FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

David Košťál

Complex experiment support through the IVIS framework

Department of Distributed and Dependable Systems

Supervisor of the master thesis: prof. RNDr. Tomáš Bureš, Ph.D.
Study programme: Software Systems

Prague 2025

I declare that I carried out this master thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I would like to thank my supervisor, Prof. RNDr. Tomáš Bureš, Ph.D., for allowing me to participate in a project he is involved in. It was a great experience, and I am grateful for it. I would also like to thank him for the consultations.

I would also like to thank the team from Charles University working on the ExtremeXP for giving me their time for project discussion.

I would like to thank my wife for being here for me.

I would also like to thank my parents and other relatives for creating the environment for me to study and for pushing me when needed.

Chtěl bych poděkovat mému vedoucímu, Prof. RNDr. Tomáš Bureš, Ph.D., že mi umožnil se podílet na projektu, kterým se také zabývá. Byla to skvělá zkušenost za kterou jsem vděčný. Také bych mu chtěl poděkovat za konzultace.

Také bych chtěl poděkovat týmy z Univerzity Karlovy, který pracoval na projektu ExtremeXP za to, že mi dali jejich čas pro diskuzi projektu.

Chtěl bych poděkovat mé manželce za to, že tu pro mě byla.

Také bych chtěl poděkovat mým rodičům a dalším příbuzným, za to že mi vytvořili prostředí vhodné pro vzdělávání a postrčili když to bylo potřeba.

Title: Complex experiment support through the IVIS framework

Author: David Košťál

Department: Department of Distributed and Dependable Systems

Supervisor: prof. RNDr. Tomáš Bureš, Ph.D., Department of Distributed and Dependable Systems

Abstract: Data experimentation is complex and consists of multiple steps, such as data gathering and workflow design. Because of this, an experiment management framework is needed. This thesis is part of the ExtremeXP project, which aims to implement a modern and versatile experiment management framework. The framework relies on a data storage system to store data from the experiments, which are often big and of various types and do not have a fixed schema. This thesis designs and implements such a data storage system based on partners' requirements with a focus on scalability, high performance, and traceability, which was previously not sufficiently addressed by experimentation management frameworks. This includes selecting the optimal database and proposing a data access layer with an API, which enables efficient querying of stored data yet is not too broad. Docker was chosen as the optimal deployment method.

Keywords: Experiments, API, Data access layer, Elastic search

Název práce: Komplexní podpora experimentů s využitím IVIS frameworku

Autor: David Košťál

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: prof. RNDr. Tomáš Bureš, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Datové experimentování je complexní proces, který se skládá z různých kroků jako například sběr dat a vytváření postupu. Z tohoto důvodu jsou potřeba nástroje pro řízení experimentů. Tato diplomová práce je součástí projektu ExtremeXP, který si dává za cíl vytvořit moderní a všestraný nástroj pro řízení experimentů. Tento nástroj závisí na systému datového úložiště, který ukládá data z provedených experimentů. Data z experimentů mohou být velká a nemusí mít předem známé schéma.Tato diplomová práce vytváří návrh a implementaci takového systému datového úložiště. Návrh je postaven na sběru požadavků od partnerů a je kladen důraz na škálovatelnost, vysoký výkon a sledovatelnost, což dříve nebylo dostatečně adresováno nástroji pro řízení experimentů. Práce obsahuje výběr optimální databáze a návrh přístupové vrstvy k datům, včetně návrhu API. Návrh API byl vytvářen tak, aby bylo umožněno efektivní dotazování, a zároveň aby API nebylo příliš rozsáhlé. Jako optimální metoda pro nasazení byl vybrán Docker.

Klíčová slova: Experimenty, API, Přístupová vrstva k datům, Elastic search

Contents

1 Introduction

With machine learning on the rise, data management and analysis are becoming increasingly important. As machine learning models are trained for various problems, previously unthinkable achievements are reached.

However, it can be challenging to train an optimal model for the given problem. First, data needs to be collected and processed, and then a model is trained. The performance is evaluated based on relevant metrics (e.g., accuracy). The performance of models differs based on the hyperparameters used to train them (e.g., learning rate). There is no universal solution when selecting hyperparameter values; thus, finding the optimal model may lead to extensive experimentation.

In such cases, the experimentation encompasses multiple experiments with various configurations. Each configuration consists of a different set of input datasets, parameters, and computed metrics. It may also differ in the algorithm used, which may be a sequence of complex workflows. This can easily lead to losing track of the origin of the results, and the experimentation might need to be partially or entirely repeated. However, repeating experiments usually requires many manhours and expensive computing power. In the worst cases, the configuration leading to an optimal model might even be completely forgotten.

Experiments are also often part of larger projects, and experiment results must be shared amongst the project participants. This brings collaboration challenges, as the experiment specification, including input datasets, parameters, and used algorithms, must be provided to share information about an experiment completely.

For these reasons, various tools for data experimentation were developed. These tools aim to make experimentation easier by helping researchers with the experimentation process. Big developments are happening, especially in AutoML, a research area targeting the automation of machine learning. These tools aim to make machine learning less reliant on machine learning experts. This includes automating data preparation, constructing features, and optimizing hyperparameters. If deep learning is used, it can even include designing the topology of neural networks. An example of such a tool is autoWEKA, which extends the WEKA [1] package, automatically selecting the machine learning algorithm and its hyperparameters using Bayesian optimization [1]. Another example is Auto-PyTorch, which is based on the PyTorch [2] framework and focuses on deep learning. Auto-PyTorch optimizes both hyperparameters and network architecture.

However, commonly used tools and frameworks still lack sufficient attention to some problems, including ensuring traceability and building a knowledge base. By storing experiment information (e.g., how long the model was trained or what the accuracy was), a knowledge base is built and can then be used to optimize experiments on similar datasets and hyperparameters. ExtremeXP (Horizon Europe) [2] [3], to which this master's thesis contributes, focuses on those areas.

[1]WEKA is an open source java package containing various machine learning algorithms for data mining. WEKA was developed at the University of Waikato in New Zealand.

[2]PyTorch is a python machine learning library used in applications involving image recognition or natural language processing. It was developed at the Meta AI research lab.

It provides versatile support for each experimentation process step, including designing an experiment, implementing and running the experiment, and analyzing its results. It also differs from other experimentation frameworks by providing adaptive planning of experiments, which helps the experimentator select parameter values during an experiment. Chapter 2 provides a more detailed description of ExtremeXP, including its architecture and metamodel.

ExtremeXP is a complex framework that is being developed by partners from multiple universities. It is divided into several parts (e.g., knowledge management, experiment modeling, or experiment execution), each developed by a different team. Each part takes inputs and produces outputs. Each team may imagine passed data differently. Thus, for the framework to be developed, frequent communication between teams must be maintained, and expectations must be synchronized. One of the data used in the framework is data about performed experiments.

The goal of this thesis is to design and implement a data service to support the storage and retrieval of data about performed experiments. This data service provides optimized support for viewing experiment results, comparing different approaches to an experiment, and ensuring the traceability of the resulting data. A strong focus is laid on storing the metrics and experiment metadata. Metadata (e.g., hardware specification) is essential for repeatability and analyzing data as it provides a broader context and may provide additional reasoning for the origin of outliners.

