Singularity containers, results can only be stored there, thus enforcing compliance with data security regulations per design. After the job has finished or was killed by the resource manager, all mounted LUKS data containers are unmounted and the stored keys are deleted from the *tmpfs*. This behavior can be enforced within the *Slurm Epilog*. In the end, the user can download the LUKS output container, where the results are stored for further inspection.

Since the authentication of the user is done by the Slurmd within the Slurmd Prolog, one has to ensure that the execution of the Slurmd Prolog can not be circumvented. For admins, this is possible by using an interactive session without an allocation. This feature needs to be removed within the source code of the Slurmd.

7.5 Extension for Multi-Node Support

The described secure workflow is so far only intended for single-node jobs. When extending this approach to multi-node jobs, which run for instance via *MPI*, one is confronted with three distinct challenges. The first is to ensure secure communication between the nodes running the job. This is already achieved by the previously presented secure OPA configuration. The second challenge is the key distribution. Since there might be multi-node jobs where the number of nodes is not known before the job actually starts, or the orchestration across nodes is difficult and therefore error-prone, it is advantageous to keep the general idea of single-use tokens and distribute the keys after the successful retrieval. The third challenge is to provide suitable, encrypted parallel I/O.

7.5.1 Parallel Starter

In order to port the single-node approach to support multi-node applications, the decryption keys need to be distributed to all involved nodes in a secure manner. Since multi-node support requires a secure network to allow for safe MPI communication, one can reuse this network to safely distribute the keys to all nodes. For this, a *Parallel Starter* can be used, which is an MPI-capable program, where only the process with Rank 0 will access the single-use tokens to receive the keys from the key management system. Afterward, the process with Rank 0 uses a broadcast to distribute the keys to all other processes, where then each process needs to acquire a lock, like a node-local semaphore, for the particular node they are located on. This process is depicted in Figure 7.4. If multiple processes are running on a single node, the first process that gets the lock will write the file to the `/keys` path, which is a `tmpfs`, and initiates the mounting of the encrypted data on the node. The other processes on a shared node will skip this step. This ensures that on a single node that the keys are only written once to the `/keys` path and no race condition occurs. Since this is a standalone tool, it is completely independent of the used scheduler. Alternatively, when using Slurm, one can also simply use a `srun -procs-per-node=1`.

It is noteworthy, that there might be other, more specific methods like the `sbcast` tool from Slurm. Although this can be a full-fledged drop-in replacement, one needs to ensure that the communication only takes place inside the secured network. This is guaranteed if the node is properly isolated as described in Section 7.4.4.

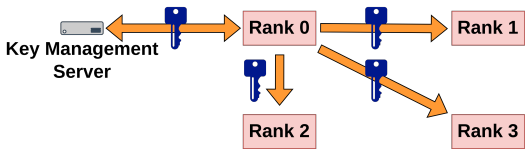


FIGURE 7.4: Schematic sketch of the parallel starter.

7.5.2 Creating and Managing a Secure Partition

A *Secure Partition* specifies an isolated sub-cluster within the used HPC system. This sub-cluster is also managed by the resource manager within a private partition which only predefined users can submit to. In addition, the nodes that comprise the secure partition can communicate with each other over a high-speed interconnect. This is securely possible since the secure nodes of a secure partition are all within a dedicated vFabric. This means, that no traffic is being routed from an unsecured node to a secure node, therefore enforcing network isolation of the secure partition on the switch level. Of course, all of the nodes in a secure partition are booted into the same secure system image. In summary, all changes necessary to create a secure partition can be done in software, rendering an additional procurement unnecessary. Since only software and configuration changes are necessary, switching nodes from insecure to secure operation can be done dynamically to match the provided resources to the actual demand. For this, a script can be executed on the admin node which automatically changes the configuration of the resource manager, the cluster manager, the fabric manager, and the reverse proxy settings on the key management system to adjust the IP filtering of the HTTP verb, and triggers a reboot of the corresponding machine to boot it into the secure/insecure system image. Due to this high degree of automation, the partitioning can be done quickly without much effort by the administrators. The required time to reconfigure a secure partition as described above is mainly determined by the time a node needs to reboot.

7.5.3 Parallel Input/Output

The concurrent write access to a LUKS container from multiple nodes is not possible by design and would corrupt the file system. Read-only mounts are supported as they prevent the modification of the file system, therefore, the input containers could be mounted on all nodes.

Assuming that the amount of data to be written by the application is unknown prior to execution but can be bounded, e.g. 10 GiB, a LUKS container with the maximum expected size can be created as a file with holes and formatted[9]. Thus, the actual occupied capacity of the LUKS container is small regardless of its size. For example, a 10 GiB container file formatted with ext4 requires 69 MiB of space and 85 MiB if created with LUKS and then formatted with ext4.

In the following, when usability is mentioned it refers to the overall complexity that is introduced. Although some tools can be provided that automate and simplify the handling for any of these cases, some difficulty would still remain for the application. In order to support parallel writes, there are various design alternatives with individual advantages:

[9]For example using the command `$ dd if=/dev/zero of=/tmp/test bs=1024 count=1 seek=$((1024*1024*10))`.

Single rank I/O with an encrypted LUKS container (SL): Only one rank in particular Rank 0 writes to a dedicated output container. *Pros:* the container could be automatically created and mounted; no change of applications with traditional (non-parallel I/O) is necessary; the output container can be fetched by the user and directly mounted. *Cons:* It does not allow output from multiple nodes, hence performance scalability is limited. This strategy is suitable for small volumes of output.

Parallel I/O to independent LUKS containers (PIL): Here, the ranks of each node write output to the node-local LUKS containers. *Pros:* it utilizes a parallel file system with the maximum performance and provides a node-local metadata cache. *Cons:* usability, the number of containers is dependent on the number of nodes - making it difficult to gather and aggregate the results for the user. This may require adjustment of the application and post-processing toolchain. This strategy is useful for cases with demanding I/O and a robust post-processing pipeline.

Parallel I/O to encrypted files with keys stored in a LUKS container (PFKL): Each process writes the data directly to the parallel file system but they encrypt each file (individually). Practically, at program runtime, the processes decide upon an encryption key and Rank 0 stores the key for this file in the initially prepared LUKS container where it can later be fetched by the user. A file might be created collectively (requires coordination of the key) or individually. *Pros:* individual keys for each file increase security. Compatible with independent or shared files. *Cons:* difficult to handle key retrieval in the post-processing and data analysis. Either the I/O path or the application must support encryption. This strategy is not advised due to the introduced complexity.

Parallel I/O to encrypted files with pre-shared key (PFSK): In this scenario, the user would embed a key into the application or retrieve it from *Vault* that then is used for the encryption of any created files. *Pros:* individual keys for each file increase security. Compatible with independent or shared files. *Cons:* the I/O path or the application must support encryption. As the key is known by the user a-priori, this strategy is superior to PFKL while integrating the advantages of the other methods.

Parallel I/O with an overlay file system with pre-shared key (POSK): In this scenario, a user would start an MPI job, where Rank 0 would fetch the key from Vault. This key is then distributed across all nodes and an encrypted mount is created on all compute nodes using a stacked cryptographic file system like eCryptFS, GoCryptFS, EncFS[10], or fscrypt[11]. *Pros:* easy way to provide a POSIX-compliant file system. No support by the application is needed. Minimal barrier for the user. *Cons:* user must be cautious to not write files unencrypted, i.e. to a wrong path. If a container is used, this can be handled similarly to LUKS containers by only bind-mounting the encrypted path into the container. Parallel I/O to a shared file is not possible. If multiple processes should write to a single file, strict file locking is required.

A qualitative comparison of the different methods is given in Table 7.2. Compatibility refers to the supported I/O modes in the application and the complexity of the overall setup and resulting artifacts. PFSK and POSK are currently the preferred

[10]https://github.com/vgough/encfs
[11]https://github.com/google/fscrypt

Method	Performance	Compatibility	Usability	Complexity
SL	-	-	+	+
PIL	++	-	-	-
PFKL	+	+	-	-
PFSK	++	+	+	0
POSK	+	++	++	+

TABLE 7.2: Comparison of different storage strategies. A ++ indicates the best solution within a category, a + a sufficient solution, a 0 indicates a neutral evaluation, whereas a - indicates an explicit lack.

solutions for multi-node support. However, for PFSK to achieve optimal compatibility the transparent encryption in the utilized I/O library is necessary. Therefore, PFSK can be superior for special applications that support this behavior but cannot be used as a default option. POSK is a good default option, although some performance penalties might occur. Files with holes are a problem for stacked cryptographic file systems, as a file that is sought beyond current EOF and written is completely filled.

7.6 Security Analysis

Based on the general design, presented in Section 7.3, and the actual implementation, presented in Section 7.4, a concluding assessment of possible attack scenarios along with their respective mitigation strategies is done in this section.

7.6.1 Man-in-the-Middle attack

A man-in-the-middle attack can happen in this secure workflow during the execution of Step 2, as shown in Figure 7.3. One can see, that on the one side, a man-in-the-middle attack can happen during the communication with Vault. This communication is done via the provided REST API and is secured via TLS. On the other side, an interception of packets can also happen during the upload of data to the HPC system. Here, data is secure since it is encrypted on the client side and the communication itself is guarded via SSH.

In both cases, the attacker would end up with state-of-the-art encrypted data, which can not be used without the corresponding decryption key. As presented, these are highly guarded and only retrievable for authorized users. Thus, access to the network infrastructure outside of the HPC system can not diminish the security of this workflow.

7.6.2 Privilege Escalation

A user only uploads encrypted data and encrypted Singularity container images, thus the attacker can neither gain access to the decrypted data nor can the software environment that accesses the data directly be compromised. The same argument holds for the submitted batch scripts. These are encrypted as well and thus ensure the confidentiality of the token of the KMS.

As discussed, a *root* user can submit jobs from the *uid* of a legitimate user. This can neither be prevented by the kernel nor by the resource manager relying on the kernel. The obvious mitigation would be a multi-factor authentication which is

prompted upon the submission of a batch script by a trusted management server. This, however, needs to be supported by the individual resource management software in use. A resource manager independent way is presented before, where the batch script needs to be signed.

To summarize, a *root* user can neither get access to the decrypted data, tamper with the software or system image, and can not impersonate a user on the system.

7.6.3 IP-Spoofing

In order to prevent an attacker from retrieving the keys stored in the KMS with a stolen token, *Nginx* is used as a reverse proxy in front of the KMS. Here, it filters out GET requests from an IP address, which is not a secure node. This is configured on the node of the KMS. To change that, access to this system is required, including access to the administrative network where the SSH port of the KMS node is available. An attacker can, however, use a false source IP address and mimic that the request is done from a secure node. Then, the KMS would send the requested keys but would do so to the specified secure node. Thus an attacker would still need to get access to such an isolated node.

7.6.4 User Operating Errors

Since the presented secure workflow has quite some steps that a user has to execute correctly to ensure the integrity of the processing, mistakes can happen and potentially impair the security measures. In order to simplify the application for a user, wrapper scripts are provided, which, for instance, automatically create and mount LUKS containers on the local system of a user while using strong random passwords. Furthermore, it is ensured, that the created keys are only uploaded to our *Vault* instance, and not accidentally on an untrusted system. Lastly, once a user has written locally a batch script that is ready for submission, a script can be used locally, to encrypt, sign, upload, and submit the batch script. That means, that a secure client only needs to be set up and configured correctly once, and can then be used safely by the users since all steps are executed automatically. This is still flexible enough to allow manual or automatic changes to the batch script a user wants to submit on a per-job basis.

7.6.5 Network Manipulations

Depending on the high-speed interconnect, there are additional threats associated. In Section 7.4.4 it was discussed that an OPA fabric can be securely locked down to ensure reliable operation even in the case of a privilege escalation on the connected, user-accessible nodes.

7.7 Performance Analysis

In order to determine the performance costs when switching from the unsecured workflow depicted in Section 7.2.2 to the secure workflow presented in Section 7.3 and Section 7.4, different benchmarks are done. These benchmarks can be roughly divided into two distinct groups. One type of benchmark is designed to quantify the static overhead associated with the secure workflow, while the other measures the dynamic cost of the used encryption. The performance measurements of the encryption are done once for a single node with LUKS and once with multiple nodes

Operation (unit)	Performance	
	Encrypted	Unencrypted
ior-easy-write [GiB/s]	0.6	2.8
mdtest-easy-write [kIOPS]	15.2	24.4
ior-hard-write [GiB/s]	0.06	0.01
mdtest-hard-write [kIOPS]	15.9	6.2
find [kIOPS]	270.8	211.8
ior-easy-read [GiB/s]	0.6	2.2
mdtest-easy-stat [kIOPS]	194.0	121.1
ior-hard-read [GiB/s]	0.3	0.4
mdtest-hard-stat [kIOPS]	69.6	44.4
mdtest-easy-delete [kIOPS]	22.6	33.4
mdtest-hard-read [kIOPS]	0.7	2.1
mdtest-hard-delete [kIOPS]	17.8	3.5

TABLE 7.3: *IO500* results on *BeeGFS*.

on an eCryptFS and GoCryptFS stacked cryptographic filesystem. Since the secure nodes are otherwise isolated, there are no additional costs during compute.

7.7.1 Measuring Encryption Costs

Encryption and decryption take place during write and read operations to a storage device, like a parallel file system. In order to simulate different I/O patterns to get a better understanding of the potential performance decrease, the *IO500* [103] benchmark is used. The encryption is done with *AES512* [132] in the case of LUKS and with AES-256 in the case of eCryptFS and GoCryptFS.

Performance Comparison on the Parallel File System on a Single Node with LUKS

As discussed in Section 7.3, the typical use case for the secure workflow is assumed to be that users upload their encrypted data onto the shared parallel file system and only decrypt them on the secure nodes. In order to measure the performance costs, two different scenarios are benchmarked. In the first case, an unsecured workflow is used, where an unencrypted Singularity container executes the before mentioned IO500 benchmark on a native bind mount on the parallel file system. In the second case an encrypted LUKS container, using *cryptsetup*, is mounted locally on the node with a loopback device. The latter case represents the secure workflow, therefore also an encrypted Singularity container is used to perform the IO500 benchmark. Both container images in these two benchmarks are created using the same *Recipe*-file.

The benchmarks are done on a dedicated node of the *Scientific Compute Cluster* hosted by GWDG. It features an Intel Xeon Platinum 9242 CPU with 376G of DDR4 memory operating at 2934 MT/s and runs on an *3.10.0-1160.36.2.el7.x86_64 Linux* Kernel. The used filesystem runs *BeeGFS* and has 4 metadata servers and 14 storage servers. The node is connected to the BeegFS storage via a 100 Gb/s *OPA* fabric.

Before the benchmarks is started, 343G of the 376G available memory is filled up and the swap was deactivated. The LUKS container is opened via *cryptsetup 2.3.3* and is mounted as an ext4 file system.

The results of the performed benchmarks are presented in Table 7.3. The first observation is, that the *ior-easy-write*, which is sensible to streaming performance, reaches in the encrypted case only $\approx$ 23% of the bandwidth of the unencrypted case. For the other operations, it is much harder to interpret these results correctly.

The reason is, that two concurrent effects which are close to impossible to disentangle have an influence on the performance. The first effect comes from the previously mentioned encryption which is done by `dm_crypt`, which therefore leads to a decreased read/write performance when being flushed out of the page cache. The second effect is, that the unencrypted IO500 run uses for its metadata operations the metadata servers of the BeeGFS cluster. The encrypted IO500 run, however, does (almost all) metadata operations on the local node, since it has a locally mounted ext4 filesystem. For the BeeGFS filesystem this LUKS container containing this ext4 filesystem is only one large file. The metadata operations, which are primarily benchmarked in the different `mdtest` runs, are therefore handled by the local node itself, not by the metadata servers from the BeeGFS filesystem. This problem becomes even greater when comparing the `ior-hard` runs since here 47008 Bytes are written/read/stat/deleted. During the `mdtest-easy` runs no data is written, and during the `mdtest-hard` runs only 3901 Bytes are written. Therefore, it is tricky to directly compare the metadata performance between those two runs.

In summary, one can observe a non-negligible performance degradation, particularly during streaming IO, when compared to the unencrypted measurement. This can be seen in the operations containing an *easy*.

Analysis of Cryptsetup

A recent analysis of the dm-crypt implementation found that the different work queues used to enable asynchronous processing of I/O requests can actually drastically slow done performance. To circumvent this problem, dm-crypt can be instructed to avoid a queuing of I/O requests and execute them synchronously. This feature was merged into the Linux kernel in version 5.9[12].

In order to further analyze the origin of the previously observed performance difference between the encrypted and the unencrypted use case, the kernel of the used node is updated to the most recent version 5.16.3, and *cryptsetup 2.4.3* is compiled from source. Since the clients for the parallel file systems of the *Scientific Compute Cluster* do not support newer kernel versions, the performance difference can only be measured on the node. For this, a tmpfs is used, which has the additional advantage of offering the lowest latency and highest bandwidth. This means, that any additional overhead can not be hidden by bottlenecks located on the storage device. The file for the *loopback device* has a size of 340G of the available 376G. In order to support the *vader BTL* of *OpenMPI* [62] additional 10G is provided in a tmpfs.

The results of these measurements can be seen in Table 7.4. In both cases, i.e., the encrypted and the unencrypted ones, an ext4 filesystem residing in a tmpfs is mounted via a loopback device. This means, that unlike in the case before, here also the performance of the metadata operations can be compared, since all deviations can be accounted to the overhead of *dm-crypt*. One can see, that there are only slight differences, however no clear trend can be recognized.

One important observation is that it can be confirmed that using encrypted Singularity containers does not have any measurable performance impact at runtime. The second observation is, that in this particular case one profits from an asynchronous execution of the encrypted I/O operations during parallelized execution with 10 processes. This can clearly be seen in the highlighted cells containing the results from the streaming intensive *ior-easy-write* and *ior-easy-read* operations as well as in the *ior-hard-read* test. The reason for this could be the use of loopback devices

[12] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=39d42fa96ba1b7d2544db3f8ed5da8fb0d5cb877

Operation [unit]	Synchronous				Asynchronous			
	10 Processes		1 Process		10 Processes		1 Process	
	Encrypted	Unencrypted	Encrypted	Unencrypted	Encrypted	Unencrypted	Encrypted	Unencrypted
ior-easy-write [GiB/s]	1.2	1.2	1.1	1.0	1.6	1.7	1.0	1.0
mdtest-easy-write [kIOPS]	111.7	123.3	70.0	70.6	111.4	111.5	69.0	69.0
ior-hard-write [GiB/s]	0.6	0.6	1.1	1.1	0.7	0.7	1.1	1.0
mdtest-hard-write [kIOPS]	18.8	19.8	31.6	33.9	15.5	15.3	31.1	35.6
find [kIOPS]	3821.4	3858.6	1493.4	1460.7	7093.1	5847.2	1456.0	1490.6
ior-easy-read [GiB/s]	1.0	1.1	1.0	1.0	2.1	1.8	1.3	1.3
mdtest-easy-stat [kIOPS]	558.9	537.9	183.5	180.0	566.7	567.1	179.1	181.6
ior-hard-read [GiB/s]	1.3	1.3	1.4	1.5	1.9	1.9	1.5	1.5
mdtest-hard-stat [kIOPS]	391.7	422.5	186.4	186.7	448.3	403.7	184.8	187.6
mdtest-easy-delete [kIOPS]	78.3	74.2	102.6	103.3	81.4	82.2	103.5	102.4
mdtest-hard-read [kIOPS]	188.0	180.7	48.5	49.2	213.8	209.3	45.7	46.2
mdtest-hard-delete [kIOPS]	64.9	62.4	75.3	79.4	63.2	60.7	78.6	73.7

TABLE 7.4: Results of the IO500 benchmark on an encrypted LUKS container residing in a tmpfs. The specification Encrypted and Unencrypted refers to the Singularity container.