The data service is composed of two main parts which need to be designed and implemented. The first part is a data storage system, which stores the data from the experiments. The requirement for the system was that it must be universal enough to store data from any type of experiment, including their metrics. Metrics can produce data of various kinds, and it was essential to select a suitable database capable of storing entries of each data type and effective enough even for large data sets, such as time series created by frequent sampling. SQL and NoSQL databases needed to be compared to select the best-suited database for our purposes. The design of the data storage system is described in Chapter 3, and the technical realization is described in Chapter 7.

The second part of the data service is the data access layer. The data access layer abstracts interactions with the database, separating the database from the rest of the framework. This has multiple benefits, including simplifying data operations and hiding database-specific details. It also ensures data consistency and provides additional safety for the system from potential attacks. Building the data access layer includes gathering requirements from the partners and using those requirements to design and implement the application programming interface (API).

It is essential to design and implement an API that enables partners to query our data efficiently. One of the goals during the API design was to provide callees with all relevant data with as few API requests as possible while not sending more data than needed. Another goal was to propose an API that would be neither too broad nor overly complicated. Those two goals needed to be balanced to provide the optimal API.

For this reason, various data retrieval scenarios were designed based on partners' requirements. The scenarios were then divided into categories. The API was designed based on those data retrieval scenarios to optimize data retrieval. Chapter

4 describes the API's design, and Chapter 8 describes its technical realization, including the selection of the API protocol.

Due to existing project constraints, we decided to use the existing IVIS project as a base for the data service. IVIS was developed at Charles University and provides core components and models needed for the development of modern web applications for data visualization. IVIS contains client and server modules. The client uses ReactJS for the user interface. The server is built on Node.js. However, this project has not been maintained for a few years, so refactoring needed to be done to catch up with the development of the dependencies. The React library had substantial development. For example, hooks were introduced, completely changing the building and organizing components in React. The IVIS framework and the refactoring needed to bring the framework to a state suitable for ExtremeXP are described in Chapter 5.

After designing and implementing the data storage system and the data access layer, an effective deployment solution selection was fundamental to ensure scalability. This is one of the reasons why Docker was selected. Docker uses containerization, as it packs the application and its dependencies into a container that can be run on different platforms, ensuring portability. Compared to traditional virtual machines, Docker containers are lightweight and faster to start and shut down. Also, Docker is becoming increasingly popular, providing an active community and promising long-term viability as Docker is to be maintained for the foreseeable future. The deployment is discussed in Chapter 9.

2 ExtremeXP

The data service this master thesis designs and implements is a part of the ExtremeXP project (Horizon Europe). The goal of the thesis is to propose a data service capable of storing data about executed workflows, metrics, and metadata. For that, the context of the whole project needs to be known first. This includes understanding different parts of the system, connections between them, data used, and data produced. This chapter contains a description of the work done before this thesis.

2.1 Introduction to ExtremeXP

ExtremeXP helps users avoid relying on static or rigid models and instead use experimentation as the primary driver for analytics. The user iteratively experiments with the data, gradually revealing insights. This is what we call experiment-driven analytics.

ExtremeXP differs from other experimentation frameworks in that it proposes a human-in-the-loop experimentation approach. This approach puts the user at the center of the analytic processes. This maximizes the collaboration between the system and the user at each step, from data collection to decision-making. Based on observed user interactions, knowledge is built, and the system is trained to come up with complex analytic workflows that correspond to the user's needs and help them make decisions.

Users have different levels of understanding of the domain and expect results with different levels of detail. By storing the knowledge base, we get a user context. This enables the creation of more fit-for-purpose outcomes based on user level and needs, maximizing the user's takeaway from the experiment.

This approach helps critical domains such as crisis management effectively make decisions, which otherwise may not be possible due to a dramatic increase in data production.

Modern data analytic solutions usually work with big data and with data of different modalities. For example: IoT devices collect data by periodical sampling via their sensors, creating time-series data. This data can be quite large, based on sampling frequency and the number of parameters measured. Social media companies are developing personalized ad targeting by storing all available user data. This can create large matrices or data cubes. Speech processing uses audio recordings. Also, the system might work with video data, for example, for surveillance systems, and based on the quality of the video, the data size can dramatically increase.

ExtremeXP aims to work with data of various modalities, various sizes and from different owners.

Despite these challenges and the fact that data can be of low quality, the ExtremeXP aims to provide insights that will be accurate and trustworthy. In order to provide trustworthiness, the system provides traceability, which this thesis designs and implements. To achieve traceability, each output dataset and computed metric must provide a trace, leading to input datasets, parameters, and algorithms used. This does not mean that data can be created again exactly the same because experiments often involve some kind of heuristics. The idea is that the metrics (e.g., accuracy) will usually have approximately the same results if an experiment is repeated with the same input datasets, parameters, and algorithms.

2.2 Architecture

Figure 2.1 captures ExtremeXP's architecture. The main components are knowledge management, experiment modeling, experiment execution, and user interaction.

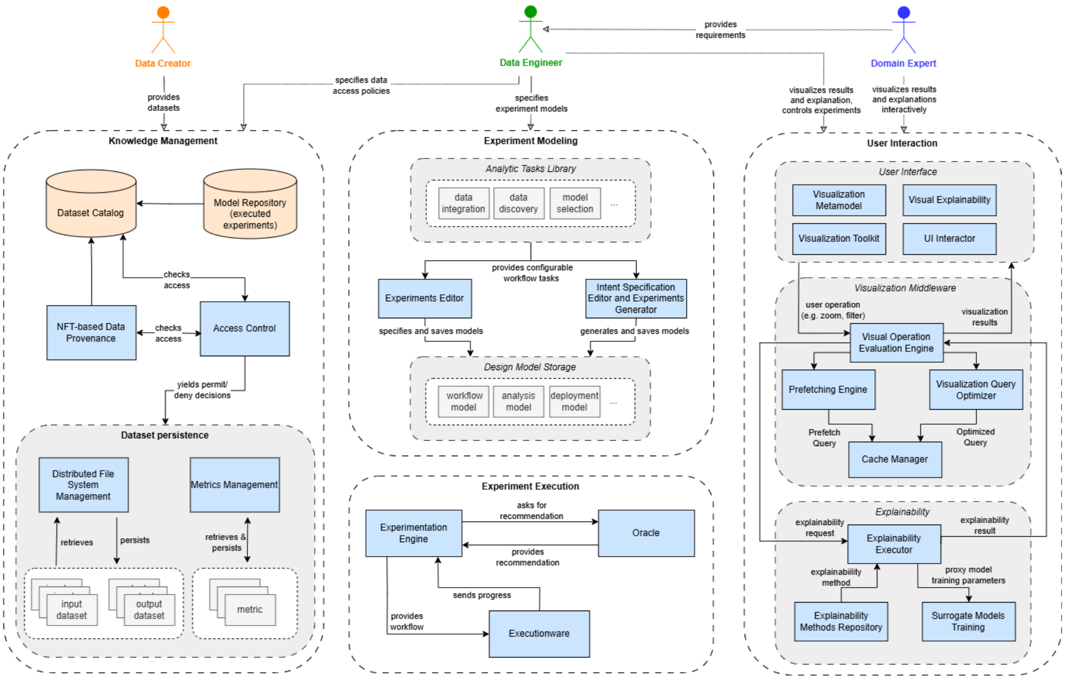

Figure 2.1 System architecture from ExtremeXP D2.1. The main components are knowledge management, experiment modeling, experiment execution, and user interaction.