Operation (unit)	Performance	
	1 Process	10 Processes
ior-easy-write [GiB/s]	0.9	2.1
mdtest-easy-write [kIOPS]	68.9	131.0
ior-hard-write [GiB/s]	0.8	0.8
mdtest-hard-write [kIOPS]	32.4	30.8
find [kIOPS]	1391.2	5832.1
ior-easy-read [GiB/s]	2.1	2.6
mdtest-easy-stat [kIOPS]	180.3	584.3
ior-hard-read [GiB/s]	2.6	2.0
mdtest-hard-stat [kIOPS]	173.8	380.4
mdtest-easy-delete [kIOPS]	98.7	72.4
mdtest-hard-read [kIOPS]	72.8	206.9
mdtest-hard-delete [kIOPS]	77.5	61.4

TABLE 7.5: *IO500* results on an *ext4* mounted loopback device residing in an *tmpfs*.

and device mappers, which causes differences in the execution of the I/O requests at the block device level when compared to a natively encrypted block device, like a hard drive or an SSD.

In order to estimate the actual encryption cost, a baseline for an unencrypted scenario is measured, wherein the exact similar setup the same file is mounted as an ext4 file system with a loopback device without the usage of cryptsetup. The results are shown in Table 7.5. By comparing the performance increase when scaling from 1 process to 10, there is still very limited scalability exhibited. The source for this issue is assumed to lie within the usage of a loopback device. Comparing the results of the *ior-easy-write*, where by far the most data is being written and therefore is mostly hit by the cryptographic overhead one can see that by comparing to the asynchronous test in Table 7.4 $\approx$ 80% performance is achieved. The achieved value of $\approx$ 2.1 GiB/s is very close to the value of $\approx$ 2.3 GiB/s one obtains when running the provided benchmark suite of cryptsetup.

In summary, one can clearly see a performance advantage of newer Linux kernels, however, it is not possible to replicate the advantage of synchronous cryptographic I/O execution on this system.

Operation (unit)	eCryptFS			GoCryptFS				Native			
Nodes, Procs. per Node	1,1	1,10	10,1	1,1	1,10	10,1	80,10	1,1	1,10	10,1	80,10
ior-easy-write [GiB/s]	0.4	1.5	3.4	0.2	1.6	1.6	6.0	1.4	3.8	3.8	9.2
mdtest-easy-write [kIOPS]	1.9	7.6	13.4	1.9	11.8	15.5	59.4	5	25.9	31.3	83.5
mdtest-hard-write [kIOPS]	1.7	2.7	6.7	1.05	2	5.7	7.8	2.4	5.1	7.5	12.0
find [kIOPS]	83.1	271.8	28.3	12.9	34.5	53.2	156.1	74.9	251.2	268.4	966.1
ior-easy-read [GiB/s]	0.4	1.5	2.5	0.4	1.4	2.5	3.2	0.5	2	2.9	8.7
mdtest-easy-stat [kIOPS]	20.5	98.3	4.6	10.6	30	84.1	374.1	15.4	124.4	137.8	392.7
mdtest-hard-stat [kIOPS]	20.6	56.8	2	9.1	9.3	76.4	75.1	17.9	55.5	112.8	138.6
mdtest-easy-delete [kIOPS]	5.1	8.8	1.9	3	17.3	16.4	53.1	7.1	31.1	32.3	57.5
mdtest-hard-read [kIOPS]	0.2	2	1.2	0.3	3.2	2.5	11.8	0.4	2.1	2.4	23.8
mdtest-hard-delete [kIOPS]	4.4	3.6	1.2	2.3	2.8	6	10.5	3.9	2.8	4.6	13.3

TABLE 7.6: *IO500* results of an eCryptFS layer on top of a BeeGFS cluster compared to a native BeeGFS mount.

Performance Comparison on the Parallel File System using multiple nodes with eCryptFS

In order to extend the secure workflow to multi-node support, and to offer alternatives to LUKS, also filesystem-level encryption is examined. For this eCryptFS is set up on 10 nodes, which are otherwise identical to the system described in Section 7.7.1. On these 10 nodes, the same directory on the shared parallel filesystems, i.e., the previously described BeeGFS cluster, is mounted using the same passphrase. The specified symmetric cipher is AES with a 32-byte key and activated filename encryption. Unlike the OpenPGP standard, eCryptfs allows for random access in a single file [76]. To this end, eCryptFS breaks a file into different distinct parts so-called *extents*, which have been encrypted with a cipher operating in block chaining mode. Therefore, any read or write operation only requires the entire extent to be decrypted, not the entire file. This however also entails, that parallel I/O of multiple processes from multiple nodes needs to respect this offset which is defined by the size of the extents. Unlike the case of an unencrypted file system, it is not enough to define a non-overlapping offset among all processes to ensure undisturbed file access, but this non-overlapping needs to be ensured on an extent level.

Therefore, the IO500 configuration needs to be adapted, to accommodate this issue. Here, all shared file I/O is disabled to ensure an undisturbed run. The results of these reduced runs are shown in Table 7.6. One can see nearly linear scalability in streaming operations, i.e., primarily ior-easy-write and ior-easy-read, when scaling from one node to ten. Please note, that in Table 7.6 the ior-easy-write for one process on one node is rounded up from 0.35 GiB/s.

When comparing the performance of the eCryptFS stacked filesystem against the native BeeGFS mount, one can see a performance drop of 75% when comparing the performance of the ior-easy-write for a single process on a single node. However, when scaling out to 10 processes on a single node, this difference is already reduced to a 60% performance drop. When using 1 process per node and 10 nodes the performance difference is reduced further to 8% performance penalty. These values are obtained by dividing the unrounded values from the eCryptFS rows with the matching ones from the native table. The rather constant performance difference in the *mdtest-easy-write*, where an empty file is created, shows the additional latency of going through the FUSE filesystem before actually triggering the underlying filesystem. This performance difference, due to the additional latency, shrinks again drastically in the *mdtest-hard-write* operation, where 3901 bytes are written to the file. Here, one also has an additional overhead due to the initialization vectors and the mismatch of the 4KB sized extents, which need to be fully processed for

the employed block cipher. Since there is a slightly higher performance drop when there are 10 processes on a single node, there might also be the effect of shared, and stacked caches, as described by [204]. Similar arguments hold for the reverse operation, i.e. the read.

Performance Comparison on the Parallel File System Using Multiple Nodes with GoCryptFS

GoCryptFS also provides file-based encryption based on a FUSE filesystem and is inspired by EncFS. It encrypts files using AES-256. Similar to eCryptFS, it also splits a file up into individual blocks of 4KiB in size and generates a 128-bit Initialization Vector. Additionally, filenames are also encrypted using AES-256. Analog to the previous analysis in Section 7.7.1, the IO500 benchmark is run with 1 process on a node, with 10 processes on a node, with 1 process, and 10 nodes, and with 10 processes on 80 nodes.

Comparing the performance of eCryptFS with GoCryptFS one can see a clear advantage of eCryptFS when comparing the scaling from one node to ten nodes. Here, the single process and the multi-node *ior-easy-write* performance is twice as large, while the saturation on a single node with multiple processes is similar. However, GoCryptFS exhibits still scalability when extending from 10 nodes to 80. Although it can not catch up to the performance of the native BeeGFS mount, the measured performance shows reasonable results with two-thirds of the native performance, while still having a more predictable behavior when scaling since it does not depend on the process distribution on the different nodes as much as eCryptFS does.

When comparing the metadata performance, e.g. *mdtest-easy-write* with the native BeeGFS client in Table 7.6 one can see again the performance drop caused by the additional latency of the FUSE filesystem which sits between the application and the actual filesystem and needs to process every request. In the case of the *mdtest-hard-write* operation, one can see a lower performance as the eCryptFS, hinting that the actual processing of the 3901 Bytes is less efficient. This matches with the observations from the *ior-easy-write*.

7.7.2 Measuring the Static Overhead

In order to determine the static overhead of this secure workflow, a node is booted into the secure image, and the workflow is executed 1000 times. The static overhead contains the verification of the signature, the consecutive decryption of the batch script, the retrieval of the keys from Vault, the mounting and umounting of the LUKS containers, decrypting and starting the Singularity container, and the deletion of the keys residing in memory. The reference job is executing `sleep 10` on bare metal as a baseline and within an encrypted singularity container for the secure workflow measurements. The complete wall clock time is measured with `time`. This job is submitted 1000 times with the normal workflow discussed in Section 7.2.2 and 1000 times with the secure workflow as implemented in Section 7.4. For each job, 3 keys have to be retrieved, one for the Singularity container, one for the LUKS container with the input data, and another LUKS container to store the output data. Both LUKS containers have a size of 20 GB. The batch script, which needs to be decrypted, has a size of 544 bytes unencrypted. The result of the benchmark is obtained by subtracting the average amount of the 1000 normal submissions from the individual wallclock time spent in the secure submission. The resulting distribution is shown in Figure 7.5. One can see that it follows a normal distribution with an expectation

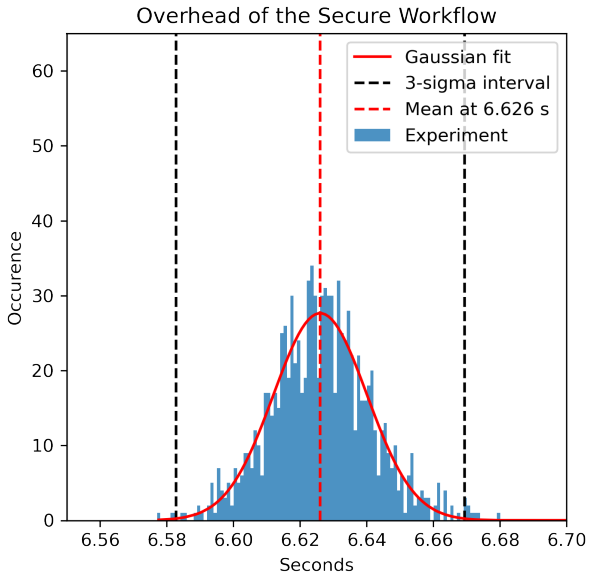


FIGURE 7.5: Distribution of the individual static overhead measurements of the secure workflow when compared to the same job executed with the normal workflow.

value at 6.63 s and a 3 sigma limit of $\pm$ 0.04 s. This overhead is negligible compared to a typical runtime of tens of minutes to several hours.

7.8 Use Cases

Within this section, three different use cases are presented. First, a detailed analysis of the presented secure workflow is done using a trained machine-learning model to analyze nocturnal polysomnography data. The remaining two use cases cover the discussed MRI project, where first a k-space reconstruction is down. Then the resulting artifacts, i.e., NIfTI files are used for further analysis. The latter is implemented using SecureHPC as a stand-alone service which is included in a clinical workflow.

7.8.1 Sleep Analysis

This section describes a use case, where the secure workflow is used to analyze nocturnal polysomnography data. The intent is to further illustrate a concrete setup of an end-to-end secure workflow, with a particular focus on the client machine, and to compare the performance of the secure workflow with a bare metal system on a real use case and hereby offer additional insight into its applicability compared to the synthetic IO500 benchmark. For this, a machine learning use case is chosen, since those are typically heavily relying on IO performance. This use case resembles the case discussed in Section 7.7.1, where in the naive setup without any optimizations a performance drop of $\approx$ 75% in streaming performance is observed. The measurements were done by Nicolai Spicher.

General Description

In order to compare the secure workflow introduced in Section 7.3 to an unsecured workflow as described in Section 7.2.2, the performance using a typical example from the life sciences is evaluated. This workload is provided by a sleep stage assessment which analyzes nocturnal Polysomnography (PSG) data. This data is acquired in sleep laboratories and is inspected usually by medical experts who divide the data into epochs of 30 seconds and assign a class to each epoch depending on the depth of sleep [18]. Due to the large amounts of data, this is a tedious and costly task, and – although the sleep classifications are intended to provide an objective assessment – there is still room for subjective interpretation [217]. This raised the interest in developing algorithms for faster and more objective classification.

PSG recordings typically include different biosignals such as EEG, Electrooculography (EOG), Electromyography (EMG), and Electrocardiography (ECG) which are measured continuously. These signals not only contain information about the patient's health status but can also be used to identify him/her [20, 205] which makes the protection of this data highly relevant.

For the performed experiment, the SIESTA project [99], which includes recordings of 98 patients with sleep disorders (e.g. sleep apnea, periodic limb movement syndrome) and 194 healthy controls, is used. The data was recorded in different European sleep laboratories and is stored in European Data Format (EDF) with a total file size of 104 Gigabytes. From this dataset, 100 EDF files are randomly picked for analysis.

Sleep stages classification is performed using the pre-trained "Stanford Stages" algorithm [187] which consists of three independent convolution neural networks for EEG, EOG, and EMG data which are jointly fed into a long short-term memory network. The final output layer assigns the labels "wake", "Stage 1 sleep", "Stage 2 sleep", "Stage 3/4 sleep", "Rapid Eye Movement (REM) sleep", or "Unscored" to 15-second intervals. This algorithm is pre-trained on multiple thousands of PSGs stemming from 12 different sleep centers in diverse recording environments and following different protocols.

The source code of the master branch[13] is used, `conda` for installing dependencies as described in the README file is applied, and a custom JSON file containing the configuration is defined. The algorithm processes each EDF file separately and generates a visual output called a "hypnodensity plot" showing time on the x- and probability of each class on the y-axis. Fig. 7.6 shows this visualization exemplary for one night (7.5h) of a single patient. The y-axis shows the probability distribution per sleep stage and thereby allows to assess the certainty of the algorithm: Columns with constant colors (stages) depict time ranges where the algorithm has a high certainty and the more colors (stages) there are in a column, the lower the certainty. A higher level of uncertainty can typically be observed in the transition between stages and is also present when medical experts annotate data. For the data depicted in Fig. 7.6, typical patterns of sleep become visible with phases of deep sleep in the first half of the night, more often REM sleep in the second half of the night, and phases of brief awakenings in the early morning.

[13] `https://github.com/Stanford-STAGES/stanford-stages`

Setup of the Secure Workflow

For illustration purposes and reproducibility, the code of this use case is provided on GitHub [14]. As shown in Figure 7.3, the starting point of the secure workflow is a safe client machine. In this particular setup a Virtual Machine (VM) in an OpenStack environment [176] is used to simulate this client machine. Once the described input data is copied into a LUKS container using a provided helper script, this LUKS container can be uploaded to a specific path in the user's scratch space. An additional empty LUKS container for the output data is created and uploaded. The two keys remain securely on the local VM. A completely analogous procedure is followed with the encrypted Singularity container, which is built once and is then uploaded into the user's home.
The remainder of the secure workflow is processed by a fully automated script. The job-specific commands, i.e. the generalized batch script that should be executed on the HPC systems, are written into a single file called `command.sh.template`. After this job-specific file has been written, the automated secure workflow script can be executed. Here, `command.sh` is created as a copy of the template. Then the keys are uploaded and the necessary single-use tokens are generated in Vault, which is then automatically inserted into the `command.sh` using `sed`. Afterward, the `command.sh` script is encrypted and copied into a `run.sh` as shown in the listing in Section 7.4.5. Lastly, using `gpg -detach-sign` a detached signature is created and stored as `run.sh.sig`. Both files, `run.sh` and `run.sh.sig` are copied to the HPC system into the same folder, which must be readable for `root`. Since Slurm is used in this particular use case, `sbatch` is used to submit the job to a *secure* partition. After the job is finished, the output LUKS container is downloaded to the secure client machine, in this case, the beforementioned VM, using `scp`, and the LUKS container is mounted locally to get access to the data.

It should be mentioned, that the creation of the input data LUKS container is not part of this fully automated process, since the job is run on the same input data multiple times. Generally, the creation and uploading of the input data container can be part of this automated script to offer a complete end-to-end secure processing pipeline.

Results

Fig. 7.7 shows a comparison of run times when using an unsecured workflow (max: 302s, min: 208s, median:213.5) vs. the secure workflow (max: 338s, min: 212s, median:220). The runtime is measured using differences in time stamps of the hypnodensity plots. As can be seen, results are bimodal with two clusters with the majority of run times being in the first cluster within the interval of 208s-250s. The second cluster begins at around 300s and represents larger EDF files due to longer night sleep. In 79% of the times, the secure workflow has a longer runtime, in 4% it is equal, and in 17% the secure workflow is faster than the insecure one. This is a clear indication, that the runtime of at least 40% of all runs is so close to each other that statistical fluctuations of the system performance are the dominating factor, not the overhead of the secure workflow. When comparing the medians one can see that the secure workflow takes $\approx$ 3% longer than the insecure workflow.

Considering that the secure workflow enables the processing of sensitive data and therefore creates the opportunity to reuse a shared and existing HPC cluster

[14] `https://github.com/gwdg/secure-hpc`

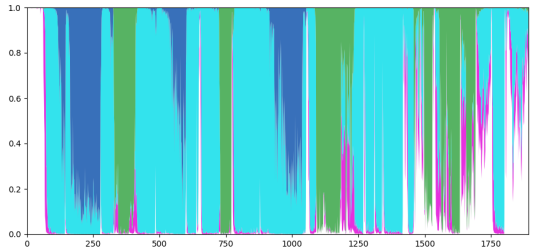


FIGURE 7.6: Resulting hypnodensity plot for a single night with values on the x-axis representing 15s windows. Stages are ”wake” (white), ”Stage 1 sleep” (pink), ”Stage 2 sleep” (turquoise), ”Stage 3/4 sleep” (blue), ”REM sleep” (green).

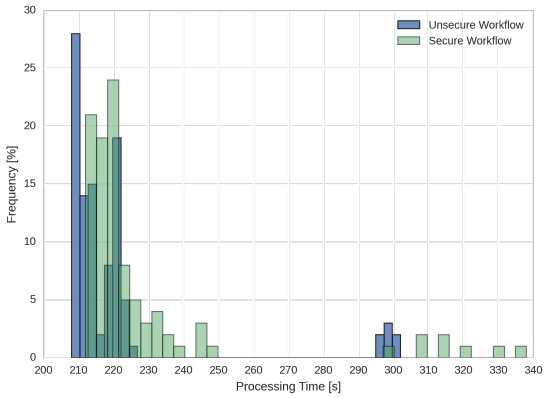


FIGURE 7.7: Distribution of the running times when processing single nights for sleep stage classification using the secure workflow compared to the insecure workflow.

for analysis of patient data which, due to privacy constraints, would be impossible otherwise, a 3% performance drop seems fair.

7.8.2 K-Space Reconstruction

Although confidential computing is not explicitly required for the k-space data used within the "Big Data MRI"-project, an example reconstruction using BART is created by Vitali Telezki to serve for a tutorial[15]. Unlike the previous data lake example presented in Section 6.4.4, where BART is dynamically developed and can therefore be built for each submission, this can not be done in this scenario. That is because the bind-mount would have an exposed binary which an attacker could tamper with. Therefore, all dependencies must be included within the container image. An example of such a recipe is shown in Figure 7.8. In order to trigger the reconstruction of Shepp Logan using a container image derived from this recipe, it expects to get passed a path as a command line argument, where a folder called `shepp_logan` is located. It then performs a uniform fast Fourier transform on the provided input data and saves the result in `output_fft/`. Afterward, the LUKS container containing the reconstruction can be downloaded and locally decrypted.

This simple example can be extended to enable more functionalities, for instance, the parallel image reconstruction using the `bart pics` command, as described in Section 6.4.4. This might require, that the corresponding git repository gets cloned and to build one specific branch within the `post` section in Figure 7.8.

[15]https://github.com/gwdg/secure-hpc/blob/master/tutorial

```
Bootstrap: docker
From: debian:testing
Stage: spython-base

%post
apt-get update
apt-get -yy install bart

%runscript
bart fft -u -i 3 $*/shepp_logan $*/output_fft
```

FIGURE 7.8: Excerpt from the used Singularity recipe to perform an inverse fast Fourier transform.

7.8.3 Volumetric Analysis of MR Images

Once a k-space image is reconstructed and formatted to the NIfTI format, further processing can take place. For brain scans one of these tasks is a volumetric analysis and surface-based thickness analysis. These can be done with *FastSurfer* [84], a deep-learning-based neuroimaging pipeline that is fully compatible with *FreeSurfer*[16]. It allows to run segmentations and volumetric calculations of the brain and can then reconstruct the cortical surfaces and map cortical labels. Due to the deep-learning approach, the segmentation can run on an accelerator, like a GPU. However, the following surface reconstruction requires a Central Processing Unit (CPU). The result of this pipeline can be seen in Figure 7.9.

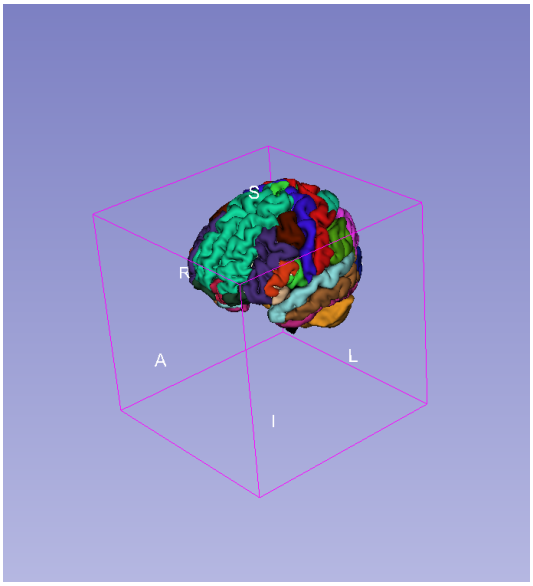


FIGURE 7.9: Segmented and reconstructed brain scan after the FreeSurfer-based processing is done.

This pipeline can not only be run within the context of the discussed "Big Data MRI"-project as part of the overarching data lake but the novel SecureHPC service can also be used as a stand-alone service to serve clinical workflows, as is demonstrated in Figure 7.10 for the discussed volumetric analysis in a real use case.

If a patient signs the consent form, agreeing that his/her data is processed at GWDG, a node within the *PATLAN* is triggering the automatic segmentation. For this, the MRI console sends the T1-weighted MRI scans to the Picture Archiving and Communication System (PACS) and additionally to the secure client. There, the images are formatted from the Digital Imaging and Communications in Medicine (DICOM) file format to NIfTI. The NIfTI file is then put into an encrypted LUKS container, which is uploaded to the HPC system. The key from the LUKS container is put into the KMS, and the short-lived, single-use token is put into the batch script, which is then encrypted, uploaded to the HPC system, and then submitted. Since the same job is running on different input data, the corresponding Singularity container image is residing on the HPC system the entire time and is re-used across all jobs. The key for the container image is also

[16] https://surfer.nmr.mgh.harvard.edu/fswiki/FreeSurferMethodsCitation

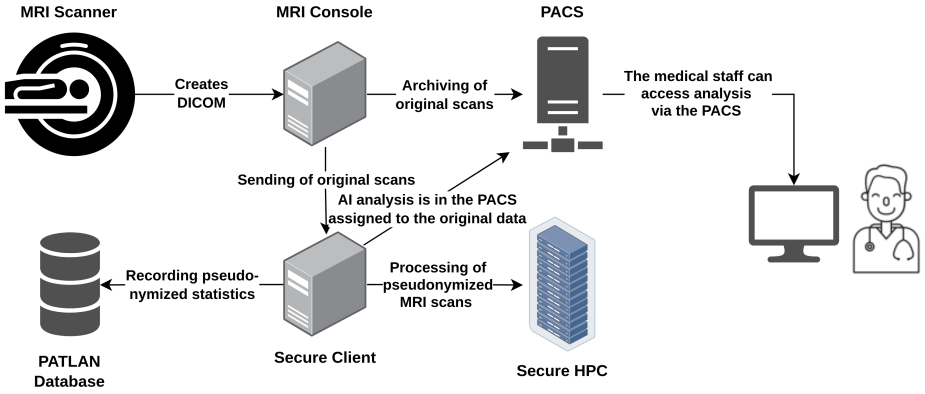


FIGURE 7.10: Complete clinical workflow from the MRI scanner to the doctors.

stored on the secure node and needs to be uploaded to the KMS for each unique job execution. Within a dedicated database, personalized statistics are recorded to build a large dataset over time. This has to be done using pseudonymization instead of anonymization to ensure that individual patients can be deleted upon revocation of their consent according to Art. 7(3) GDPR.

To get the most benefit from this workflow, a doctor wants to have the result of this pipeline, compare Figure 7.9, at the same time s/he looks at the scanner reconstructions, i.e., the final MR images, for diagnosing. In order to fulfill this time criticality two things are necessary. First, the scanning protocols need to be adapted to make the necessary T1-weighted measurements at the beginning, and the complete measurements have to be immediately sent to the Secure Client by the MRI Console. By linking the analysis to the original data within the PACS, doctors can access the segmentations similar to the associated scans. Second, the computation has to be done within a maximum of 20 minutes, ideally less, in order to be still relevant for the diagnosing doctor. To satisfy this strict time limit it is advantageous that the presented secure workflow enforces an isolation on a per node granularity within the kernel space. This allows to transparently use accelerators like GPUs to drastically speed up tasks like inferences of neural networks. This feature would not been possible by other isolation methods, like the discussed TEEs, since these do not support peripherals.

7.8.4 Evaluation

The presented workflow to use the novel SecureHPC service is extensively tested using synthetic and real use cases to demonstrate the general applicability. Due to the modular design of the data lake, the processing of sensitive data can be treated as an isolated and independent service. This successfully maximizes the overall impact of this research, since it can not only be used for the data lake within the driving "Big Data MRI"-project but it can be deployed outside this scope even within the intranet of a hospital as part of an overarching clinical workflow serving real patients. The additional cost due to the encryption is determined to be minimal in practice while showing a significant impact on synthetic workloads. This performance cost reached up to three-quarters of the entire computing time, compared to an unencrypted case. On a realistic use with a more balanced workload, this is diminished to only three percent. However, in practice, a more fundamental drawback is the current coarse-grained isolation. This is done on a per-server level to provide full HPC capabilities,

including secure access to GPUs. Suppose a single user, or institute sharing a secure partition, cannot fully exploit all the resources of the provided partition. In that case, these resources are currently unused, since no further sharing across the fixed organizational boundaries is possible.

7.9 Summary

Within this chapter, a secure partition on a shared HPC system is developed that can be accessed using a specific secure workflow. This partition provides the necessary isolation required to process sensitive health data. It utilizes non-interactive batch jobs, which are fully defined in advance within a batch script. These batch scripts are written and fully encrypted on the client side. Therefore, these secure HPC jobs represent fully encapsulated functions. These functions can be invoked either by using SSH connections or by using HPCSerA, proposed in Chapter 5. Thus, the interaction paradigm of the developed SecureHPC service integrates seamlessly with the previously utilized FaaS interfaces. Therefore, the proposed SecureHPC service can be considered within the context of the data lake as another compute backend, next to the normal HPC and cloud systems, which can be accessed using the discussed FaaS interface. However, the proposed SecureHPC service can also be used completely independent of the overarching data lake architecture within use cases solely requiring scalable and trusted execution environments.

Chapter 8

Secure Metadata Management

Within this chapter, two different methods for secure and scalable metadata handling on HPC systems are designed. One extends the in Chapter 7 presented SecureHPC service, while the other is a completely independent method. First, the background and related work is discussed in Section 8.1 followed by the design of a cluster spawner, a throughput benchmarker, and an attribute-based encryption method in Section 8.2. The experimental setup is described in Section 8.3 followed by a discussion of the results in Section 8.4. Based on these measurements a concluding evaluation is done in Section 8.5. Parts of this chapter are accepted for publication in [143] and are submitted to [142].

The data lake, that is presented in Chapter 6, uses a modular design with three main pillars, i.e., storage, compute, and metadata handling, which are unified by the abstraction of (F)DOs. Due to this modular design, it is possible to build a secure data lake for sensitive data by securing each individual component. While storage and compute are secured in the previous chapters Chapter 6 and Chapter 7, the last missing service is the overarching data catalog where all metadata are indexed. Such a secure data catalog service can then be referenced within the (F)DO record, pointing to the location of the metadata, and providing machine-actionable instructions on how to retrieve them. Therefore, this problem is again well encapsulated since a well-defined interface can be provided to integrate this service into the overarching data lake. In order to maximize the overall impact, this well-encapsulated problem should be tried to be generalized, to provide a stand-alone service, that can also be used outside the scope of the data lake. During the implementation of the data lake presented in Chapter 6 Elasticsearch (ES) is used to provide the indexing of the metadata of the DOs. Therefore, ES is also used here as well to better facilitate the required changes when going from a normal to a secure deployment. For this transition, two completely different workflows are proposed. The first utilizes the in Chapter 7 presented SecureHPC service where the deployment of on-demand ES clusters and the consecutive execution of data ingests or queries within usual HPC batch jobs are considered to be just one very specific kind of workload that can be executed on this generic platform. Based on the experiences summarized in Chapter 7, only GoCryptFS is used for filesystem encryption, since it exhibits the best scalability and has a dynamic storage size since it is a stacked cryptographic filesystem. This is in contrast to the presented LUKS containers, where the maximal size of the filesystem needs to be known beforehand. The second proposed method is completely independent of SecureHPC and solely relies on client-side encryption and does therefore not require a trusted execution environment on the server side.

8.1 Background and Related Work

Historically, semantic metadata is often encoded by researchers within filepaths and filenames. This practice has the disadvantage, that globbing has a much worse performance compared to indexing due to its recursive nature, which does not scale. In addition, higher-order logic like aggregations always have to be implemented by hand and therefore exhibit similar scalability problems. Thus, this approach is slow and work-intensive. Therefore, tagging and indexing data on HPC storage systems has become increasingly popular. Examples of this are VAST Data, which offers a tagging mechanism based on an SQL engine, or the Ngenea Hub which uses ES on top of a GPFS to provide unstructured data management. These systems generally share the characteristic that they rely on a fixed number of dedicated servers to support their query engine. This has the disadvantage, that these servers are idle, if they are not sufficiently utilized, or if the number of provided servers is too small to handle the incoming requests. Therefore, it seems more appropriate to use the available nodes of an HPC system to dynamically scale up or down on-demand ES clusters, reuse existing resources, and offer dynamically and use-case optimized scaled clusters. For this, a workflow is required to spawn on-demand ES clusters in user space using standard Slurm job allocations.

To enable the usage of sensitive data within such a workflow additional security measures have to be taken, similar to Chapter 7. For this two approaches will be developed. First, the SecureHPC partition is used to run this workflow, thereby protecting the data in the same way any other workflow is protected using SecureHPC. A different approach for encrypting information in a NoSQL database to ensure data privacy even in remote environments like a public cloud is to encrypt the data on the client side. One example of this is *SecureNoSQL* [2], where a trusted Proxy is used to encrypt the JSON-based documents before ingesting them into a NoSQL database in an untrusted environment. On this proxy, a security plan defines the security policies like the cryptographic modules. It uses Order-Preserving Symmetric Encryption (OPE), a deterministic encryption method that maps integers from a range of [1,M] to a range of [1,N] [26, 97]. If this mapping is uniform it maximizes the Shannon entropy, while the total value of it depends on the value of N-M. The larger this value, the higher the Shannon Entropy and the more secure this encryption scheme is. The advantage of OPE against other encryption methods is that it still enables efficient range queries on the encrypted data. A different approach is demonstrated in [39] where an index in a document-based NoSQL database is constructed using an *Adelson-Velsky Landis* tree [1] which is encrypted using an Order Revealing Encryption (ORE) [109], a generalized formulation of OPE. Since no extensive changes on the server side are desired, a similar approach to *SecureNoSQL* is designed, which works completely on the client side, i.e., without the proxy for easier integration into HPC workflows.

In order to determine the applicability of such a workflow, once designed and developed it needs to be benchmarked. The canonical tool for ES benchmarking is Rally[1], the official macrobenchmarking tool from *Elastic*. It runs distributed benchmarks with standard and user-defined datasets. Different benchmarking scenarios are defined as so-called *tracks*. Every track contains one or more *corpora*, which contain all JSON documents to be ingested into the according indices. Furthermore, every track contains many *operations* such as the ingestion or specific queries, which are then structured into a *challenge's schedule* in a fork-join model. This benchmark

[1] https://github.com/elastic/rally

is sensitive to latency increases, therefore limiting its usefulness in high-load and high-throughput workloads.

8.2 Design

In this section, the design of the proposed on-demand cluster spawner, the throughput benchmarker, and the Attribute-Based Encryption (ABE) are presented.

8.2.1 On-Demand Cluster Spawner

The on-demand cluster spawner uses MPI to automatically discover the available hardware nodes of an HPC job, by using the pre-configured MPI environment of a batch job. This means that no previous configuration by the users is required, like the hostnames or IPs of the involved nodes. This dynamic host configuration is also capable of managing changing hostnames from one batch job to another. This also implies that the cluster can be dynamically sized, basically supporting any cluster size. To increase portability and reproducibility, ES is packaged into a controlled Singularity container, which is fully node-transparent and runs using the `--cleanenv` option. Additionally, it supports statefulness between jobs, which removes the need for reingesting on every job spawn. The container is based on the Ubuntu 22.04 official docker image. ES itself can be optionally included in the image, requiring that each user builds the container image themselves to ensure the correct uid/gid settings. Alternatively, it can be bind-mounted at runtime. For this, a `tar.gz` is shipped with the spawner from which ES is extracted, and the uid/gid are set to the correct user. In this way, ES can be deployed fully in userspace without any need to reconfigure anything on the HPC-server side. To achieve statefulness the data path where the ingested data, i.e., indexes are stored is bind-mounted. In addition, by bind-mounting also the configuration and logging paths, the dynamic movement between different nodes, IPs, and even network interfaces, is achieved. Since the hostnames and IPs are not persistent from run to run, these are abstracted by the MPI ranks of the spawner. For this, using an `MPI_GATHER`, all hostnames and their ranks are sent to the root rank. Then, the root creates or updates the ES configs while the other nodes are waiting at a barrier. The MPI root then updates the ES configs for all nodes with their new IP addresses. This does not affect the mapping of nodes, i.e., the mapping of MPI ranks to the individual indexes stored on the shared filesystem. The first ranks are master-eligible nodes, from which only one node can be the active master. All nodes are active data nodes. Due to the abstraction of the hostnames by the MPI rank, any arbitrary permutation of hostnames will work, without the users realizing it. After creating the configs, each host starts the Singularity container with its config and the previously created ES configs bind-mounted in. The only change that is required on the nodes of Emmy is a higher-than-default number of memory-mapped areas per process, i.e., `vm.max_map_count` of 262144, where the default value is 65530. This is needed because ES uses a `mmapfs` directory to store its indices.

8.2.2 Distributed Thoughput Benchmarker

The proposed benchmarker is currently split into two tools. A write-focused benchmark measuring the ingestion throughput of a new index as well as a read-focused benchmark measuring the query performance of a previously ingested index. The benchmarker itself is inspired by Rally and is in many parts compatible with it.

Therefore, it can even be used alongside Rally to provide a more comprehensive picture.

Ingestion Benchmarker

The ingestion benchmarker measures the ingestion throughput into a new index for each configuration. It uses the Newline Delimited JSON (NDJSON) format which allows for trivial dataset creation and scaling. Furthermore, it allows for the usage of all Rally corpora which facilitates the aforementioned interoperability. Using MPI, the ingestion benchmarker can be arbitrarily scaled across nodes.

Before the ingestion can be measured, the root node creates the index. For this a configuration file containing all field mappings can be provided, i.e., each type of each attribute of the dataset is strictly defined at index creation time. Although ES allows for dynamic schemas, the benchmarker uses strict type mappings for reproducibility. The type definitions use the same ES Domain Specific Language (DSL) as Rally. The ES index is by default configured with one shard per node but can be overwritten using a command line argument. After index creation, the offsets of the globally shared NDJSON file containing the dataset are calculated and shared across all processes which use offset *lseek* to start at the specified place for each rank. Then, all worker nodes ingest the data using the bulk API in parallel. For this, the number of documents per request can be freely configured as a command line argument. All worker processes are simultaneously starting to ingest data and are then independently recording the individual timings of each single request. In the end, all measurements are gathered on the root rank and are streamed into a JSON file for further analysis. Lastly, in order to isolate the ES performance, caching is explicitly disabled for the index.

Query Benchmarker

The query benchmarker is also MPI-based and measures the documents per second as well as the latency of each individual request. Inspired by Rally, the benchmarks are structured into multiple steps using a fork-join model. The query benchmarker has a custom, JSON-based DSL for the benchmark design. The DSL embeds the ES query language internally, allowing for easy development for ES users as well as trivial portability to Rally tracks. Each request explicitly bypasses the cache, resulting in more realistic measurements. Lastly, it supports multiple alternating queries in a single fork-join task to create a mixed usage load.