2.2.1 Experiment Modeling Component

The experiment starts with a domain expert who provides requirements to a data engineer. The data engineer then uses those requirements to specify experiment models using the experiment editor provided by the experiment modeling component. The models are stored in EMF-based design model storage. The component also enables the specification of an intent.

2.2.2 Experimentation Execution Component

The experimentation execution component retrieves and uses stored workflow models, workflow metrics, and metadata. This component includes the experimentation engine, which leads the experimentation process. The experimentation engine improves the experimentation by selecting the optimal workflow variant using knowledge from previous experiments and having the user in the loop [4].

The experimentation engine then creates a deployed workflow that does not contain any unassigned variables. A recommendation from Oracle can be requested for assigning values to variables.

The deployed workflow is then provided to the Executionware, which executes the experiment. Afterward, it informs the experimentation engine about the progress, which may be based on user feedback.

2.2.3 UI component

The user is included in this process via the UI component, which enables the users (Data engineers or domain experts) to view data, analyze it using a visualization framework for different graph types, and influence the experiment by giving feedback on the experiment's progress.

2.2.4 Knowledge Management Component

The last component is knowledge management. This component enables the Experiment execution component to access input datasets from or store output datasets in distributed file system management. The access control system is blockchain-based and attribute-based, providing a decentralized and secure access system. The access policies are provided by a domain expert and specified by the data engineer. Datasets are provided by the data creator (e.g., a user of an app). Experiment metadata, metrics, and dataset metadata are saved in their repositories inside the component.

2.3 Metamodel of workflow package

Figure 2.2 contains the meta-model of the workflow package, which was created in order to fully understand the workflow of an experiment.

2.3.1 Workflow class

The main element is the workflow class. A workflow may or may not be complete as it may contain references to abstract tasks, which need to be specified for deployed workflow. The design makes workflows composable as they can be combined to form a more complex system. Two views are possible. First, composing from bottom to top means that small tasks can be joined as a more complex workflow. For example, smaller tasks, such as fetching data from multiple sources and preprocessing them, can be merged as a data-preparation workflow. The second view is composing from top to bottom, which means dividing a complex workflow into smaller, possibly abstract, tasks. For example, model training workflow can be divided into batch preparation, batch prediction computing, loss computation, gradient computation, and parameter update. Complex tasks can be modeled as sub-workflow because workflow extends tasks,

A workflow is composed of multiple nodes and links. A node is a common parent for an action, which may be an event (e.g., start or end), task, or operator.

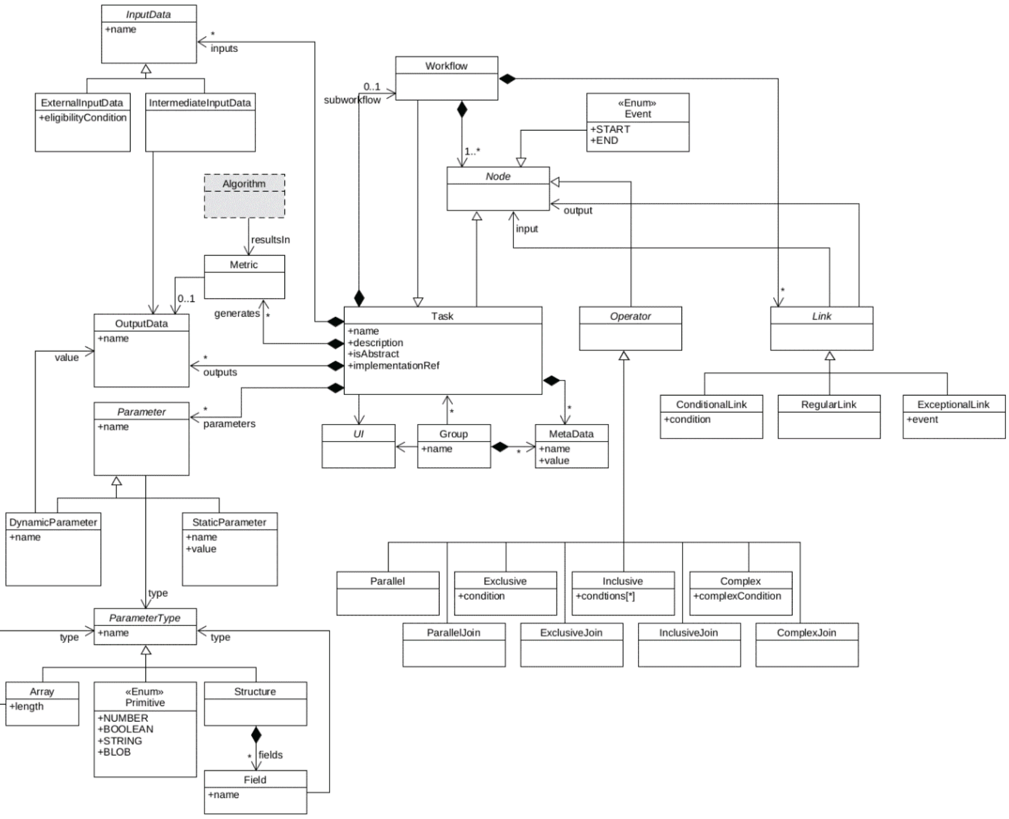

Figure 2.2 The metamodel of workflow package, depicting relationships between tasks, workflows, datasets, and parameters.

2.3.2 Operators and Links

An operator enables the workflow to be branched. Two tasks can run sequentially or in parallel and have an exclusive or inclusive relationship. Based on more advanced conditions, more complex branching can be modeled.

On the other hand, a link is a connection between nodes (tasks or decision points). The connections are oriented. One node is an input, and another one is an output. Links enable the definition of the overall flow of the workflow.

There are three types of links. The first one is the regular link, which is a connection between two nodes without any condition. The second type is the conditional link, where the condition needs to be true to the path to be followed. The last type is the exceptional link, which handles exceptions and unexpected situations.

2.3.3 Tasks and Parameters

The task class represents an individual task in the workflow. It may take multiple data input datasets and multiple parameters. Apart from static parameters, which are specified by name and value, a task can also rely on dynamic parameters, which are obtained by another. A task can be simple, in which case it is defined by descriptor, or complex, in which case it is modeled as a sub-workflow. A task can be abstract (modeled by configuration). Group class allows horizontal grouping of tasks, and UI class enables the user to interfere with tasks.

Tasks take parameters, and parameters may be of various types. ParameterType models type possibilities, as a parameter may have a primitive type, be an array or a structure. The model is recursive, enabling a parameter to be a structure with a field of an array, where each element is an integer, for example.

2.4 Use cases

The ExtremeXP is designed to provide general support for any kind of experiment.