For each of the fully disjunct benchmark steps the following workflow is executed: First, the root node waits until the ES cluster's health is green, to ensure the ES cluster has fully formed. If multiple queries are defined for a given task, each rank chooses a random permutation. This is done to maximize the randomness of the request pattern sent to the ES cluster. After that, if configured in the benchmark definition, warmup requests are made to fill the page caches and optimize the Java Virtual Machine (JVM) Just-In-Time (JIT) compiler. If this is not done, the query performance will increase over time, overshadowing other effects. Once the warmup is done, each worker starts the benchmark until the configured execution time is reached. The cache is explicitly disabled using the `request_cache` parameter. Per default *Session Objects* are used to facilitate persistent connections and reuse the underlying TCP connections. Thereby, additional HTTP overhead is limited which could otherwise taint the measured data, since handshaking and authentication are embarrassingly parallel tasks. If a request is successful, the latency and number of

received documents are saved, otherwise the error HTTP code is recorded. A sleep between each request can be optionally configured. Lastly, all measurements get aggregated in the root rank and dumped into a JSON file.

8.2.3 Attribute-Based Encryption

GocryptFS has two limitations. First, it does need a secure environment to be executed in, i.e., it can't be deployed in any managed hosting. In addition, in order to spawn a new ES instance, the entire storage path has to be encryptable by the user. This means a user has either full access or none at all. Both of these limitations can be mitigated using an ABE scheme. Here, both the documents to ingest, but also the queries later on will be encrypted. This allows hosting an ES cluster in an untrusted environment and also enables to use of different keys, not only on a per document basis but also on a per attribute basis, offering a very fine-grained access control. However, performing operations on encrypted data might not be possible. For instance, a range query implies that the encrypted numeric values maintain their original ordering. To enable specific numeric operations on ABE documents, numeric data are encrypted using an OPE method based on[26]. The proposed ABE algorithm is based on the AES256 and OPE algorithms. While AES256 is used for non-ordered string values OPE is used for ordered data types such as integers or floats to preserve any ordering feature, like a range query, or an *after* or *before* query on a date. Since OPE is only defined on integers, further data type mappings are required. Thus, the following data type case distinction is done:

- **Strings:** Strings and attribute keys are encrypted using AES256.
- **Integer**: Integers are mapped from an excepted input range to $[0, 2^{64})$ using OPE.
- **Dates:** The date `YYYY-MM-DD` gets interpreted as the integer `YYYYMMDD` which will get encrypted as an integer using the input range $(0, 30000000)$.
- **Date-Time:** The time `YYYY-MM-DD hh:mm:ss` works analogously to the date datatype with the input range of $(0, 30000000000000)$.
- **Float:** Float to integer conversion inherently introduces loss of precision. Furthermore, in order to provide cryptographically stronger encryptions, the input range should be chosen as small as possible. This provides a tradeoff that should be handled on a case-by-case basis by letting the user specify the required precision. Therefore, the proposed algorithm takes not only the expected input range $[l, r]$ but also the step size s that describes the mapping's granularity by partitioning the input range. Starting at l, the mapping returns the number of steps required until it is equal to the float. Thus, given a float input x, the mapping is defined as

$$m(x) := \left\lfloor \frac{x - l}{s} \right\rfloor$$

The input ranges and step sizes have to be provided by the user. While this ensures the necessary data privacy, this algorithm has certain limitations. Firstly, both the AES encryption and OPE's integer mapping increase the document size, resulting in larger indices. Furthermore, since all ordered types are mapped to integers, data-specific ES operations such as date aggregations are not supported. Next, due to

the nonlinearity of OPE, relations requiring a fixed distance such as $|x - y| < 5$ are not possible. Additionally, if different keys are used for different numeric attributes, they cannot be compared anymore. Lastly, in order to only require one key per row, the same Initialization vector is used for all entries.

8.3 Experimental Setup

To determine the general applicability of the designed solutions, they are extensively benchmarked. In the following, the general experimental setup is described.

8.3.1 Datasets and Queries

Two different datasets are used for the benchmarks. The NYC Taxis dataset, based on the `nyc_taxis` Rally track as well as a custom use-case-driven MRI benchmark that is developed based on extensive requirement engineering with the involved doctors and researchers.

NYC Taxis

The *NYC Taxis* dataset contains all rides that have been performed in yellow taxis in 2015. It is comprised of 165 million entries, resulting in an uncompressed size of 74.3 GiB, which makes it the largest corpus of all tracks provided by Elastic. Moreover, its Rally track is one of the canonical benchmarks commonly used for benchmarking ES, making the results comparable to previous work. Furthermore, while most Rally tracks are used for tail latency performance regression analysis, NYC taxis are designed for maximum throughput, making it a realistic use case for HPC computing.

Custom MRI

For the data model of the MRI benchmark the model as it is defined in Section 6.4.1 is used and extended with a few extra defined keys like the *brain age*. Since only a limited number of scans can be done within the scope of the project, an artificial data set needs to be created to simulate a large cohort. For this, the corresponding values and their distribution are determined and modeled. For instance, the *brain age* of a patient is modeled as a normal distribution around the patient's real age with a variance of 5 years. Other examples are the *KSpace* resolution, the magnetic field strength, the Larmor frequency, or the vendor-dependent measurement protocols, which are all chosen from a range of valid values of the existing scanners. Compared to the previous NYC taxis documents, these documents are rather large, with at least 61 different keys, although the overall corpus with 1 million documents is rather small, but realistic considering that each document represents an MRI scan.

8.4 Results

The individual benchmark runs are done in a grid search manner, scaling the cluster size by increasing the node count from three, over seven to 15, and for each cluster size increasing the number of ingestion or query workers from 1,2,4,8 to 16. This exhaustive approach results in a large dataset of which only a limited subset can be presented and discussed to highlight interesting caveats. The benchmarks are

performed on Emmy[2], using double socket nodes with 2 Xeon Platinum 9242 CLX-AP, with 368 GiB of available RAM. The used storage was a Lustre consisting of 2 ES14KX with 500 12TB SAS HDD each and an SFA7700X with 16 1TB SAS SSD connected to 4 metadata servers. The compute nodes are running CentOS 7, and Singularity 3.11.0 and ES 8.6.0 are used. In all cases `xpack.security` is disabled, there is one shard per ES instance without any replica, and before each individual benchmark run, the cache is cleared, and the caching of the index within ES is disabled.

8.4.1 Rally Benchmarks

The first benchmarks are executed with Rally on a single node cluster using the NYC taxis corpus. On all three setups, i.e., the baseline, the GoCryptFS, and the ABE scenario the same throughput of three operations is recorded, which corresponds to a total maximum of 30,000 documents per second for a simple *match_all* query, which has a filter that is always true, i.e., just returns any document. The observed problem is, that Rally throttles itself when the latency starts to increase too much, or even timeouts occur. This limitation could not be circumvented by configuring Rally differently. However, it can be assumed that real-world use cases utilize less sensitive client/user workloads. To also simulate these and see, what an ES cluster is really capable of through putting, a new benchmark is needed.

8.4.2 Throughput Benchmarking

In this section, the proposed throughput benchmarking tool is presented.

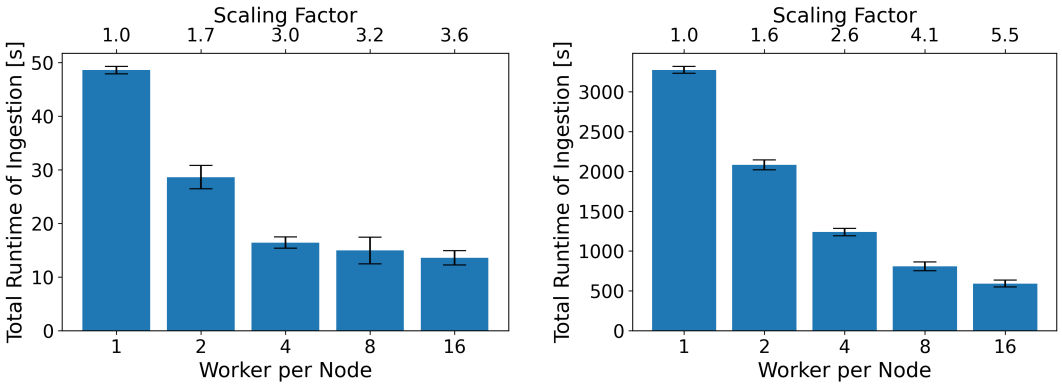
Ingestion

To benchmark the performance and scalability of the ingestion process the two corpora, i.e., NYC taxis and the custom MRI dataset, are once fully ingested. To facilitate the different numbers of workers, or ingestors, the used corpus is split into equally long parts and each worker is solely responsible for ingesting one unique part of the corpus. Each of these calls ingests 10,000 documents at once, for which the time is measured and stored within an array in each worker process. When the entire corpus is successfully ingested, all unique timing arrays from all workers are streamed into a single result file.

Analyzing Single Runs A single benchmark run is evaluated by summing up all individual timings of a single worker. This yields the total waiting time of a worker for an ES response. The measured times include the network latency, as well as the time ES requires to calculate the response. From these timings, the mean and standard deviation of all workers are calculated. The results are shown for two examples in Figure 8.1 for a three-node ES cluster with varying workers per node for both the MRI and the NYC datasets using an ethernet interface. Overall, the ingestion time is reduced from (49 ± 1) s to (14 ± 1) s when going from a single worker per node to 16 workers per node for the MRI use case, while only showing a diminishing return when going from four workers per node to 16. Similarly, the ingestion time of the NYC benchmark is reduced from (3270 ± 44) s to (590 ± 42) s, however exhibiting a larger benefit of more workers considering the runtime drop from (1238 ± 46) s for

[2] A German 6 PFLOP CPU-based system.

four workers to (592 ± 42) s for 16 workers. The resulting scaling factors are 3.6 for the MRI dataset and 5.5 for the NYC dataset when scaling from 1 worker to 16.



(A) MRI Dataset (B) NYC Dataset

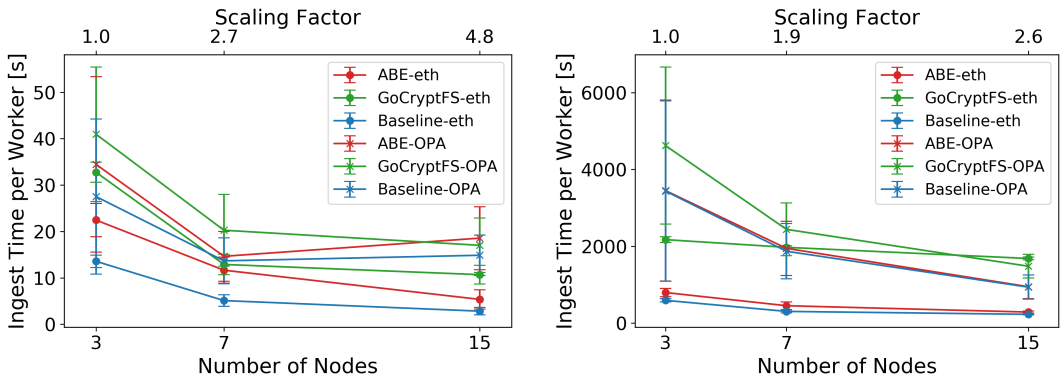
FIGURE 8.1: Geometric mean of the ingestion times on a 3 node cluster via ethernet without encryption.

Analyzing Different Cluster Sizes To determine the influence of the cluster size, the chosen network interconnect, i.e., OPA or Ethernet, and the cost of the encryption, the single run analysis is done for all possible configurations. This gives a total of 180 independent measurements, as exemplified in Figure 8.1. To ease the representation, the number of considered measurements is reduced to 36 by only considering the fastest run depending on the number of workers. This would mean, that for the measurements shown in Figure 8.1 only the runs with 16 workers are used.

The results are shown in Figure 8.2. The first observation is that in all cases the Ethernet performance is much better than the OPA used via Internet Protocol over Infiniband (IPoIB). The second observation is that generally, GoCryptFS exhibits the worst performance. This is expected since every operation has to go through the additional FUSE layer and needs to be transparently encrypted. This is different from the baseline scenario, where no encryption is employed, and the ABE scenario, where the encryption is not covered by the measured round-trip time since the encryption is done by the client. This approach is chosen to measure the scalability of the server-side NoSQL database setup. If the encryption had been included in the measurements presented in Figure 8.2, the embarrassingly parallel properties of the encryption of the individual documents would have overshadowed the effects of the server-side setup. Therefore, the ABE encryption costs are analyzed in isolation in Section 8.4.2. The increased ingestion time of ABE compared to the baseline can be explained by the overall larger dataset, where the one million documents in the MRI use case have a cumulated size of 1.5 GB whereas the ABE-MRI dataset has a size of 4.3 GB. Similarly, the size of the approximately 168 million documents for the NYC taxis data set is 75 GB for the unencrypted dataset and 184 GB for the encrypted.

The resulting speedups are largest for the MRI use case and the Ethernet interface ranging between five to three times and for the OPA interface between two and three. For the larger NYC dataset speedups for the Ethernet interface are in the range of one to three and for the OPA interface very homogenously three. Considering that the observed scalability is much better for the MRI data set than for the larger NYC

dataset, it could be that caching effects become more dominating in the MRI dataset.



(A) MRI Dataset (B) NYC Dataset

FIGURE 8.2: Ingestion time for varying cluster sizes for Ethernet and Omnipath interconnects. The scaling factor is calculated using the ethernet-based baseline measurement.

Measuring the ABE Encryption Cost

In order to measure the encryption cost of the ABE method, 10 times a random 10,000 document subset is picked from the NYC taxis and the custom MRI corpus. Then the encryption operation is measured end-to-end within the encryption function itself, therefore excluding other effects like the I/O overhead when reading the data from the filesystem. From these measurements, the expectation value and the standard deviation are calculated. This yields:

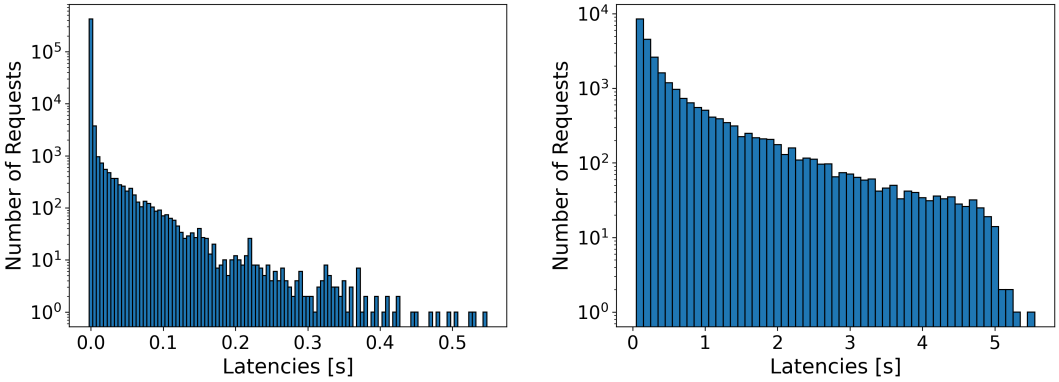
- NYC: (0.456 ± 0.001) s per document
- MRI: (0.079 ± 0.002) s per document

These results are extraordinarily slow because a very inefficient OPE implementation is used, i.e., *pyope*[3], which implements the encryption as a recursive function in *Python* without further optimizations. This is also the reason, why an NYC document takes so much longer than an MRI document. Since in the NYC dataset a large amount of numbers are used, the ABE algorithm has to be used more often than in the MRI dataset, where more text is used, which is more efficiently processed with an AES256 encryption.

Match All Queries

To establish a baseline, the *Match_All* query is used. This query just matches all documents. Thus, no complicated logic has to be done by ES and it can just stream the requested documents as soon as it gets the request for it. However, other effects can outweigh this simple model. For instance, ES dynamically allocates threads to execute and manage requests. On an unoptimized index with a high segment number the overhead of creating too many tasks to be executed on these segments can diminish the advantage of applying no logic within the filtering of documents.

[3]https://github.com/tonyo/pyope



(A) One worker per node with a return size of 10 documents per response with a 5ms steps.

(B) 16 workers per node with a return size of 10k documents per response with 100ms steps.

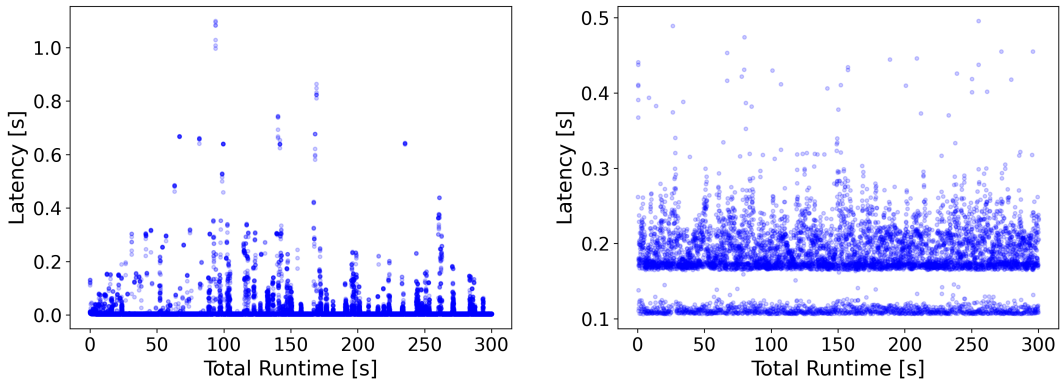
FIGURE 8.3: Histogram of the number of requests with regard to the full round-trip latency for the 7-node ethernet cluster using the NYC dataset.

Thus, the *Match_All* query is not necessarily the very baseline query, but it is useful for either server configuration fine-tuning or network overhead optimizations.

Since the query benchmarker records for each single request the latency as well as the number of returned documents, one can do a fine-grained analysis of the distribution of the measured latencies. These are shown for two different *Match_All* workloads in Figure 8.3. One can see in both cases that the distributions of the measured latencies for the individual requests follow some kind of super-exponential decay. However, for the less intensive workload with only one worker per node, shown in Figure 8.3a, it peaks strongly around 50 ms and only has very few tail latencies around 500 ms whereas the more intense benchmark configuration with 112 workers in total and a response size of 10,000 documents peaks around 80 ms and shows a constant super-exponential decay to 5 s. These two measurements demonstrate how the ES cluster continuously gives in to an increasing load, which results in larger latencies.