One of the use cases is a prediction of urban flash flooding [5]. The data needed for the predictions are complex, variable, and sometimes only partially available. Various data are needed to predict flash floods successfully. Digital elevation models and rainfall time series are essential. Furthermore, accuracy can be improved by identifying ground elements such as buildings and roads. Soil types are also advantageous, as they enable the correct prediction of the absorbency of the soil. The data may be large, as the finer the scale, the better the results. The use of AI can increase forecasting accuracy even with partial data.

The second use case is failure prevention for the manufacturing industry. During this use case, problems are analyzed, and their cause is searched for. The goal of this use case is to detect severe anomalies before they lead to critical machine component failure. For this purpose, high-frequency raw data are collected, and an ML algorithm is trained to detect the anomalies. A high sampling rate leads to more fine-grained insights and captures small variations, but it also leads to enormous time series. Data analysts with machine operators may specify an intent and asset description to generate the anomaly detection workflow automatically. This use case relies on the human-in-the-loop paradigm as the data analysts and machine operators may examine the indicators from the workflows and give feedback, which is used to perfect the accuracy.

Other use cases include cybersecurity situation awareness, situational intelligence for PPDR, and Flexible transportation (MOBY).

3 Design of the data storage system

This chapter contains thoughts from ExtremeXP meetings with partners or within the Charles University ExtremeXP team. The thesis contributed with the division of data into data categories, the creation of the data model, and the further analysis of traceability and reproducibility.

One of the goals of this thesis is for the ExtremeXP project to store data about executed experiments. The broader context of the experimentation needs to be analyzed to propose and design the optimal data service. Discussions with partners were necessary to ensure that the resulting data service would best fit their purposes.

This includes analyzing the steps of an experiment and various categories of tasks and workflows. This results in the sense of stored data, their types, and volume. Also, the relationship between the data is needed to know which data will likely be retrieved together. Data include computed metrics, which also need to be explored and divided into categories to ensure that the data service will enable efficient storage of any kind of metrics. This analysis of the system's data leads to the creation of a data schema. The data schema was created with the focus on traceability.

This chapter describes the analysis that led to the data model proposal. The data model describes data connected with executed experiments and the relations between them. The information from the data model was used during API design and data service implementation.

3.1 Experiment context

Each experiment consists of workflows. For example, a machine learning experiment may contain the following workflows: dataset preparation, data augmentation, model training, evaluation, and deciding modifications. An experiment may be recursive (e.g., a predictor is trained based on its prediction).

Each workflow takes a set of input datasets and parameters. During its run, it produces output datasets and computes specified metrics. A workflow consists of tasks that enable the division of behavior into smaller parts. If a task is still too large, it can be modeled as a workflow and divided into tasks as well. A task can also have its input datasets and parameters. It also generates output datasets and computes metrics. A dataset generated by one task can be used by another task within the executed workflow or even across the executed workflows.

Each entity also needs to store its metadata. Metadata for an executed experiment may be its type, for example. Metadata enables grouping experiments with the same characteristics. Metadata may capture the specification of the hardware hosting the experiment or allocated resources in the case of cloud computing. Software information, such as versions of languages and libraries, can also be captured as metadata. The data service enables the storage of the metadata and queries based on them.

The system needs to expect any kind of task. Some tasks are completed after running only once, e.g., processing large volumes (like 1TB) of images. These are typically computationally expensive. Other tasks may require running repeatedly multiple times, e.g., benchmarking an approach on different hardware. In this case, output datasets would not be analyzed; rather, the aggregation results of hardware resources would be the main focus.

The user feedback is wanted after the experiment is run or even in the middle of an experiment. Analysis is the ultimate goal. In order to provide an analysis, we need to store metrics. Metrics also have various kinds. One kind is sampling, producing a time series, e.g., of used computational resources. One run of a task may produce billions of data points based on sampling frequency and the number of sampled factors. Other types of metrics may result in one number (e.g., precision), a vector (e.g., precision of the approach for different categories), or not interpreted binary large object (BLOB). These types of metrics do not produce many data points, but the result may still be quite large.

Metrics could be considered a subtype of datasets. Merging all results into inputs would benefit output datasets and metrics by creating a more concise description. A task would simply generate outputs without the additional description of the division between metrics and outputs. Also, the API would be simpler. However, this simplification would also lead to hiding necessary information.

3.2 Data categories

To correctly propose the data service, we need to know which data will be stored. The stored data can be divided into datasets, information about users, and information about experiments. Figure 3.1 shows an overview of the stored data. It focuses on metric categorization and provides examples.

3.2.1 Datasets

Tasks use datasets as inputs and produce them as outputs. Datasets are stored in the distributed file system. ExtremeXP aims to work with data of various modalities from various owners. This thesis focuses on executed experiments and metadata; thus, further analysis of datasets is out of the scope. Each executed experiment, workflow, and task should provide a link to the input and output datasets. After discussing this with the partners, it was decided that this link is to be a Uniform Resource Identifier (URI), enabling executed workflows to refer to datasets stored outside of the framework.

3.2.2 Experiment data

Data about an experiment contains the experiment specification and metrics. An experiment specification includes inputs, parameters, and algorithms used. Metrics can be divided into two groups: task-independent and task-dependent. Task-independent metrics do not know the inner workings of a task and view it as a black box. An example of such a metric is the execution time of a task, user satisfaction, or hardware resources used, such as CPU, memory, inputs and outputs (IO), and network. Another example is repeatability, which may be boolean or continuous, computing the variance over multiple experiments. The

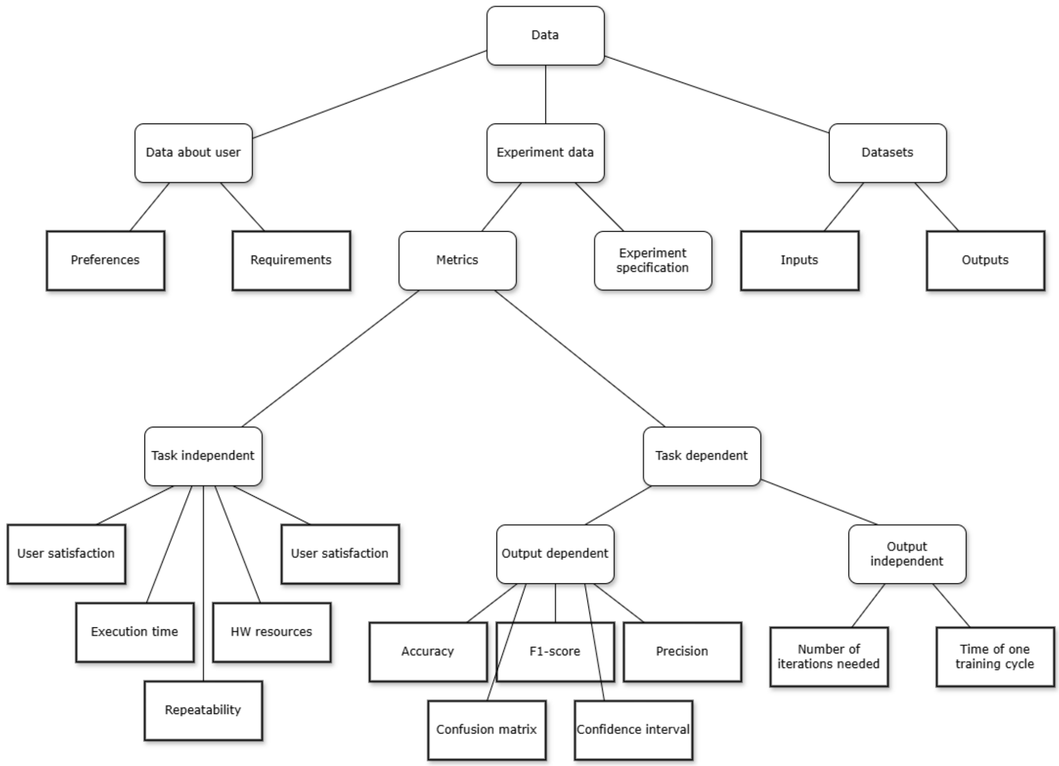

Figure 3.1 The overview of the stored data in the system with a focus on metrics

second category of metrics is task-dependent metrics, which have to understand the task to be computed. Task-dependent metrics can be divided into output-independent and output-dependent metrics. Output-independent metrics can be computed without accessing the output, and an example of such a metric is the number of iterations needed or the execution time of a part of a task (e.g., time of one training cycle). Output-dependent metrics need to access the output to be computed. An example of task-dependent output-dependent metrics is the accuracy of a model over a test dataset, precision, recall, or F1 score. Important metrics are also a confusion matrix and a confidence interval. Those metrics show that the metric result does not have to be a float number or a time-series, but also more complex structures.

Based on the analysis of the metric kinds, metrics are connected with the executed workflows. If a metric result is a simple value, it is contained directly in the executed workflow data. Small structures such as confusion matrices and confidence intervals are considered simple values in this context. If a metric produces a time series, then the executed workflow contains only the ID of the metric result. The executed workflow also contains a field describing the metric result type.

3.2.3 User data

Data about users, including their preferences and requirements, needs to be stored in the system to provide fit-for-purpose insights. This thesis also does not focus on user data, but it is important to take them into account.

3.3 Traceability

One of the main advantages of ExtremeXP is its support for traceability. Traceability is an important term in software development, as it enables following a life of software artifacts and keeping the relations between them [6]. In the context of ExtremeXP, we define traceability as the ability to obtain information about the experiment performed from any produced output dataset or computed metric. Obtained information includes inputs, parameters, and a snapshot of the used workflow algorithm.

Traceability is especially important for experiments involving machine learning. This is because machine learning experiments often involve iterating over multiples of different configurations, and the experimenters may fall into a spiral, where they run an ML algorithm, get results, save them, and then change some parameters and run again. Even datasets can be augmented in addition to changing a parameter. This can easily lead to losing track of differences between the runs.

For traceability, we envision that whenever there is a dataset (either external or internal) or a computed metric, we have a complete trace of how it was created so that we can repeat the process of creating it.

ExtremeXP makes it easier for the experimenter by keeping a trace of the data without user interaction. We allow the user to focus on performing an experiment, with the possibility of looking back and exploring the data that were used back then and branching by modifying some parameters or augmenting the data.

3.3.1 Traceability of workflow's model

The first step of experimentation is to design a workflow model. This model is stored in the EMF-based model repository. Then, the experiment is run and stored in an executed experiment repository. A way to obtain the model used for the executed experiments needs to exist.

The first option would be to store a model reference pointing to the model repository, as with datasets. This optimizes the space used, as a simple URI or even just the model ID would need to be stored in the executed experiments repository.

However, the user may decide to modify the model. In that case, the versions would also need to be remembered, which brings complications. A mechanism to store and track various versions would need to be implemented in the model repository. Versioning might also imply a link between the workflows; for example that the inputs are the same, but this may not be true. Because of this, versioning during design should be avoided. This is based on the same principle that a program is not versioned upon each run.

A snapshot needs to be remembered to prevent versioning while maintaining a trace of the used model. If a user runs a workflow a snapshot of the run workflow is taken and stored with the executed workflow. Each snapshot must contain a transitive closure to store all relative information.

The structure of the model does not have to be considered in the executed experiments repository, as this is done by the EMF-based model repository. However, the data service should enable to store some model information in the executed workflow's metadata to enable querying based on them.

3.3.2 Requirements for achieving traceability

An experiment usually produces output datasets and computes metrics. The goal is to have a complete trace for every metric and dataset (either external or internal) created by ExtremeXP, describing its creation. Thus, the process of creating the dataset can be repeated.

What must be avoided is having results that cannot be traced back to the executed experiment that created them. Whenever a dataset is created, it should point to the executed workflow that created it.

During analysis, the concept of "closed task" was introduced. This is essentially the computation's closure, including used workflow models, all inputs, and parameters.

The invalid state would also be to have a trace to the workflow model, which was later modified. This was one of the reasons why it was decided that each executed workflow contains a snapshot of the workflow's model. The executed workflow also keeps its parameters and metadata by containment. Metadata is stored as a key-value array. Metadata helps to capture additional experiment information needed to recreate the experiment [7] and, therefore, is essential for achieving traceability.

Each executed workflow keeps references to its input datasets. However, the datasets reside on an external distributed system and may not be under the framework's control. This means that the input datasets may change. For this reason, a checksum is stored with each input dataset's reference. The traceability can be guaranteed as long as the data correspond to the checksum.

Each computed metric and output dataset contains a reference to the executed experiment that created it. The executed experiment contains a list of references to its executed workflows, and each executed workflow contains its executed tasks.

3.3.3 Comparison of traceability with NFT's chain of trust

There is a small similarity between our work on the traceability of datasets and the work of our project partners working on NFT-based data provenance and specifically ensuring the chain of trust for the stored data. This section explains those terms and discusses their differences from traceability.

NFT stands for non-fungible tokens. This means that the tokens are not identical or even mutually replaceable. NFT is built on blockchain, which enables validation and tracking of the token's ownership to its origin. Blockchain works in a decentralized environment. This is possible because of the distributed consensus mechanism and the use of cryptography for hashing and digital signatures [8].

In NFT-based data provenances, tokens can be used to track data history and verify its origin and ownership. A chain of trust provides a certificate of authenticity. NFT-based data provenances have several advantages, including trust and accountability and fraud prevention.

The difference between traceability and providing the chain of trust in NFT-based data provenance is that the chain of trust provides information on how much the data can be trusted. Traceability, on the other hand, allows experiments to be reproduced based on the output data or metrics.

3.4 Reproducibility

It must be said clearly that traceability does not imply reproducibility. Usually, experiments can be reproduced only to some extent. Because of stochasticity, output data can not be reproduced exactly, but metrics like accuracy are expected to still produce approximately the same results. Experiments produce various results - output datasets and metrics. Metrics include task-dependent and task-independent. Task-dependent can also be divided into output-dependent and output-independent. This section analyses the reproducibility of each result type and necessary conditions.

First, let's analyze the reproducibility of output datasets. There are cases where executing an experiment with the same algorithms, parameters, and input datasets produces the same output datasets, but this is true only for experiments not using heuristics, or the randomness (e.g., random seed) needs to be controlled and stored as well.