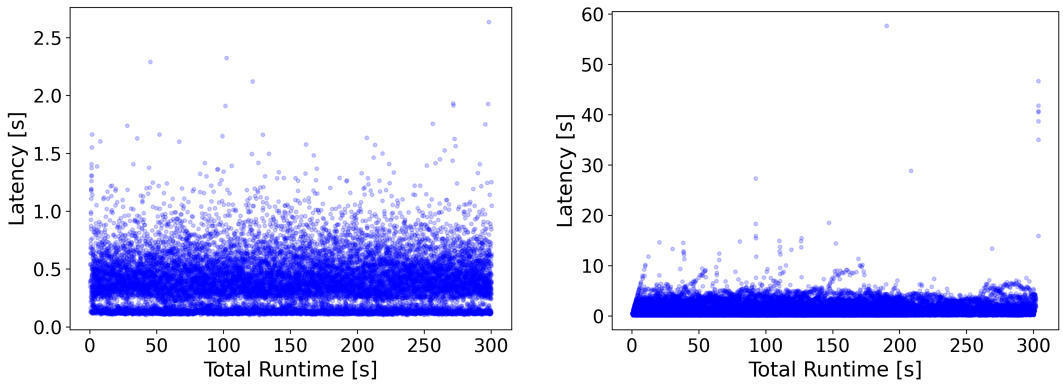
To further analyze how these distributions are measured, an exact timeline is measured and shown in Figure 8.4 and Figure 8.5 for the measurement of the seven-node cluster for the NYC baseline measurement. One can see that in Figure 8.4a for the request size of 10 documents and one worker per node a striping pattern is visible during the entire benchmark run. These could be caused due to beating patterns within the request frequency. Increasing the response size from 10 to 10,000 documents as shown in Figure 8.4b leads to two distinct bands. This can be explained that here some workers are randomly assigned to the localhost. The reduced network overhead then leads to lower latencies. Due to the one thousand times increased response size, this network overhead becomes increasingly relevant and measurable. When further increasing the workers to four per node as shown in Figure 8.5a, one can see the bands become blurred until they are indistinguishable in Figure 8.5b. Increasing the number of workers leads to a much larger dispersion and an overall increased latency.

Since always a super-exponential decay is observed in Figure 8.3 with a mixed dispersion over time, the geometric mean is used to average over an entire measurement to obtain the average latency for a single request. The mixed dispersion should emphasize, that depending on the exact query and load beating patterns, bands, or



(A) Return size of 10 documents per response. (B) Return size of 10k documents per response.

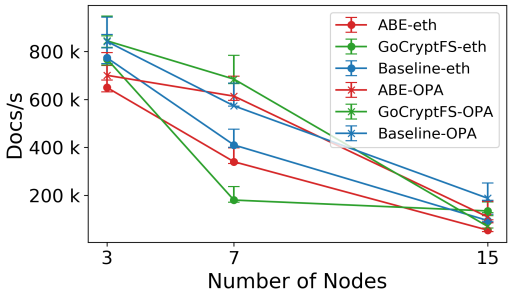
FIGURE 8.4: Timeline of the individual request latency for the seven-node NYC baseline benchmark with one worker per node.



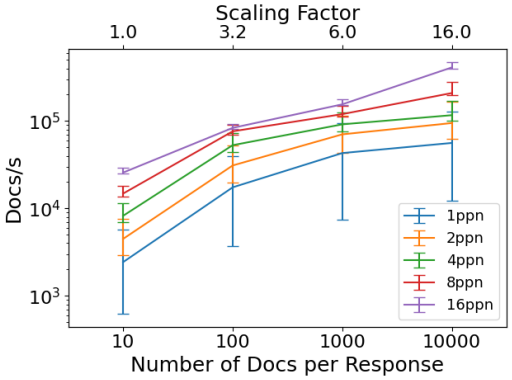
(A) Four workers per node. (B) 16 workers per node.

FIGURE 8.5: Timeline of the individual request latency for the seven-node NYC baseline benchmark with 10k documents response size.

a homogenously decreasing dispersion can be observed. All of these different distributions would require different modeling, making comparisons between them nearly impossible. Therefore, the geometric mean is chosen to be a simple, easily comprehensible but fair measure for the general tendency for the measured latencies, while somehow acknowledging the probably rather multiplicative nature of the data than an additive. Dividing the number of documents per request by this averaged latency yields the averaged documents per second, as can be seen in Figure 8.6b for the seven-node cluster running the NYC baseline benchmark. One can see, that by increasing the number of documents per response the overall throughput increases as well. Similarly, the overall throughput is increased by utilizing more workers to increase the number of parallel requests. In order to determine the encryption cost and the advantages of larger clusters, similar to the analysis of the ingestion, only the best-performing values are taken into account for each cluster configuration. Through this approach, the individual optimum can be used, thus optimally representing a specific server configuration while still reducing the 360 individual results to only 18 which can be more easily comprehended. That means for the measurements in Figure 8.6b only the one with 16 workers per node is selected. The result is shown in Figure 8.6a. One can see that with increasing cluster sizes the throughput decreases. The reason for this behavior can be that in order to create



(A) Geometric mean of the documents per second for the best-performing worker configuration for a match_all query.



(B) Throughput for a seven-node cluster baseline for the NYC dataset.

FIGURE 8.6: Geometric means of different Match All query measurements.

Cluster-Size	Baseline [s]						GoCryptFS [s]					
	Ethernet			OPA			Ethernet			OPA		
	gmean	1st	99th	gmean	1st	99th	gmean	1st	99th	gmean	1st	99th
3	7.4	6.8	65.5	8.5	7.9	57.7	7.1	6.9	19.2	7.1	6.9	16.3
7	2.9	2.9	3.7	3.5	3.4	3.6	2.9	2.9	3.1	3.0	2.9	3.6
15	1.6	1.6	1.7	1.6	1.6	1.8	1.4	1.4	1.5	1.6	1.6	1.8
Speedup	4.6			4.6			5.1			4.4		

TABLE 8.1: Geometric means and the first and 99th percentile of the latencies of a histogram aggregation.

the response, the requested ES node needs to communicate with all the other nodes which are also holding an active data shard within the index. Since this benchmark is configured in such a way, that every ES node is an active data node, every node has to communicate with every other node. This adds a lot of overhead but yields no parallelization advantage since the request does not contain much logic that has to be processed.

NYC Aggregations

Analyzing the *Match_All* queries has demonstrated potential drawbacks to the performance when scaling out. However, it can be expected that positive effects can be observed when the workload required on the node to fulfill the request is increased. Such a more compute-bound query is a histogram aggregation within the NYC data set, where first all trips with a distance of greater than zero and smaller than 50 (miles) are selected and then aggregated into histograms according to their trip distance and cost. The result is shown in Table 8.1. One can see, that for the baseline as well as for the GoCryptFS scenario the latency drops from around 8 s per request for a three-node ES cluster, to 3 s for a seven-node cluster and to 1.6 s for a 15-node cluster. Not only can almost linear speedups be observed, but particularly the dispersion decreases drastically, since the first percentile, the geometric mean, and the 99th percentile are much closer for larger clusters. The speedup shown in Table 8.1 refers to the speedup of the geometric mean of the three-node cluster to the 15-node cluster. Only this value is discussed, to motivate the average speedup that can be expected, when scaling the cluster size for this workload. A speedup value of five when scaling from three to 15 nodes corresponds to linear scalability. Therefore,

for the baseline scenario, 92% efficiency is obtained when scaling out, while for the GoCryptFS OPA setup 88% efficiency is achieved. The observed performance benefit of GoCryptFS could be due to additional page-caching introduced by the FUSE layer or it could be due to measurement uncertainties since the variation would also allow the actual performance of GoCryptFS to be slightly lower than the baseline. Therefore, the additional encryption cost when doing compute-intensive operations is negligible.

MRI Custom Benchmark

Based on previously made experience with the standard NYC dataset, the presented benchmarking tool is used to run a custom MRI benchmark, which model is based on the in Section 6.4.1 developed data model with a few extra attributes like the *brain age*. For this, the following set of queries is defined together with researchers who want to use the data catalog in the future:

- **Body Part:** A *term* query for a body part, in this case, the head. Corresponds to "return all MRI scans of the head".
- **Systemvendor:** A *match_phrase* query for the Systemvendor, e.g., Siemens, Philips, or General Electric. Corresponds to "return all MRI scans from a specific system vendor".
- **Age:** A *range* query to select an age range of patients. Corresponds to "return all MRI scans of patients with an age between 50 and 60 years".
- **Body Part and Resolution:** Boolean-chained *term* queries selecting a body part and two resolution parameters. Corresponds to "return all MRI scans of the head with a resolution, i.e., *KspacePhaseEnc* of 256 times 256".
- **Brain-Age:** A *filter* which is based on a script included in the query which determines if the brain age is larger than 5 years than the real patient age.
- **Vendor and Body Part:** Boolean-chained *term* and *match* queries to query a body part and then match it against a specific system vendor and protocol. Corresponds to "return all MRI scans of heads done on either a Siemens scanner with an MP2RAGE or a General Electric scanner with a BRAVO sequence".
- **Age-Weight Distribution:** A chained *match* query for the patient gender, followed by a *range* query of the age with an ending histogram aggregation. Corresponds to "select all men with an age lower than 70 and do a histogram of the patient weight with a binning size of 5".

To further determine the ideal setup, the same grid search as for the NYC dataset is done. In Table 8.2 the maximum throughput for documents per second is determined for each cluster size and query. To simplify the comparison between the queries, only the geometric mean is provided as a measure of general tendency, and the standard deviation is provided to provide some kind of measure of dispersion. This is done, since the additive nature allows for a very compact, easily understandable, and comparable notation, as it is shown in Table 8.2. That this statistical analysis provides only a limited representation is extensively discussed in Section 8.4.2. However, for an overview, the proposed, compact notation seems more useful.

For this, each run is individually analyzed similarly as shown in Figure 8.6b to determine the ideal setup for each cluster size and query. One can see in Table 8.2

Setup	Cluster-Size	Body Part	System-vendor	Age	Body Part Resolution	Brain-Age	Vendor+ Body P.	Age-W. Distr.
Baseline	3 Node	190k±15k	180k±13k	190k±15k	124k±6k	94k±5k	96k±5k	4k±160
	7 Node	212k±10k	244k±4k	103k±5k	114k±6k	101k±4k	166k±4k	12k±700
	15 Node	39k±9k	37k±4k	49k±4k	48k±34k	37k±4k	72k±5k	5k±2k
GoCryptFS	3 Node	180k±13k	157k±10k	176k±13k	119k±4k	170k±13k	153k±5k	5k±100
	7 Node	71k±9k	100k±12k	51k±4k	89k±7k	50k±5k	103k±4k	6k±480
	15 Node	29k±6k	33k±3k	49k±3k	41k±19k	39k±3k	60k±4k	4k±670
ABE	3 Node	120k±15k	110k±11k	160k±6k	90k±80k	-	140k±7k	-
	7 Node	80k±20k	80k±7k	80k±7k	110k±20k	-	95k±7k	-
	15 Node	6k±8k	10k±8k	12k±8k	11k±9k	-	12k±8k	-

TABLE 8.2: Documents per second for the MRI use case with the Ethernet interface.

Setup	Cluster-Size	Body Part	System vendor	Age	Body Part Resolution	Brain-Age	Vendor and Body Part	Age-Weight Distribution
Baseline	3	0.6±0.1	0.7±0.2	0.6±0.1	0.02±0.02	0.6±0.1	0.7±0.1	0.020±0.005
	7	0.7±0.1	0.7±0.1	0.7±0.1	0.02±0.02	0.7±0.1	0.5±0.1	0.013±0.003
	15	0.9±0.3	0.9±0.2	0.7±0.2	0.03±0.05	0.9±0.2	0.5±0.1	0.009±0.004
GoCryptFS	3	0.6±0.1	0.6±0.1	0.6±0.1	0.02±0.01	0.6±0.1	0.5±0.1	0.022±0.004
	7	0.7±0.1	0.7±0.1	0.7±0.1	0.02±0.02	0.7±0.1	0.5±0.1	0.013±0.003
	15	1.0±0.3	0.9±0.3	0.8±0.2	0.03±0.05	0.9±0.3	0.5±0.1	0.010±0.005
ABE	3	1±0.2	1.2±0.3	1.2±0.3	0.04±0.001	-	1.1±0.2	-
	7	1.4±0.6	1.4±0.6	1.4±0.6	0.04±0.001	-	1.2±0.2	-
	15	2.5±3.7	2.6±3.6	1.8±2.5	0.05±0.09	-	1.2±1.3	-

TABLE 8.3: Latencies for the MRI custom use case with the Ethernet interface in seconds.

that for the baseline scenario, a seven-node cluster performs best for most cases, with only the Age and the Body Part and Resolution query peaking for the three-node cluster. For the Body Part, Body Part and Resolution, and the Brain-Age queries, the results lie within the error, so no direct conclusion is possible.

While for the GoCryptFS the three-node cluster performs best, except for the Age-Weight Distribution query which peaks for the seven-node cluster. Generally, all queries performed slightly worse in the GoCryptFS setup compared to the baseline, for individual performance drops ranging from a 5% to 13%, when comparing the two three-node setups. The observation, that the Brain-Age and the Vendor and Body Part are faster on the GoCryptFS setup is not clear beyond any doubt but could be explained by a temporary noisy-neighbor effect. The very limited effect of scaling from three to seven nodes can be explained by the smaller dataset size compared to the NYC corpus. Here, only a million documents, each representing a single MRI scan, are used, while the NYC taxis dataset has more than 165 million documents. Therefore, the additional network overhead of the intra-ES cluster communication and thread management can not be outweighed by large parallelization gains.

To further analyze the advantages or disadvantages of scaling out, the latencies for single requests are analyzed in Table 8.3. One can see that for all queries except the Age-Weight Distribution the latencies increase or stay at least constant for larger cluster sizes. The Age-Weight Distribution seems to profit from a larger parallelization for the decrease of a single request latency. The reason, that the overall latency of the Age-Weight Distribution is very small compared to the others, excluding the Body Part and Resolution query, is that generally 10.000 documents are returned per response. Thus the overall payload of the response of the Age-Weight Distribution is three orders of magnitude smaller compared to the others, drastically reducing the network overhead which is included in all presented benchmarks.

8.5 Evaluation

Two different encryption techniques are presented which can be used to provide secure metadata indexing functionalities for sensitive information. Both of these techniques can be used within the modular design of the data lake to provide the required metadata management. While ABE is implemented purely on the client side, it can not only be used in high-throughput scenarios on HPC systems but also within (public) cloud environments, while the described GoCryptFS-based method using the SecureHPC platform cannot be deployed outside of this scope. However, while the presented ABE method currently has a few shortcomings when equally spaced distances are required, the GoCryptFS method provides the full capabilities of the underlying database. One interesting result from the benchmark using the synthetic MRI dataset is that under the condition of 1 million documents, i.e., MRI scans, higher throughput cannot simply be achieved by scaling out. Therefore, it needs to be carefully verified, that the recorded throughputs and latencies are sufficient. If not, further node-based optimizations should be done, which can also be tested using the presented benchmark. Within the "Big Data MRI"-project, only less than 200 scans have been processed using the developed data lake. For this use case the achieved performance is sufficient and considering the timings of the synthetic benchmark no further actions seem necessary, since both the throughput and latencies are sufficient to still cover this use case.

8.6 Summary

In conclusion, a workflow to deploy on-demand, user-owned ES clusters for sensitive data on HPC systems is presented, which allows for dynamically scalable, workload-defined cluster sizes. Two different encryption techniques are implemented and are systematically benchmarked with a novel tool that can be used to create very aggressive throughput-oriented workloads, as well as latency-sensitive benchmarks. The importance of accurate data and workload modeling is demonstrated, showing that for simple queries even on large datasets consisting of more than 165 million documents, larger ES cluster can decrease performance due to additional network and thread management overhead, while more compute-intensive operations can largely benefit from larger clusters, as is demonstrated for the histogram aggregations. The presented method, ranging from the data and workload modeling to the analysis of the benchmark is a blueprint that can be similarly applied for other use cases as well. The interpretation of the gathered results relies on the determined requirements of the users. Combining an ES cluster with one of the proposed encryption techniques can then be used as a secure and scalable backend for a data catalog. By generalizing the modeled workload, one can provide a first template for more abstract and use case-specific data catalog functionalities. To integrate the proposed workflows into the overarching data lake architecture, one needs to include a reference to the index within the (F)DO record, including machine-actionable instructions on how to get access to the metadata and the corresponding keys.

Chapter 9

Conclusion and Future Work

In this chapter, a short summary is provided in Section 9.1, highlighting the main contributions of this work. Afterward, an outlook for future work is provided in Section 9.2, alongside a discussion of corresponding improvements of the presented work.

9.1 Conclusion

In the following, the main contributions of this thesis are highlighted by summarizing how the individual contributions answer the initial research questions introduced in Section 1.2. For this, first, the more detailed sub-questions are addressed, before everything is set into perspective when focusing on the overarching goal of this thesis.

9.1.1 Data Lake Architecture

> *What data lake architecture can be used to organize a flat-hierarchical data lake with respect to data maturity and offered functionality that is built in a modularized manner using scalable components?*

Within the initial requirement analysis in Chapter 2 a set of requirements are identified. Two of these key aspects are generality and modularity, which should ensure that specific systems can be included in the future to provide new functionalities when extending beyond the original scope or that components can be exchanged due to technological progress. In order to enable drop-in replacements of backend services, which the users would not necessarily recognize, one needs to abstract the complexity of these utilized systems. Within an extensive literature survey, different data lake architectures are discussed in Section 3.3, where the current state-of-the-art of abstracting based on systems is discovered and the resulting disadvantages like a high complexity for users and admins are motivated. Therefore, a novel data lake design is proposed which uses (F)DOs to organize the otherwise flat namespace. The type of the (F)DOs can be used to infer an order indicating the data maturity, or in the case of tasks, the functionality. The proposed data lake design is therefore one concrete implementation of the class of hybrid architectures [167]. Hereby the (F)DOs provide a homogeneous interface to users by the data lake frontend or as (F)DO records of a reference service. Access to the underlying services is managed by functions which are hiding the complexity of the actual implementations from the end users.

9.1.2 Scientific Support

How can this data lake architecture be made FAIR by design, allowing for the integration of diverse scientific workflows and enforcing overarching data management plans across the lake?

Within the gap analysis in Section 3.7.5, it is identified that there is a general lack of ecosystem-agnostic compute capabilities in existing data lakes. These purpose-built data lakes can therefore not easily integrate new scientific workflows that rely, for instance, on HPC systems. Since there are in particular no data lakes available that support offloading of compute-intensive tasks to HPC systems, the general challenges of doing data-intensive research on HPC systems are analyzed in Chapter 4. The proposed governance-centric interaction paradigm relies on the creation of an experimental description in advance, detailing the input and output data sets, and the tasks which process them. These constituents of the overarching DMP can be considered to be functions, which are well-defined tasks, within well-defined environments. Executing pre-defined functions is also very common within clouds following the FaaS computing model. Thus, this computing idiom is ideally suited to provide a homogenous interface to both cloud and HPC infrastructure. For this, a REST-based FaaS interface for HPC systems is developed in Chapter 5.