Because output datasets can differ and not be fully reproducible, output-dependent metrics also experience changes between experiment runs. However, for metrics computing average behavior, like accuracy or precision, those changes are usually small in comparison with changes in the raw data because the changes tend to even out. Repeatability can be increased by experiment size in some scenarios. For neural networks, it was proven that increasing the network size or number of iterations leads to convergence of the distribution of neural network parameters to the solution of a non-linear partial differential equation. This is similar to the Law of large numbers [9]. The law of large numbers theorem and the Central limit theorem were proven for Stochastic Gradient Langevin Dynamics (SGLD) [10].

Task-independent metrics see the task as a black box. An example of such metrics is the used CPU resources. Those metrics are usually affected by the environment outside of the system. For example, if the application runs on a server with other tenants, it may suffer from the Noisy neighbor problem. For these metrics to be repeatable, the environment must be completely controlled.

The last category is task-dependent output-independent metrics. An example of such a metric is the number of iterations needed for a machine learning algorithm to reach convergence. This is also influenced by heuristics, which are used, for example, during batch sampling. The effect of heuristics can again be lowered with higher experiment size as supported by the Law of large numbers and the Central limit theorem.

3.5 Data model

The Data model was done by this thesis to help with the communication regarding stored data in the system. Figure 3.2 depicts the executed workflow and related entities and their properties.

The main entity is the executed workflow, which represent a sequence of tasks, for example machine learning pipeline. Executed experiment contains a snapshot of the workflow's model and execution information such as start date, end date, and status. The model captures all the algorithms used. An executed workflow is identified by its ID and title. Each executed workflow takes inputs and parameters and produces outputs and computed metrics. Metrics enable to summarize results and help to make decisions. Each metric is identified by an ID, contains its name, and provides the result, which, based on its type, can be an embedded value or a reference. Data references contain an URI to the stored data and their checksum.

The executed workflow contains executed tasks, which correspond to individual tasks, for example, data preparation and augmentation. The introduction of executed task enables the decomposition of the workflow into smaller parts and provides more fine grained information as each executed task also stores information about its inputs and parameters, produced outputs and computed metrics.

Each entity can also contain its metadata, enabling to store additional information, which allows to capture the broader context of the computation and data creation.

The data model also captures relation cardinalities. Cardinalities provide additional, but also necessary, information about relations between entities. Cardinalities are depicted as numbers on relation lines and are intervals. For example, an executed workflow may produce 0 to n output datasets. This means that the number of created output datasets by a single executed workflow is unbound. On the other side, an output dataset can be produced by exactly one executed workflow. Cardinalities are also beneficial for data storage structure. For example, an array of output dataset information can be stored in the executed workflows as is implied by the cardinalities.

The proposed data model is made to depict dependencies. For example, a parameter must not exist without a task and workflow, which it is a parameter to. This is captured in the data model in values of cardinalities. Each parameter has a relation with exactly one task and exactly one workflow. This does not mean that, for example, a learning rate can not be shared amongst multiple tasks, but rather that each task has to capture the learning rate's value independently.

The same can be said for output, as it must hold that for every output, there is a task run or workflow run that has created it. This dependency is especially significant for traceability.

The data model does not allow input data not assigned to any workflow run or task run. The actual data of the input datasets may exist, but either somewhere outside of our system or as a result of another task or workflow run.

Additionally to the data model, executed experiment entity was added later in the project development and is reflected in the data storage and the API. An Executed experiment is a collection of executed workflows IDs with name of the experiment and an intent attribute enabling the project description.

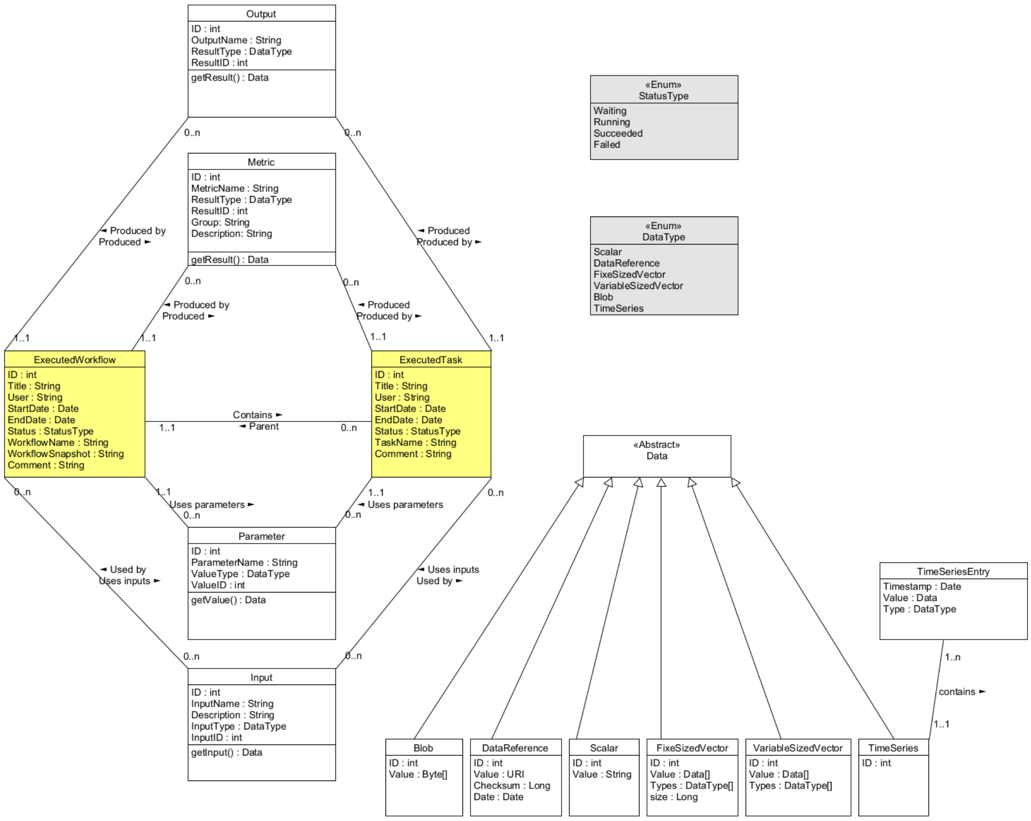

Figure 3.2 The data model of the executed workflows and related data

4 Design of the API

This chapter describes the design of the API done by this thesis.

4.1 API usage scenario

Rest API needs to enable callees to create an executed experiment. During the experiment creation, the intent may be specified. Each executed experiment also contains a list of executed workflow IDs to group them.

After the experiment is created, the execution engine runs workflows and needs to store information about executed workflows. After a workflow is executed, it is to be stored in the data service, and the stored executed experiment's list of executed workflows must be updated.

The executed workflow contains its tasks, which may be filled by the execution engine during the execution. As each task produces its outputs and computes metrics, the API enables the execution engine to update corresponding fields. The API also enables the creation of time-series metric definitions, containing schema, and uploading the data in batches during the computation.

The API provides means to add metadata for the experiment, a workflow, a task, or a metric. This may be done, for example, during the computation or afterward.