Since the general computing paradigm revolves around the fundamental idea of executing functions, the previously proposed DO-based data lake architecture can be extended to use FAIR DOs. FDOs are designed in such a way, that they fulfill the FAIR principles. Their definition extends these of DOs, and also includes the notion of operations. These are functions, that can be executed with the corresponding FDO as input data. Thus, using FDOs as the central building block allows scientists to work FAIR, while the generic, function-centric processing model provides a convenient interface for diverse compute infrastructure. Since a DMP consists of a structured workflow connecting input/output data to their processes, they can be defined on the data lake level, to ensure global consistency.

9.1.3 Secure Processing of Sensitive Data

What are the attack vectors to gain access to a user's data while processing them and how can trusted and secure compute capacity be provided which ensures the full data sovereignty to the users?

Since the proposed data lake focuses on integrating HPC systems to offload compute-intensive tasks, only these are focused on providing secure processing capabilities. This platform should be used to process sensitive health data, which is one of the key requirements gathered in Chapter 2. The gap analysis in Section 3.7.5 also revealed a negligence of existing data lakes on this topic. Therefore, a novel service called SecureHPC is proposed in Chapter 7, which is designed around the basic assumption that users on HPC systems have bare-metal access and the available software stack of these machines has vulnerabilities. Thus, to provide a secure computing platform, it needs to be isolated from an attacker with administrative privileges. For this, a security onion is utilized to provide an isolated partition on a shared HPC system, which fully restricts the lateral movement of an attacker to this partition. Access to this partition is only possible using a special workflow, which ensures full end-to-end encryption as well as the legitimacy of the compute request by confirming the authenticity with a second factor. This workflow can then execute pre-configured batch jobs, which completely integrates with the execution model of asynchronous

functions, which can be triggered on HPC systems using a common SSH connection, or the in Chapter 5 proposed HPCSerA service. Using this FaaS interface, it can be homogenously integrated into the proposed (F)DO-based data lake architecture.

9.1.4 Secure Metadata Handling of Sensitive Data

> *What are the attack vectors to gain access to data stored in a database or search engine and how can a secure platform be provided that ensures the full data sovereignty to the users?*

A data catalog, containing semantically meaningful information about the data and their linkage, is a key component of a data lake. Within the proposed (F)DO-based data lake architecture, every (F)DO has to be accompanied by describing metadata, or a reference to it. To provide a secure and highly scalable data catalog, capable of indexing sensitive, health-related metadata, two different methods are proposed in Chapter 8. One uses client-side encryption of the metadata before they are ingested, thus it can be used for data catalogs hosted on untrusted systems. The second option utilizes the previously developed SecureHPC service as a platform, on which the ES clusters and the specific operations on them are executed. This relies on server-side encryption. Both techniques guard against an unsecured REST interface of the ES clusters, as well as against direct storage access by attackers.

9.1.5 Summary

> *How can a data lake be designed so that it can serve different scientific domains as the central data repository and processing platform, with a specific focus on sensitive data?*

In conclusion, a generic data lake is proposed that models data as (F)DOs and provides a homogeneous interface through its reference services, i.e., the data lake front end. Using FDOs ensures transparently that scientists work FAIR and introduces the notion of operations that can be executed on (F)DOs. This interaction paradigm matches ideally with the FaaS idiom. By providing a FaaS interface for HPC systems, this method can be used to offload compute tasks to cloud or HPC systems and thus provide ecosystem-agnostic compute capabilities. Additionally, since the complexity of the underlying services is abstracted by the data lake frontend, a modular design is established which allows to add or switch backend services required for the integration of new scientific workflows. Additionally, this approach also enables seamless integration of the secure compute and metadata platforms that are proposed within this thesis. Using the *Decomposition and Encapsulation* and the *Problem Generalization* steps of the employed scientific method, three services are successfully spun out of the overarching data lake and are available as stand-alone components outside the scope of the data lake. Their successful deployment is described, including their deployment in clinical workflows. Thus, the employed scientific method has successfully maximized the overall impact as it is illustrated by the different use cases outside of the original scope.

9.2 Future Work

The current state of the presented research can be further extended, to either extend the capabilities of the services, mitigate shortcomings of the current state, or adapt

these research artifacts to novel technologies. These opportunities are discussed in the following.

9.2.1 Integration of Data Management and Compute Resources

The proposed *Governance-centric* interaction paradigm depends on a DMP tool that can process the information flow originating from the processing tasks. This tool has only been theorized within this work but has not been prototyped, which would be the first step towards a viable service. Since the DMP tool depends on an experimental description of the users' workflow, it is favorable to integrate it into the compute time proposal, as it is common on larger, i.e., Tier-2 clusters and upwards. This integration should be done in a meaningful way so that it will not appear as an additional section, and burden the researchers even more, but that it can ease and streamline the overall process by reducing the work at other places. One such point could be the proof of scalability. Here, it would be enough if users could demonstrate that they could utilize at least one node, and achieve any further parallelization through data-parallelism. However, simply dropping these sections in favor of a machine-readable DMP might impede the job of the reviewers evaluating these proposals for their applicability to get compute time.

9.2.2 HPCSerA

The presented REST service for remote HPC access is focused on the general, FaaS-based interaction paradigm, alongside an extensive security architecture. Within this work, it is assumed, that the functions a user wants to execute are already preconfigured. Therefore, an efficient way of providing these functions is still required. One option would be to build up a repository of functions that users can import. This can work even entirely through the REST API of HPCSerA, where the agent can import these functions using the code-ingestion endpoint, which is secured with a second factor.

The general idea of the code-ingestion endpoint can also be extended to provide better ad-hoc processing support. This requires either an extension of the FaaS-paradigm, to also cover the execution of unconfigured functions, or an extension of the security architecture if code is dynamically ingested via streams. The first approach is more in line with the general idea of HPCSerA. An initial approach could be to provide functions that can accept code as arguments. One can then make a function call for each code snippet that one wants to execute. In the current security design, this would require for each function execution an acknowledgment by the user within HPCSerA. This workflow needs to be adapted to provide a similar security level while providing more user-friendliness, e.g., by only requiring an acknowledgment once within a fixed time frame.

In the current setup, the execution of the agent is triggered by a cron job, or a *systemd timer unit*. For time-critical executions, this lag is too large. Instead, the execution of the agent needs to be triggered with the actual function call to the API server. There are generally different ways to do it. For instance, the agent can be long polling on the endpoint from which it accepts incoming functions. Another option would be to have a dedicated user on the HPC system with an SSH force command, which is allowed to `sudo` to other users and only execute the agent there. This user is then remotely accessed via the API server to trigger the agent on the HPC system within the user space of the requesting user.

Lastly, data transfers are only handled for small files, while for larger data transfers more specialized services like S3 are used. To increase efficiency and user-friendliness, the boundaries of small and large data transfers need to be systematically benchmarked, and a tighter integration of S3 into HPCSerA is envisioned where ideally users do not need to set their secret and access keys but can use their HPCSerA credentials to configure access to their S3-bucket.

9.2.3 Data Lake Design

The implementation of the presented data lake design is currently only usable with the abstraction of DOs in a programmatic way. This means, that there is no web interface available. Instead, interactions with the data lake and the DOs within are currently done via a Python-based Software Development Kit (SDK). Simply developing a web UI on top of this SDK is not sufficient, since specifically with graphical interfaces, domain-specific nuances are a main concern. For this, it is important to find patterns to standardize the process to map this generic, DO-based data lake design on domain-specific use cases and their graphical representations. In addition, the communication protocol currently used should be substituted or extended to use the DOIP once it is available to provide true FDOs. This will also ensure full machine actionability and will provide interoperability with other services and frontends, which can then provide the user-specific web UIs with the required domain-specific refinements.

While the current design aims to be as flexible, and therewith as powerful as possible, this also entails that a deep knowledge of the system and its concepts is required to be able to quickly set up new pipelines and reuse existing infrastructure. Here, further research is necessary to define a restricted subset of functionalities, which allows untrained users to quickly set up workflows to use the data lake with a basic set of features, and provide this in easily accessible interfaces.

The data lake holds data during the entire lifecycle. This means that it contains cold, warm, and hot data. Therefore, different storage tiers are required to provide either the required performance or the durability. During the application within the "Big Data MRI"-project, this is solved by including an explicit data staging step, i.e., copying the data to a fast filesystem before processing it on the HPC system. Although this works fine for this specific use case, in other use cases a more dynamic data access from the HPC system to the cold storage tier might be required. This can lead to a situation where many, even hundreds of compute nodes of the HPC system need to access the same data on the cold storage tier. This will then bottleneck either on the file system, or network congestion will limit the overall throughput, leaving the compute nodes idling while waiting for the data. To solve this problem, a more efficient way of loading data is required, where instead of loading the same data n times from n nodes, it is only read one time from the remote storage and then distributed within the HPC system using the high-performance interconnect. One way to transparently do this is through a FUSE, which uses MPI to synchronize file access. For example, one process could load a file and then broadcast it to all other processes on all participating nodes. Similarly, using one-sided communication would enable other nodes and processes to fetch missing data from a participating HPC node, instead of loading the data from the remote filesystem. Providing these functionalities via a FUSE mount allows the application to use a common file descriptor without any knowledge of the underlying filesystem, including any MPI communication. A prototype of such a FUSE mount is already implemented within this work but is not explicitly discussed within this thesis.

Within the presented data lake, encryption is used to securely store data and metadata. However, currently, the key management is entirely up to the user and is not discussed at all. The presented KMS within the secure HPC partition and the secure metadata handling are only used as a communication layer where keys are only retained for a short period of time. Here, a proper KMS needs to be integrated on the client side to manage the keys over the entire lifecycle. Here, further research can be done particularly with regard to a suitable granularity. Encrypting each file with a unique key is technically possible, however, when a multitude of different files should be processed within a single job, this introduces a large overhead. On the other hand, a finer granularity can limit the extent of a security breach and can be used for additional, decentralized access control.

9.2.4 SecureHPC

For the IO500 measurements of the multinode setup presented in Table 7.6 the *ior-hard* operations have to be excluded, since neither *eCryptFS* nor *GoCryptFS* had support for parallel IO, meaning, that multiple processes from multiple nodes write to a single file. Following this discovery, this problem is fixed in *GoCryptFS* by using non-POSIX *Open File Description Locks* using the `F_OFD_SETLKW` command. These provide byte-range locks which are then used by each *GoCryptFS* process to lock access to one dedicated byte-range within the file. These modifications are successfully deployed on the clusters of GWDG. However, further research can be done with respect to the overhead cost of using this lock. In addition, currently, all processes are always calculating the full block cipher when writing into overlapping byte ranges. Here, further optimizations,e.g., by using collective I/O can be researched.

The current implementation only allows one user group on a single dedicated server. That is because it can currently not be reliably detected or prevented if a user makes an attack, e.g., a malicious change to OS. Therefore, changing the user group that can access this isolated server requires a reboot of the server to unlock the second factor again and allow a different user group to use it. In addition, a dedicated partition needs to be provided for each user group, or the Slurm state has to be locked to prevent non-eligible jobs from trying to run on this server. These two limitations can be circumvented in future work by providing additional hardening and a fine-granular isolation of the jobs. One direction to research is to start jobs within an isolated runtime, instead of running them directly on the host. Such runtime environments can be lightweight virtual machines like *Kata Containers* [164] which provide additional isolation of the processes running within this container from the host OS compared to other runtimes like *Singularity*.

Another approach to increase the trust of the secure server is to use remote attestation of its state between different jobs using runtime integrity monitoring. Here, all files are assigned a hash value, which is evaluated upon the file access. By combining this with a Trusted Platform Module, the measured state defined by the chaining of all hashes of all accessed files can be securely maintained and is even shielded from manipulation of a local root. These two measures are completely orthogonal and can therefore be efficiently combined.

Lastly, TEEs are constantly evolving and might be a suitable alternative in the future. For example, Intels Trust Domain Extension is not only capable of isolating an entire VM from the host, instead of only limited commands within the SGX environment, but it can also provide GPU passthrough. These efforts are however in a very early state. Currently, it is only possible with NVIDIAs H100 without runtime encryption for its on-card memory.

9.2.5 Secure Metadata Management

The developed benchmark and its use cases can in further work be used to compare the capabilities and performance of the presented on-demand ES clusters which are spawned in the context of normal Slurm allocation, with other databases, or filesystem services with search capabilities like *pixstor*[1]. Here, the presented measurements can be used as a baseline to better assess the performance of such storage systems.

To make this service more accessible for a wide range of users, a basic set of functions needs to be defined which can be wrapped into a simpler command line interface. This requires further research of diverse use cases that integrate such a data catalog service and then finding common patterns in their interactions with the data catalog. Similarly, the interaction of the proposed DMP tool in Chapter 4 with the data catalog needs to be defined on a technical level to provide a well-defined interface for these two independent services to work as smoothly in conjunction as conceptually envisioned.

Lastly, further improvements with the presented ABE method are desirable. For instance, a highly optimized implementation of the used OPE algorithm [26] can drastically reduce the encryption overhead. Similarly, the applicability of order-preserving versus order-revealing algorithms can be further analyzed. Lastly, efficient, client-side algorithms can be researched to substitute for fixed-range queries, like histogram aggregations or deviations, as it is presented with respect to the brain age.

[1] https://pixitmedia.com/products/pixstor

Publications

- Sven Bingert, Christian Köhler, Hendrik Nolte, Alamgir Waqar "An API to include HPC resources in workflow systems". In: INFOCOMP 2021: The Eleventh International Conference on Advanced Communications and Computation. 2021, pp. 15–20
- Philipp Wieder and Hendrik Nolte. "Toward data lakes as central building blocks for data management and analysis". In: Frontiers in Big Data 5 (2022)
- Hendrik Nolte and Philipp Wieder. "Realising data-centric scientific workflows with provenance-capturing on data lakes". In: Data Intelligence 4.2 (2022), pp. 426–438
- Mohammed Hossein Biniaz, Sven Bingert, Christian Köhler, Hendrik Nolte, Julian Kunkel "Secure Authorization for RESTful HPC Access". In: INFOCOMP 2022, The Twelfth International Conference on Advanced Communications and Computation. Ed. by Claus-Peter Rückemann. 2022, pp. 12–17
- Christian Köhler, Mohammed Hossein Biniaz, Sven Bingert, Hendrik Nolte, Julian Kunkel "Secure Authorization for RESTful HPC Access with FaaS Support". In: International Journal on Advances in Security 3 and 4 (2022). Ed. by Claus-Peter Rückemann, pp. 119–131
- Hendrik Nolte, Simon Sarmiento, Tim Ehlers, Julian Kunkel "A Secure Workflow for Shared HPC Systems". In: 22nd International Symposium on Cluster, Cloud and Internet Computing (CCGrid). 2022.
- Hendrik Nolte and Julian Kunkel. "Governance-Centric Paradigm: Overcoming the Information Gap between Users and Systems by Enforcing Data Management Plans on HPC-Systems". In: INFOCOMP 2023, The Thirteenth International Conference on Advanced Communications and Computation. Ed. by Claus-Peter Rückemann. 2023, pp. 13–20
- Hendrik Nolte, Nicolai Spicher, Andrew Russel, Tim Ehlers, Sebastian Krey, Dagmar Krefting, Julian Kunkel "Secure HPC: A workflow providing a secure partition on an HPC system". In: Future Generation Computer Systems 141 (2023), pp. 677–691
- Vitali Telezki, Henrik tom Wörden, Florian Spreckelsen; Hendrik Nolte, Julian Kunkel, Ulrich Parlitz, Stefan Luther, Martin Uecker, Mathias Bähr, "Deployment of an HPC-Accelerated Research Data Management System: Exemplary Workflow in HeartAndBrain Study". In: Workshop Biosignals. 2024.
- Submitted/Accepted: Hendrik Nolte, Lars Quentin, and Julian Kunkel. "Secure Elasticsearch Clusters on HPC Systems for Sensitive Data". In: HIGH PERFORMANCE COMPUTING: ISC High Performance 2024 International Workshops. Springer. 2024

- Submitted/Accepted: Hendrik Nolte, Philip Langer, Julian Kunkel. "Automatisierte Analysen von MRT-Bildern". In: KI in der Projektwirtschaft Band 2
- Submitted: Hendrik Nolte, Lars Quentin, and Julian Kunkel. "Comparing Different Encryption Workflows for Secure Elasticsearch Clusters on Shared HPC Systems for Sensitive Data". In: Journal of High-Performance Storage

Bibliography

[1] Georgii Maksimovich Adelson-Velskii and Evgenii Mikhailovich Landis. "An algorithm for organization of information ". In: *Doklady Akademii Nauk* . Vol. 146. 2. Russian Academy of Sciences. 1962, pp. 263–266.

[2] Mohammad Ahmadian et al. "SecureNoSQL: An approach for secure search of encrypted NoSQL databases in the public cloud". In: *International Journal of Information Management* 37.2 (2017), pp. 63–74.

[3] Peter Amstutz et al. "Common workflow language, v1. 0". In: *Figshare* (2016).

[4] Michael Armbrust et al. "Delta lake: high-performance ACID table storage over cloud object stores". In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 3411–3424.

[5] Michael Armbrust et al. "Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics". In: *Proceedings of CIDR*. 2021.

[6] Michael Armbrust et al. "Spark sql: Relational data processing in spark". In: *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 2015, pp. 1383–1394.

[7] Sergei Arnautov et al. "{SCONE}: Secure linux containers with intel {SGX}". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 689–703.

[8] Monya Baker. "Reproducibility crisis". In: *Nature* 533.26 (2016), pp. 353–66.

[9] Anatoliy Batyuk and Volodymyr Voityshyn. "Apache storm based on topology for real-time processing of streaming data from social networks". In: *2016 IEEE First International Conference on Data Stream Mining & Processing (DSMP)*. IEEE. 2016, pp. 345–349.

[10] Mick Bauer. "Paranoid penguin: an introduction to Novell AppArmor". In: *Linux Journal* 2006.148 (2006), p. 13.

[11] Sean Bechhofer et al. "Research objects: Towards exchange and reuse of digital knowledge". In: *Nature Precedings* (2010), pp. 1–1.

[12] Amin Beheshti et al. "Coredb: a data lake service". In: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. 2017, pp. 2451–2454.

[13] Amin Beheshti et al. "CoreKG: a knowledge lake service". In: *Proceedings of the VLDB Endowment* 11.12 (2018), pp. 1942–1945.

[14] Seyed-Mehdi-Reza Beheshti, Hamid Reza Motahari-Nezhad, and Boualem Benatallah. "Temporal provenance model (TPM): model and query language". In: *arXiv preprint arXiv:1211.5009* (2012).

[15] Seyed-Mehdi-Reza Beheshti et al. "On automating basic data curation tasks". In: *Proceedings of the 26th International Conference on World Wide Web Companion*. 2017, pp. 165–169.

[16] Khalid Belhajjame et al. *Prov-dm: The prov data model*. Tech. rep. 2013.

[17] Gordon Bell, Tony Hey, and Alex Szalay. "Beyond the data deluge". In: *Science* 323.5919 (2009), pp. 1297–1298.

[18] Richard B Berry et al. *The AASM Manual for the Scoring of Sleep and Associated Events: Rules, Terminology and Technical Specifications*. Illinois: American Academy of Sleep Medicine, 2015.

[19] Anant Bhardwaj et al. "Datahub: Collaborative data science & dataset version management at scale". In: *arXiv preprint arXiv:1409.0798* (2014).

[20] L. Biel et al. "ECG analysis: a new approach in human identification". In: *IEEE Transactions on Instrumentation and Measurement* 50.3 (2001), pp. 808–812.

[21] Sven Bingert et al. "An API to include HPC resources in workflow systems". In: *INFOCOMP 2021: The Eleventh International Conference on Advanced Communications and Computation*. 2021, pp. 15–20.

[22] Mohammad Hossein Biniaz et al. "Secure Authorization for RESTful HPC Access". In: *INFOCOMP 2022, The Twelfth International Conference on Advanced Communications and Computation* . Ed. by Claus-Peter Rückemann . 2022, 12–17.

[23] Mark S Birrittella et al. "Intel® omni-path architecture: Enabling scalable, high performance fabrics". In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE. 2015, pp. 1–9.

[24] MKABV Bittorf et al. "Impala: A modern, open-source SQL engine for Hadoop". In: *Proceedings of the 7th biennial conference on innovative data systems research*. 2015, pp. 1–10.

[25] Kai Tobias Block, Martin Uecker, and Jens Frahm. "Undersampled radial MRI with multiple coils. Iterative image reconstruction using a total variation constraint". In: *Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine* 57.6 (2007), pp. 1086–1098.

[26] Alexandra Boldyreva et al. "Order-preserving symmetric encryption". In: *Advances in Cryptology-EUROCRYPT 2009: 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings 28*. Springer. 2009, pp. 224–241.

[27] Karla AV Borges, Alberto HF Laender, and Clodoveu A Davis Jr. "Spatial data integrity constraints in object oriented geographic data modeling". In: *Proceedings of the 7th ACM international symposium on Advances in geographic information systems*. 1999, pp. 1–6.

[28] Dhruba Borthakur. "The hadoop distributed file system: Architecture and design". In: *Hadoop Project Website* 11.2007 (2007), p. 21.

[29] Peter Braam. "The Lustre storage architecture ". In: *arXiv preprint arXiv:1903.01955* (2019).

[30] Jennifer Buchmüller et al. "Extending an Open-Source Federated Identity Management System for Enhanced HPC Security". In: *The International Conference for High Performance Computing, Networking, Storage, and Analysis* (2020).

[31] Patrice Calegari, Marc Levrier, and Paweł Balczyński. "Web portals for high-performance computing: a survey". In: *ACM Transactions on the Web (TWEB)* 13.1 (2019), pp. 1–36.

[32] Capgemini SE and Pivotal Software Inc. *The Technology of the Business Data Lake*. 2013. URL: https://www.capgemini.com/wp-content/uploads/2017/07/pivotal-business-data-lake-technical_brochure_web.pdf (visited on 09/24/2023).

[33] Steven B Caudill. "The necessity of mining data". In: *Atlantic Economic Journal* 16.3 (1988), p. 11.

[34] Fay Chang et al. "Bigtable: A distributed storage system for structured data". In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), pp. 1–26.

[35] Wo L Chang and Nancy Grady. "Nist big data interoperability framework: Volume 1, definitions". In: (2019).

[36] Kyle Chard et al. "I'll take that to go: Big data bags and minimal identifiers for exchange of large, complex datasets". In: *2016 IEEE International Conference on Big Data (Big Data)* (Washington, DC, USA). IEEE, Dec. 2016, pp. 319–328. DOI: 10.1109/BigData.2016.7840618. URL: https://doi.org/10.1109/BigData.2016.7840618.

[37] Amit Chavan et al. "Towards a unified query language for provenance and versioning". In: *7th USENIX Workshop on the Theory and Practice of Provenance (TaPP 15)*. 2015.

[38] Haogang Chen et al. "Linux kernel vulnerabilities: State-of-the-art defenses and open problems". In: *Proceedings of the Second Asia-Pacific Workshop on Systems*. 2011, pp. 1–5.

[39] Lanxiang Chen et al. "Secure search for encrypted personal health records from big data NoSQL databases in cloud". In: *Computing* 102 (2020), pp. 1521–1545.

[40] Mohamed Cherradi and Anass EL Haddadi. "Data Lakes: A Survey Paper". In: *Innovations in Smart Cities Applications Volume 5*. Ed. by Mohamed Ben Ahmed et al. Cham: Springer International Publishing, 2022, pp. 823–835. ISBN: 978-3-030-94191-8.

[41] Mandy Chessell et al. "Governing and managing big data for analytics and decision makers". In: *IBM Redguides for Business Leaders* (2014).

[42] Fernando Chirigati et al. " Reprozip: Computational reproducibility with ease". In: *Proceedings of the 2016 international conference on management of data*. 2016, pp. 2085–2088.

[43] Shreyas Cholia and Terence Sun. "The NEWT platform: an extensible plugin framework for creating ReSTful HPC APIs". In: *Concurrency and Computation: Practice and Experience* 27.16 (2015), pp. 4304–4317.

[44] Jason Christopher, Gary Jung, and Christopher Doane. "Making it More Secure: The Technical and Social Challenges of Expanding the Functionality of an Existing HPC Cluster to Meet University and Federal Data Security Requirements". In: *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*. 2019, pp. 1–5.

[45] Sophie Cockcroft. " A taxonomy of spatial data integrity constraints ". In: *GeoInformatica* 1.4 (1997), pp. 327–343.

[46] Diana Coman Schmid et al. "SPHN–The BioMedIT Network: A Secure IT Platform for Research with Sensitive Human Data". In: *Digital Personalized Health and Medicine* 270 (2020), pp. 1170–1174.

[47] Peter Corbett et al. "Overview of the MPI-IO parallel I/O interface". In: *Input/Output in Parallel and Distributed Computer Systems* (1996), pp. 127–146.

[48] Julia Couto et al. "A Mapping Study about Data Lakes: An Improved Definition and Possible Architectures." In: *SEKE*. 2019, pp. 453–578.

[49] Felipe A. Cruz et al. "FirecREST: a RESTful API to HPC systems". In: *2020 IEEE/ACM International Workshop on Interoperability of Supercomputing and Cloud Technologies (SuperCompCloud)* . 2020 , 21–26 . DOI: 10.1109/SuperCompCloud51944.2020.00009.

[50] Dong Dai et al. "Lightweight provenance service for high-performance computing". In: *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2017, pp. 117–129.

[51] Koenraad De Smedt, Dimitris Koureas, and Peter Wittenburg. "FAIR digital objects for science: From data pieces to actionable knowledge units". In: *Publications* 8.2 (2020), p. 21.

[52] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[53] Barry A. Devlin and Paul T. Murphy. "An architecture for a business and information system". In: *IBM systems Journal* 27.1 (1988), pp. 60–80.

[54] Claudia Diamantini et al. "A new metadata model to uniformly handle heterogeneous data lake sources". In: *European Conference on Advances in Databases and Information Systems*. Springer. 2018, pp. 165–177.

[55] Henrik Dibowski et al. "Using Semantic Technologies to Manage a Data Lake: Data Catalog, Provenance and Access Control. " In: *SSWS@ ISWC* . 2020, pp. 65–80.

[56] J. Dixon. *Pentaho, Hadoop, and Data Lakes*. 2010. URL: https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/ (visited on 03/10/2021).

[57] James Dixon. *Data Lakes Revisited*. 2014. URL: https://jamesdixon.wordpress.com/2014/09/25/data-lakes-revisited/ (visited on 09/24/2023).

[58] Chris Dunlap. *MUNGE Uid 'N' Gid Emporium*. 2022. URL: https://dun.github.io/munge/ (visited on 03/21/2022).

[59] Simon Eismann et al. "A review of serverless use cases and their characteristics". In: *arXiv preprint arXiv:2008.11110* (2020).

[60] R Elmasri and SB Navathe. *Fundamentals of database systems*. 1994.

[61] Ronald Fagin, Amnon Lotem, and Moni Naor. "Optimal aggregation algorithms for middleware". In: *Journal of computer and system sciences* 66.4 (2003), pp. 614–656.

[62] Edgar Gabriel et al. "Open MPI: Goals, concept, and design of a next generation MPI implementation". In: *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer. 2004, pp. 97–104.

[63] Todd Gamblin et al. "The Spack package manager: bringing order to HPC software chaos". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pp. 1–12.

[64] Gartner Inc. *Gartner Says Beware of the Data Lake Fallacy*. 2014. URL: https://www.gartner.com/en/newsroom/press-releases/2014-07-28-gartner-says-beware-of-the-data-lake-fallacy (visited on 09/24/2023).

[65] Corinna Giebler et al. " The Data Lake Architecture Framework: A Foundation for Building a Comprehensive Data Lake Architecture". In: *Proceedings der 19. Fachtagung für Datenbanksysteme für Business, Technologie und Web (BTW 2021)*. 2021.

[66] Corinna Giebler et al. "A Zone Reference Model for Enterprise-Grade Data Lake Management". In: *Proceedings of the 24th IEEE Enterprise Computing Conference (EDOC 2020)*. 2020. DOI: https://doi.org/10.1109/EDOC49727.2020.00017.

[67] Corinna Giebler et al. "Modeling data lakes with data vault: practical experiences, assessment, and lessons learned". In: *International Conference on Conceptual Modeling*. Springer. 2019, pp. 63–77.

[68] GitLab. *GitLab CI/CD variables*. 2022. URL: https://docs.gitlab.com/ee/ci/variables/ (visited on 03/18/2022).

[69] Matteo Golfarelli, Dario Maio, and Stefano Rizzi. "The dimensional fact model: A conceptual model for data warehouses". In: *International Journal of Cooperative Information Systems* 7.02n03 (1998), pp. 215–247.

[70] Alex Gorelik. *The enterprise big data lake: Delivering the promise of big data and data science*. O'Reilly Media, 2019.

[71] Krzysztof J Gorgolewski et al. "The brain imaging data structure, a format for organizing and describing outputs of neuroimaging experiments". In: *Scientific data* 3.1 (2016), pp. 1–9.

[72] Shashank Gugnani, Xiaoyi Lu, and Dhabaleswar K Panda. "Swift-X: Accelerating OpenStack swift with RDMA for building an efficient HPC cloud". In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE. 2017, pp. 238–247.

[73] Rihan Hai, Sandra Geisler, and Christoph Quix. "Constance: An intelligent data lake system". In: *Proceedings of the 2016 international conference on management of data*. 2016, pp. 2097–2100.

[74] Rihan Hai, Christoph Quix, and Matthias Jarke. "Data lake concept and systems: a survey". In: *arXiv preprint arXiv:2106.09592* (2021).

[75] Rihan Hai, Christoph Quix, and Chen Zhou. "Query rewriting for heterogeneous data lakes". In: *Advances in Databases and Information Systems: 22nd European Conference, ADBIS 2018, Budapest, Hungary, September 2–5, 2018, Proceedings 22*. Springer. 2018, pp. 35–49.

[76] Michael Austin Halcrow. "eCryptfs: An enterprise-class encrypted filesystem for linux". In: *Proceedings of the 2005 Linux Symposium*. Vol. 1. 2005, pp. 201–218.

[77] Alon Halevy et al. "Goods: Organizing google's datasets". In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 795–806.

[78] Alon Y Halevy et al. "Managing Google's data lake: an overview of the Goods system." In: *IEEE Data Eng. Bull.* 39.3 (2016), pp. 5–14.

[79] Dick Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. Oct. 2012. DOI: 10.17487/RFC6749. URL: https://www.rfc-editor.org/info/rfc6749 (visited on 03/21/2022).

[80] Reihaneh H Hariri, Erik M Fredericks, and Kate M Bowers. "Uncertainty in big data analytics: survey, opportunities, and challenges". In: *Journal of Big Data* 6.1 (2019), pp. 1–16.

[81] Olaf Hartig and Jun Zhao. "Publishing and consuming provenance metadata on the web of linked data". In: *International Provenance and Annotation Workshop*. Springer. 2010, pp. 78–90.

[82] Zirije Hasani, Margita Kon-Popovska, and Goran Velinov. "Lambda architecture for real time big data analytic". In: *ICT Innovations* (2014), pp. 133–143.

[83] Michael Hausenblas and Jacques Nadeau. "Apache drill: interactive ad-hoc analysis at scale". In: *Big data* 1.2 (2013), pp. 100–104.

[84] Leonie Henschel et al. "Fastsurfer-a fast and accurate deep learning based neuroimaging pipeline". In: *NeuroImage* 219 (2020), p. 117012.

[85] Frank Herold, Sven Breuner, and Jan Heichler. *An introduction to BeeGFS*. 2014.

[86] Alan R. Hevner et al. "Design Science in Information Systems Research". In: *MIS Quarterly* 28.1 (2004), pp. 75–105. ISSN: 02767783. URL: http://www.jstor.org/stable/25148625 (visited on 12/29/2023).

[87] David E Hudak et al. "Open OnDemand: Transforming computational science through omnidisciplinary software cyberinfrastructure". In: *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale*. 2016, pp. 1–7.

[88] Bill Inmon. *Data Lake Architecture: Designing the Data Lake and avoiding the garbage dump*. Technics publications, 2016.

[89] W. H. Inmon. "The Data Warehouse and Data Mining". In: *Commun. ACM* 39.11 (1996), 49–50. ISSN: 0001-0782. DOI: 10.1145/240455.240470. URL: https://doi.org/10.1145/240455.240470.

[90] William H Inmon. *Building the data warehouse*. John Wiley & Sons, 2005.

[91] Zachary G Ives and Yi Zhang. "Dataset relationship management". In: *Proceedings of Conference on Innovative Database Systems Research (CIDR 19)*. 2019.

[92] Patrick Jattke et al. "BLACKSMITH: Scalable Rowhammering in the Frequency Domain". In: *IEEE Symposium on Security and Privacy (SP)* (2022).

[93] Aude Jegou et al. "BIDS Manager-Pipeline: A framework for multi-subject analysis in electrophysiology ". In: *Neuroscience Informatics* 2.2 (2022), p. 100072.

[94] Robert Kahn, Robert Wilensky, et al. "A framework for distributed digital object services". In: *International Journal on Digital Libraries* 6.2 (2006), pp. 115–123.

[95] David Karns, Katy Protin, and Justin Wolf. *iSSH v. Auditd: Intrusion Detection in High Performance Computing*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2012.

[96] Pradeeban Kathiravelu and Ashish Sharma. "A dynamic data warehousing platform for creating and accessing biomedical data lakes". In: *Data Management and Analytics for Medicine and Healthcare: Second International Workshop, DMAH 2016, Held at VLDB 2016, New Delhi, India, September 9, 2016, Revised Selected Papers 2*. Springer. 2017, pp. 101–120.

[97] Florian Kerschbaum. "Frequency-hiding order-preserving encryption". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015, pp. 656–667.

[98] Pwint Phyu Khine and Zhao Shun Wang. "Data lake: a new ideology in big data era". In: *ITM web of conferences*. Vol. 17. EDP Sciences. 2018, p. 03025.

[99] G. Klosh et al. "The SIESTA project polygraphic and clinical database". In: *IEEE Engineering in Medicine and Biology Magazine* 20.3 (2001), pp. 51–57. DOI: 10.1109/51.932725.

[100] Vasili Korol et al. "Introducing VIKING: A novel online platform for multiscale modeling". In: *ACS omega* 5.2 (2019), pp. 1254–1260.

[101] Jay Kreps, Neha Narkhede, Jun Rao, et al. "Kafka: A distributed messaging system for log processing". In: *Proceedings of the NetDB*. Vol. 11. 2011. Athens, Greece. 2011, pp. 1–7.

[102] Thomas S Kuhn. *The structure of scientific revolutions*. Vol. 111. Chicago University of Chicago Press, 1970.

[103] J Kunkel et al. "Establishing the io-500 benchmark". In: *White Paper* (2016).

[104] John Kunze et al. *The bagit file packaging format (v1. 0)*. Tech. rep. 2018.

[105] Gregory M Kurtzer, Vanessa Sochat, and Michael W Bauer. "Singularity: Scientific containers for mobility of compute". In: *PloS one* 12.5 (2017).

[106] Christian Köhler et al. "Secure Authorization for RESTful HPC Access with FaaS Support". In: 3 and 4 (2022). Ed. by Claus-Peter Rückemann, pp. 119–131.

[107] Dayeol Lee et al. "Keystone: An open framework for architecting trusted execution environments". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–16.

[108] Jae-Kook Lee, Sung-Jun Kim, and Taeyoung Hong. "Brute-force Attacks Analysis against SSH in HPC Multi-user Service Environment". In: *Indian Journal of Science and Technology* 9.24 (2016), pp. 1–4.

[109] Kevin Lewi and David J Wu. "Order-revealing encryption: New constructions, applications, and lower bounds ". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 1167–1178.

[110] Jingmin Li. "Design of real-time data analysis system based on Impala". In: *2014 IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA)*. IEEE. 2014, pp. 934–936.

[111] Dan Lindstedt and Kent Graziano. *Super charge your data warehouse: invaluable data modeling rules to implement your data vault*. CreateSpace, 2011.

[112] Marilex Rea Llave. "Data lakes in business intelligence: reporting from the trenches". In: *Procedia computer science* 138 (2018), pp. 516–524.

[113] Jay Lofstead, Joshua Baker, and Andrew Younge. "Data pallets: containerizing storage for reproducibility and traceability". In: *High Performance Computing: ISC High Performance 2019 International Workshops, Frankfurt, Germany, June 16-20, 2019, Revised Selected Papers 34*. Springer. 2019, pp. 36–45.

[114] Michael Lustig, David Donoho, and John M Pauly. "Sparse MRI: The application of compressed sensing for rapid MR imaging". In: *Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine* 58.6 (2007), pp. 1182–1195.

[115] Antonio Maccioni and Riccardo Torlone. "Crossing the finish line faster when paddling the data lake with kayak". In: *Proceedings of the VLDB Endowment* 10.12 (2017), pp. 1853–1856.

[116] Antonio Maccioni and Riccardo Torlone. "KAYAK: a framework for just-in-time data preparation in a data lake". In: *International Conference on Advanced Information Systems Engineering*. Springer. 2018, pp. 474–489.

[117] Cedrine Madera and Anne Laurent. "The next information architecture evolution: the data lake wave". In: *Proceedings of the 8th international conference on management of digital ecosystems*. 2016, pp. 174–180.

[118] Mark Madsen. "How to Build an enterprise data lake: important considerations before jumping in". In: *Third Nature Inc* (2015), pp. 13–17.

[119] Daniel S Marcus et al. "The Extensible Neuroimaging Archive Toolkit: an informatics platform for managing, exploring, and sharing neuroimaging data". In: *Neuroinformatics* 5 (2007), pp. 11–33.

[120] Vivien Marx. "The big challenges of big data". In: *Nature* 498.7453 (2013), pp. 255–260.