For retrieving the results, e.g. for visualizations, the following scenario can be used:

1. A request to executed experiments endpoint to retrieve the list of executed workflow IDs

2. Requests to executed workflow endpoint to retrieve information containing workflow model, parameters, input datasets, produced output datasets, computed metrics, and tasks.

3. In the case of time series metrics an additional request to retrieve metric data points is expected. The request may specify a start date, the number of maximum data points to be retrieved and signals to be included. A signal is a captured attribute, for example timestamp or CPU utilization.

4.2 Data retrieval categories

Several categories of data retrieval scenarios were proposed by this thesis for the API. The categories were designed to maximize the API's capabilities while keeping it not too broad.

Those categories are

- Experiments overview

- Data overview

- Experiment result analysis

- Experiment description
- Reliability or reproducibility of an approach
- Comparison of multiple approaches
- Traceability

4.2.1 Experiments Overview

This category aims to provide support for getting a general overview of the executed experiments. This may be useful, for example, when the user wants to run an experiment and wants an overview of experiments that have already been run for inspiration or to avoid repetition. The user may need a list of executed experiments, executed workflows or executed tasks. Or the user may be interested in their models.

4.2.2 Data Overview

The second category of data retrieval scenarios is data overview. This category aims to provide a general overview of the system's used or created data. This may also be useful when running an experiment, as it provides inspiration for the experiment configuration and prevents repetition of the same experiment. The requests should provide information about input datasets, output datasets, parameters, and produced metrics.

4.2.3 Experiment Result Analysis

The third category aims to enable a user to analyze results from one executed workflow. This is useful for an evaluation of an approach. This category contains scenarios where the output of a specified experiment is wanted. Or the callee wants one specific computed metric, group of metrics, or all metrics.

4.2.4 Experiment Description

The fourth category contains scenarios returning a description of one executed workflow. By description, we mean the algorithms, parameters, and input datasets used. This is important for traceability. It is a necessary condition of traceability but not sufficient, as we will see in section 4.2.7.

4.2.5 Reliability or reproducibility of an approach

The fifth category aims to provide support for measuring the reliability or reproducibility of an approach. To measure reliability or reproducibility, the results of an approach need to be compared on various input datasets. This means fixing the values of parameters and workflow models. The requests may be to get one metric, a group of metrics, all metrics, or outputs. This category provides a way to check how much the results can be trusted.

4.2.6 Comparison of multiple approaches

The sixth category contains scenarios for comparison of multiple approaches. This means comparing the results of different executed workflows. The same approach may lead to different results based on selected input datasets, as discussed in section 4.2.5. This is why it is important to fix the input dataset.

Another approach would be to compute reliability for both approaches, and then they could be compared on different datasets with some heuristics. However, even when comparing the approaches on the same datasets, some heuristics are involved, as the dataset may be more suited for one approach than the other.

Approaches may differ in the algorithms used or parameters' values. If algorithms are fixed and some parameters, the results may answer the question of how the other parameters influence results. This category is useful for selecting the optimal solution for the given problem. Used algorithms and values of parameters form configurations. Requests may be to get one metric across configurations complying with fixed algorithms and parameter values. Other possibilities include getting a metric group, all metrics or inputs across configurations, and complying with fixed algorithms and parameter values.

4.2.7 Traceability

The seventh category is supporting traceability. This means that each piece of produced data needs to contain a trace pointing to the executed workflow that created it. This is crucial in maintaining an overview of the experimentation process. All requests have in common that they are based on output or a metric. The inputs, parameters, executed workflow, and computed metrics may be obtained based on them.

5 IVIS

This chapter describes the work done by this thesis on the IVIS framework, starting with a summary of used technologies. Section 5.3 provides examples of work done during the refactoring.

The IVIS framework was developed at Charles University. It is an open-source project designed to simplify development in data visualization and analytics.

5.1 Technologies

IVIS has two parts: client and server. The client relies on the React library, while the server is built on Node.js. Both technologies are introduced in the following subsections.

5.1.1 React

React is a Javascript library for creating user interfaces in web applications. It uses a component-based approach, where each component manages its state, leading to better encapsulation.

Each element on the page is included in the document object model (DOM), a tree-data-structure representation of the webpage. DOM is browser-independent, providing cross-browser compatibility. React uses DOM to access the current structure of the page and to make modifications based on computations or user interactions.

React is designed to promote a declarative approach. In this approach, the code specifies the expected state of the application. On the other end of the spectrum, there is an imperative approach, where code describes the steps leading to achieving the expected state. React enables describing what the UI should look like, and it encapsulates the work with the DOM. The programmer does not have to specify how the UI should be built or updated.

5.2 Node.js

The server is built on Node.js, a runtime system that enables Javascript to run on the server side. Historically, the server side was usually written in the scripting language PHP. However, unifying the server and client programming languages has multiple advantages, including code reusability and making it easier for the programmer to master the language.

Node.js is built on the V8 engine, a high-performance JavaScript engine written in C++ and used in Google Chrome [11]. Node.js's high performance stems from the programming language choice and its asynchronous event-driven architecture. The main focus of this architecture is not to waste resources by idling. Client requests are received in an event thread, which handles small tasks directly. The more resource-consuming I/O tasks are delegated to the worker's thread pool managed by the C library Libuv [12].

Other advantages of Node.js include support for various operating systems, making it cross-platform. Node.js also includes the Node package manager (NPM), the largest open-source library collection. Its vast community ensures frequent updates for the foreseeable future.

Some limitations of Node.js also need to be taken into account. While the asynchronous even-driven architecture is usually responsible for high performance, CPU-intensive tasks might lead to blockage of the event loop. Also, as NPM contains over 2 million packages [13], some may be of poor quality or even pose a security risk. NPM provides tools to audit and even fix known vulnerabilities.

5.3 Refactoring

To implement the data service using the IVIS framework, it first needed to be updated, as most of its listed dependencies were deprecated. This chapter contains a few examples of how the dependencies evolved, the reasoning behind the changes, and how the code needed to be modified to incorporate the changes.

5.3.1 React Router Route Properties

IVIS is a single-page application (SPA). This means that a single HTML page is loaded and dynamically updated based on user interactions. SPAs do not require a full reload after a user navigates; they dynamically update only the needed parts. This enables users to traverse different website parts efficiently [14].

IVIS's client uses the React Router, which allows declarative routing. When a user wants to visit a certain page, the current URL is matched and used for rendering components. The React router is still under active development, and many changes were made between versions 4.3.1 and 6.22.0.

One change was how a component's routing-related properties, including location, match, and history, are obtained.

The location property provides information about the current URL. It contains information about the pathname, the search part of the query, the hash part of the query, and the key that uniquely identifies each location in the history stack.

Match property provides information about the current route. It contains information about the matched pattern, the matched portion of the URL, a boolean indicator of whether the match is exact or partial, and key-value pairs of query parameters.

The history property enables access to the browser's session history and traversal of the visited pages forward or backward. It provides a way to obtain information about the number of entries in the history stack, current location, current stack action (PUSH, POP, REPLACE), and methods for history traversal and modifications.