[121] Christian Mathis. "Data lakes". In: *Datenbank-Spektrum* 17.3 (2017), pp. 289–293.

[122] Frank McKeen et al. "Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave". In: *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. 2016, pp. 1–9.

[123] Robert McLay et al. "Best practices for the deployment and management of production HPC clusters". In: *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2011, pp. 1–11.

[124] Xiangrui Meng et al. "Mllib: Machine learning in apache spark". In: *The journal of machine learning research* 17.1 (2016), pp. 1235–1241.

[125] R Menolascino et al. "A realistic UMTS planning exercise". In: *Proc. 3 ACTS Mobile Communications Summit 98*. 1998.

[126] Hui Miao, Amit Chavan, and Amol Deshpande. "Provdb: Lifecycle management of collaborative analysis workflows". In: *Proceedings of the 2nd Workshop on Human-in-the-Loop Data Analytics*. 2017, pp. 1–6.

[127] Hui Miao and Amol Deshpande. "ProvDB: Provenance-enabled Lifecycle Management of Collaborative Data Analysis Workflows." In: *IEEE Data Eng. Bull.* 41.4 (2018), pp. 26–38.

[128] George A Miller. "WordNet: a lexical database for English". In: *Communications of the ACM* 38.11 (1995), pp. 39–41.

[129] Steven P Miller et al. "Kerberos authentication and authorization system". In: *In Project Athena Technical Plan*. Citeseer. 1988.

[130] Paolo Missier, Khalid Belhajjame, and James Cheney. "The W3C PROV family of specifications for modelling provenance metadata". In: *Proceedings of the 16th International Conference on Extending Database Technology*. 2013, pp. 773–776.

[131] Paolo Missier et al. "Linking multiple workflow provenance traces for interoperable collaborative science". In: *The 5th Workshop on Workflows in Support of Large-Scale Science*. IEEE. 2010, pp. 1–8.

[132] Abidalrahman Moh'd, Yaser Jararweh, and Lo'ai Tawalbeh. "AES-512: 512-bit Advanced Encryption Standard algorithm design and evaluation". In: *2011 7th International Conference on Information Assurance and Security (IAS)*. IEEE. 2011, pp. 292–297.

[133] Luc Moreau et al. "The open provenance model core specification (v1. 1)". In: *Future generation computer systems* 27.6 (2011), pp. 743–756.

[134] Kiran-Kumar Muniswamy-Reddy et al. "Provenance-aware storage systems." In: *Usenix annual technical conference, general track*. 2006, pp. 43–56.

[135] Amr A Munshi and Yasser Abdel-Rady I Mohamed. "Data lake lambda architecture for smart grids big data analytics". In: *IEEE Access* 6 (2018), pp. 40463–40471.

[136] Athira Nambiar and Divyansh Mundra. "An Overview of Data Warehouse and Data Lake in Modern Enterprise Data Management". In: *Big Data and Cognitive Computing* 6.4 (2022), p. 132.

[137] Fatemeh Nargesian et al. "Data lake management: challenges and opportunities". In: *Proceedings of the VLDB Endowment* 12.12 (2019), pp. 1986–1989.

[138] Roberto Navigli and Simone Paolo Ponzetto. "BabelNet: The automatic construction, evaluation and application of a wide-coverage multilingual semantic network". In: *Artificial intelligence* 193 (2012), pp. 217–250.

[139] Iuri D Nogueira, Maram Romdhane, and Jérôme Darmont. "Modeling data lake metadata with a data vault ". In: *Proceedings of the 22nd International Database Engineering & Applications Symposium*. 2018, pp. 253–261.

[140] Hendrik Nolte and Julian Kunkel. "Governance-Centric Paradigm: Overcoming the Information Gap between Users and Systems by Enforcing Data Management Plans on HPC-Systems". In: *INFOCOMP 2023, The Thirteenth International Conference on Advanced Communications and Computation*. Ed. by Claus-Peter Rückemann. 2023, pp. 13–20.

[141] Hendrik Nolte, Philip Langer, and Julian Kunkel. *KI in der Projektwirtschaft Band 2*. Status: Submitted/Accepted. UVK Verlag, 2024.

[142] Hendrik Nolte, Lars Quentin, and Julian Kunkel. "Comparing Different Encryption Workflows for Secure Elasticsearch Clusters on Shared HPC Systems for Sensitive Data". In: *Journal of High-Performance Storage*. Status: Submitted.

[143] Hendrik Nolte, Lars Quentin, and Julian Kunkel. "Secure Elasticsearch Clusters on HPC Systems for Sensitive Data". In: *HIGH PERFORMANCE COMPUTING: ISC High Performance 2024 International Workshops*. Status: Submitted/Accepted. Springer. 2024.

[144] Hendrik Nolte and Philipp Wieder. "Realising data-centric scientific workflows with provenance-capturing on data lakes ". In: *Data Intelligence* 4.2 (2022), pp. 426–438.

[145] Hendrik Nolte et al. "A Secure Workflow for Shared HPC Systems". In: *22nd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2022.

[146] Hendrik Nolte et al. "Secure HPC: A workflow providing a secure partition on an HPC system". In: *Future Generation Computer Systems* 141 (2023), pp. 677–691.

[147] Daniel de Oliveira et al. "An adaptive parallel execution strategy for cloud-based scientific workflows". In: *Concurrency and Computation: Practice and Experience* 24.13 (2012), pp. 1531–1550.

[148] OpenAPI Initiative. *OpenAPI Specification v3.0.0.* 2017. URL: `https://spec.openapis.org/oas/v3.0.0` (visited on 03/21/2022).

[149] OpenFaaS . *Invocations* . 2022 . URL: `https://docs.openfaas.com/architecture/invocations/` (visited on 12/13/2022).

[150] Andrew Oram. *Managing the Data Lake: Moving to Big Data Analysis*. O'Reilly Media, 2015.

[151] Arvind Panwar et al. "A Blockchain Framework to Secure Personal Health Record (PHR) in IBM Cloud-Based Data Lake". In: *Computational Intelligence and Neuroscience* 2022 (2022).

[152] Thorsten Papenbrock et al. "Data profiling with metanome". In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1860–1863.

[153] P Patel, G Wood, and A Diaz. "Data lake governance best practices". In: *The DZone Guide to Big Data-Data Science & Advanced Analytics* 4 (2017), pp. 6–7.

[154] Cesare Pautasso and Gustavo Alonso. "Parallel computing patterns for grid workflows". In: *2006 Workshop on Workflows in Support of Large-Scale Science*. IEEE. 2006, pp. 1–10.

[155] Brian Pawlowski et al. "The NFS version 4 protocol". In: *In Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000*. Citeseer. 2000.

[156] Ken Peffers et al. "A design science research methodology for information systems research". In: *Journal of management information systems* (2007), pp. 45–77.

[157] Pedro F Pérez-Arteaga et al. "Cost comparison of lambda architecture implementations for transportation analytics using public cloud software as a service". In: *Special Session on Software Engineering for Service and Cloud Computing* (2018), pp. 855–862.

[158] Pierre Peterlongo et al. "Lossless filter for finding long multiple approximate repetitions using a new data structure, the bi-factor array". In: *International Symposium on String Processing and Information Retrieval*. Springer. 2005, pp. 179–190.

[159] Gregory F Pfister. "An introduction to the infiniband architecture". In: *High performance mass storage and parallel I/O* 42.617-632 (2001), p. 10.

[160] Karl Popper. *The logic of scientific discovery*. Routledge, 2005.

[161] Klaas P Pruessmann et al. "SENSE: sensitivity encoding for fast MRI". In: *Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine* 42.5 (1999), pp. 952–962.

[162] Christoph Quix, Rihan Hai, and Ivan Vatov. "GEMMS: A Generic and Extensible Metadata Management System for Data Lakes." In: *CAiSE forum*. Vol. 129. 2016.

[163] Raghu Ramakrishnan et al. "Azure data lake store: a hyperscale distributed file service for big data analytics". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, pp. 51–63.

[164] Alessandro Randazzo and Ilenia Tinnirello. "Kata containers: An emerging architecture for enabling mec services in fast and secure way". In: *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. IEEE. 2019, pp. 209–214.

[165] Franck Ravat and Yan Zhao. "Data lakes: Trends and perspectives". In: *International Conference on Database and Expert Systems Applications*. Springer. 2019, pp. 304–313.

[166] Ronald L Rivest, Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[167] Pegdwendé Sawadogo and Jérôme Darmont. "On data lake architectures and metadata management". In: *Journal of Intelligent Information Systems* 56 (2021), pp. 97–120.

[168] Pegdwendé N Sawadogo et al. "Metadata systems for data lakes: models and features". In: *European conference on advances in databases and information systems*. Springer. 2019, pp. 440–451.

[169] SchedMD. *Slurm REST API*. 2022. URL: `https://slurm.schedmd.com/rest.html` (visited on 03/18/2022).

[170] Michel Scheerman et al. "Secure platform for processing sensitive data on shared HPC systems". In: *arXiv preprint arXiv:2103.14679* (2021).

[171] Frank B Schmuck and Roger L Haskin. "GPFS: A Shared-Disk File System for Large Computing Clusters." In: *FAST*. Vol. 2. 19. 2002, pp. 231–244.

[172] Etienne Scholly et al. "Coining goldMEDAL: a new contribution to data lake generic metadata modeling". In: *arXiv preprint arXiv:2103.13155* (2021).

[173] Erik Schultes and Peter Wittenburg. "FAIR Principles and Digital Objects: Accelerating convergence on a data infrastructure". In: *Data Analytics and Management in Data Intensive Domains: 20th International Conference, DAMDID/RCDL 2018, Moscow, Russia, October 9–12, 2018, Revised Selected Papers 20*. Springer. 2019, pp. 3–16.

[174] Philip Schwan et al. "Lustre: Building a file system for 1000-node clusters". In: *Proceedings of the 2003 Linux symposium*. Vol. 2003. 2003, pp. 380–386.

[175] Ulrich Schwardmann. "Digital objects–FAIR digital objects: Which services are required?" In: *Data Science Journal* 19.1 (2020).

[176] Omar Sefraoui, Mohammed Aissaoui, Mohsine Eleuldj, et al. "OpenStack: toward an open-source solution for cloud computing". In: *International Journal of Computer Applications* 55.3 (2012), pp. 38–42.

[177] Raghav Sethi et al. "Presto: SQL on everything". In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE. 2019, pp. 1802–1813.

[178] Ben Sharma. *Architecting data lakes: data management architectures for advanced business use cases*. O'Reilly Media, 2018.

[179] Shekhar Shashi and Chawla Sanjay. "Spatial databases: A tour". In: *Upper Saddle River, New Jersey* 7458 (2003).

[180] Smadar Shilo, Hagai Rossman, and Eran Segal. "Axes of a revolution: challenges and promises of big data in healthcare". In: *Nature medicine* 26.1 (2020), pp. 29–38.

[181] David W Shimabukuro et al. "Effect of a machine learning-based severe sepsis prediction algorithm on patient survival and hospital length of stay: a randomised clinical trial". In: *BMJ open respiratory research* 4.1 (2017).

[182] Arun Kumar Singh and Samidha Dwivedi Sharma. "High Performance Computing (HPC) Data Center for Information as a Service (IaaS) Security Checklist: Cloud Data Governance." In: *Webology* 16.2 (2019), pp. 83–96.

[183] Amit Singhal. "Introducing the knowledge graph: things, not strings". In: *Official google blog* 5 (2012), p. 16.

[184] Tyler J Skluzacek, Kyle Chard, and Ian Foster. "Klimatic: a virtual data lake for harvesting and distribution of geospatial data". In: *2016 1st Joint International Workshop on Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS)*. IEEE. 2016, pp. 31–36.

[185] Stephen Smalley, Chris Vance, and Wayne Salamon. "Implementing SELinux as a Linux security module". In: *NAI Labs Report* 1.43 (2001), p. 139.

[186] Austin Smith et al. "Exploring Untrusted Distributed Storage for High Performance Computing". In: *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*. 2019, pp. 1–6.

[187] Jens B Stephansen et al. "Neural network analysis of sleep stages enables efficient diagnosis of narcolepsy". In: *Nature Communications* 9.5229 (2018). DOI: 10.1038/s41467-018-07229-3.

[188] Cathie Sudlow et al. "UK biobank: an open access resource for identifying the causes of a wide range of complex diseases of middle and old age". In: *PLoS medicine* 12.3 (2015), e1001779.

[189] Snezhana Sulova et al. "The Usage of Data Lake for Business Intelligence Data Analysis". In: *International Conference Information and communication technologies in business and education*. Vol. 18. 2019, pp. 135–144.

[190] Isuru Suriarachchi and Beth Plale. "Crossing analytics systems: a case for integrated provenance in data lakes". In: *2016 IEEE 12th International Conference on e-Science (e-Science)*. IEEE. 2016, pp. 349–354.

[191] Isuru Suriarachchi and Beth Plale. "Provenance as essential infrastructure for data lakes". In: *International Provenance and Annotation Workshop*. Springer. 2016, pp. 178–182.

[192] Isuru Suriarachchi, Quan Zhou, and Beth Plale. "Komadu: A capture and visualization system for scientific data provenance". In: *Journal of Open Research Software* 3.1 (2015).

[193] Vaclav Svaton et al. "HPC-as-a-Service via HEAppE Platform". In: *Conference on Complex, Intelligent, and Software Intensive Systems*. Springer. 2019, pp. 280–293.

[194] V. Telezki et al. "Deployment of an HPC-Accelerated Research Data Management System: Exemplary Workflow in HeartAndBrain Study". In: *Workshop Biosignals*. 2024. DOI: https://doi.org/10.47952/gro-publ-204.

[195] Ashish Thusoo et al. "Hive-a petabyte scale data warehouse using hadoop". In: *2010 IEEE 26th international conference on data engineering (ICDE 2010)*. IEEE. 2010, pp. 996–1005.

[196] Ashish Thusoo et al. "Hive: a warehousing solution over a map-reduce framework". In: *Proceedings of the VLDB Endowment* 2.2 (2009), pp. 1626–1629.

[197] Chia-Che Tsai, Donald E Porter, and Mona Vij. "Graphene-sgx: A practical library {OS} for unmodified applications on {SGX}". In: *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 2017, pp. 645–658.

[198] M Uecker and M Lustig. *BART toolbox for computational magnetic resonance imaging, .* 2016. DOI: 10.5281/zenodo.592960.

[199] Martin Uecker et al. "ESPIRiT—an eigenvalue approach to autocalibrating parallel MRI: where SENSE meets GRAPPA ". In: *Magnetic resonance in medicine* 71.3 (2014), pp. 990–1001.

[200] Massimo Villari et al. "AllJoyn Lambda: An architecture for the management of smart environments in IoT". In: *2014 International Conference on Smart Computing Workshops*. IEEE. 2014, pp. 9–14.

[201] Deepak Vohra. "Apache parquet". In: *Practical Hadoop Ecosystem*. Springer, 2016, pp. 325–335.

[202] Denny Vrandečić. "Wikidata: A new platform for collaborative data collection". In: *Proceedings of the 21st international conference on world wide web*. 2012, pp. 1063–1064.

[203] Coral Walker and Hassan Alrehamy. "Personal data lake with data gravity pull". In: *2015 IEEE Fifth International Conference on Big Data and Cloud Computing*. IEEE. 2015, pp. 160–167.

[204] Li Wang et al. "Optimizing eCryptfs for better performance and security". In: *Linux Symposium*. Citeseer. 2012, p. 137.

[205] Min Wang, Jiankun Hu, and Hussein A. Abbass. "BrainPrint: EEG biometric identification based on analyzing brain connectivity graphs". In: *Pattern Recognition* 105 (2020), p. 107381. ISSN: 0031-3203.

[206] James Warren and Nathan Marz. *Big Data: Principles and best practices of scalable realtime data systems*. Simon and Schuster, 2015.

[207] Sage A Weil et al. "Ceph: A scalable, high-performance distributed file system". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, pp. 307–320.

[208] Philipp Wieder and Hendrik Nolte. "Toward data lakes as central building blocks for data management and analysis". In: *Frontiers in big Data* 5 (2022), p. 945720.

[209] M Wilkinson, M Dumontier, and Aalbersberg. "The FAIR Guiding Principles for scientific data management and stewardship". In: *Scientific Data 3* (2016). URL: https://doi.org/10.1038/sdata.2016.18.

[210] Ming-Chuan Wu and Alejandro P Buchmann. "Research issues in data warehousing". In: *Datenbanksysteme in Büro, Technik und Wissenschaft*. Springer. 1997, pp. 61–82.

[211] Chao-Tung Yang et al. "The implementation of data storage and analytics platform for big data lake of electricity usage with spark". In: *The Journal of Supercomputing* 77.6 (2021), pp. 5934–5959.

[212] Tatu Ylonen. "SSH - Secure Login Connections Over the Internet". In: *Proceedings of the 6th USENIX Security Symposium (USENIX Security 96)*. San Jose, CA: USENIX Association, July 1996, pp. 37–42. URL: https://www.usenix.org/conference/6th-usenix-security-symposium/ssh-secure-login-connections-over-internet.

[213] Zhihao Yuan et al. "Utilizing provenance in reusable research objects". In: *Informatics*. Vol. 5. 1. Multidisciplinary Digital Publishing Institute. 2018, p. 14.

[214] Z. Wang et al. "RS-YABI: A workflow system for Remote Sensing Processing in AusCover". In: *Proceedings of the 19th International Congress on Modelling and Simulation*. MODSIM 2011 - 19th International Congress on Modelling and Simulation - Sustaining Our Future: Understanding and Living with Uncertainty. 2011, 1167–1173.

[215] Matei Zaharia et al. "Apache spark: a unified engine for big data processing". In: *Communications of the ACM* 59.11 (2016), pp. 56–65.

[216] Matei Zaharia et al. "Spark: Cluster computing with working sets". In: *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*. 2010.

[217] X Zhang et al. "Process and outcome for international reliability in sleep scoring". In: *Sleep and Breathing* 19 (1 2015), pp. 191–5.

[218] Yi Zhang and Zachary G Ives. "Juneau: data lake management for Jupyter". In: *Proceedings of the VLDB Endowment* 12.12 (2019).

[219] Paul Zikopoulos. *Big data beyond the hype: A guide to conversations for today's data center*. McGraw-Hill Education, 2015.

Declaration on the Use of AI

In this thesis, I have used ChatGPT or another AI as follows:

☐ not at all

☐ during brainstorming

☐ when creating the outline

☐ to write individual passages, altogether to the extent of ____% of the entire text

☐ for the development of software source texts

☐ for optimizing or restructuring software source texts

☐ for proofreading or optimizing

■ further, namely: Development of individual functions used within the analysis and plotting of the *Secure Metadata Management* presented in Chapter 8.

I hereby declare that I have stated all uses completely. Missing or incorrect information will be considered as an attempt to cheat.

Date and Signature _____

Hendrik Nolte