IVIS used React Router version 4.3.1, which exported the higher-order component (HOC) withRouter. HOCs are functions that take a component as a parameter and return a new component [15]. This can be beneficial for code reuse and encapsulation. The withRouter HOC can be used as decorator [1] to provide

[1] Javascript decorators provide a way to extend the functionality of a class or a function without the need to change the code inside the function or the class. It provides an efficient way

access to routing properties. Figure 5.1 shows the import of withRouter and its application on the class WorkspacePanel, providing access to location, match, and history properties.

```
import React, {Component} from "react";
import {withRouter} from "react-router-dom";


@withRouter
export default class WorkspacePanel extends Component {
    componentDidMount() :void  {
        console.log('Location:', this.props.location);
        console.log('Match:', this.props.match);
        console.log('History:', this.props.history);
    }
}
```

Figure 5.1 An example of importing withRouter decorator and applying it to WorkspacePanel to provide access to routing-related properties.

Major version V5 was fully backward compatible with version V4 as it contained no breaking changes. This version provided better support for React 16 and was not originally intended to be a major version bump.

React version 16.8 introduced hooks. Hooks allow function components to have access to the component's state. Before that, only class components could have a state, but functional components were stateless. Hooks also allow a component to access its lifecycle information [2] . Hook usage needs to follow three rules. First, Only function components can use hooks. Second, hooks can only be called at the top level. Third, hooks cannot be conditional. The first rule renders class components of little use; however, they can still be used if a functional wrapper is provided, passing the routing properties.

React Router version 5.1 introduced hooks for router properties, such as useParams, useLocation, useHistory, and useRouteMatch. UseParams makes it easier to access dynamic parts of the URL. The rest of the hooks enable access to the properties, which in version 5.1 could also be accessed via withRouter HOC.

React Router V6 removed withRouter, forcing the usage of hooks and functional components. This is a peculiar decision as, on the one hand, it feels like restricting programming styles. On the other hand, introducing hooks into class components would lead to multiple ways of managing state and side effects. Class components would allow both this.state and useState patterns, leading to a confusing code. Figure 5.2 shows obtaining router props in React Router version 6.22.0.

5.3.2 React Router Routing

Defining routes is based on pattern matching. The set of routes needs to be defined, each containing a path that is to be matched and an element to be

to reuse expected behavior. It makes the code more readable and maintainable. Decorators are an experimental feature and a transpiler like Babel is needed to use them, as they are not part of the ECMA standard.

[2]Component lifecycle consists of three parts - mounting, updating, and unmounting. Before React 16.8, only class components could access their lifecycle information.

```
import React, {useEffect} from 'react';
import {useLocation, useParams, useHistory, useRouteMatch} from 'react-router-dom';

export default function WorkspacePanel() : void {
    const location : Location<LocationState> = useLocation();
    const params : {} = useParams();
    const match : match<{...}> = useRouteMatch();
    const history : History<LocationState> = useHistory();

    useEffect( effect: () : void => {
        console.log('Location:', location);
        console.log('Match:', match);
        console.log('Params:', params);
        console.log('History', history);
    }, deps: [location, params, history]);
}
```

Figure 5.2 An example of obtaining router props in React router version 6.22.0.

rendered upon the match. A route can also specify an error element, which will be rendered when the primary element render results in an error. The paths can be either literal or contain placeholders. A placeholder is, for example, the ID of an executed workflow. The component rendered for each executed workflow is based on the same class, but each gets passed a different ID, which may then be used to gather the executed workflow's data.

React Router also supports protected routes, which enable user authentication, for example. The protection policy can be specified using a custom HOC. The URL can be modified using a Redirect tag. Figure 5.3 contains an example of a redirect.

```
import {Redirect} from "react-router-dom";
import React, {Component} from "react";

export default class WorkspacePanel extends Component {
    render() {
        if (!isAuthenticated()) {
            return <Redirect to="/login"/>;
        }

        return <Account entity={props.resolved.user}/>;
    }
}
```

Figure 5.3 An example of a redirect to login if the user is not authenticated

However, in version 5.1, the redirect tag could not be directly in the switch tag, and switches could no longer contain any custom components [16]. Custom components were used in switches to wrap the plain routes with additional routing logic, such as checking privacy settings. The wrappers needed to expect all properties of the nested route and its own [17]. This complicated maintaining compatibility between the wrapper and the React routes. The current recommended approach is to add logic by passing composition to the element property of the route tag. Figure 5.4 shows an example of the replacement of a wrapper for a protected route.

In v6, the redirect tag was removed, and the recommended redirect approach was changed. The issue was that the redirecting was done in a way that the initial page was served with response code 200, and then the state was replaced. This

```
import {Route,Routes} from "react-router-dom";
import React, {Component} from "react";

export default class WorkspacePanel extends Component {
    render() {
        return (
            <Routes>
                <Route
                    path="/account"
                    element={
                        <RequireAuth redirectTo="/login">
                            <Account/>
                        </RequireAuth>
                    }
                />
            </Routes>
        );
    }
}
```

Figure 5.4 An example of the replacement of a wrapper for a protected route. RequireAuth contains logic for user authorization and either renders the account component or redirects to the login page.

led to issues with search engines, as the search engine crawler was not aware of the redirect and it indexed the page [18]. So the redirect tag was removed, and the recommendation is to do the redirect manually before rendering and to return the correct response code.

In v6, the Switch component was replaced with the Routes component. This component introduced several changes. The first of them is that the first matched route is no longer selected but rather the route with the best path match. Also, routes need to be defined in one place, leading to a better overview of the routes. Patterns used in the path were simplified, and only two types of placeholders are supported - placeholders for parameters (such as ":workflowId") and wildcards denoted by an asterisk. Wildcards can be only at the end of the path expression.

5.3.3 D3.js

D3.js library is used for graph rendering and enables user interactions such as zooming and data filtering.

IVIS was using D3.js V5, and the dependencies were updated to use V6 during work on the data service. This brought various changes, and the code needed to be refactored to render the graphs and process user interactions correctly.

The first change was that D3.event is no longer a global variable. In current versions, each listener gets the event as the first argument. This change conforms to ES2015, a modern Javascript standard encouraging the removal of global variables. This makes the code more readable, easier to debug, and thread-safe. A global event variable could lead to race conditions.

D3.mouse was also replaced with D3.pointer as input is no longer limited to the traditional mouse; various other pointers, such as touchscreens, are common. D3.pointer provides a unified way to process user interactions on various device types. D3.pointer still enables the use of features specific to the given device type.

For example, pressure, twist, or tilt can be accessed. If the code is running on a device that does not have the ability to detect the exact pressure, twist, or tilt, default values are returned.

5.3.4 Other Dependencies

Other updated dependencies include

- CoreUI - Front-end framework for developing dashboards
- Axios - Hypertext Transfer Protocol (HTTP) client handling requests between the Node.js server and the browser.
- Bootstrap - Framework using HTML, CSS, and JavaScript for building responsive websites.
- I18next, React-i18next - Internationalization frameworks
- Elasticsearch - NoSQL search engine
- Babel - Javascript compiler for extended browser compatibility
- Sass - CSS extension language
- Webpack - Module bundler into static assets
- Css-loader - Webpack plugin for CSS loading

5.4 API layer design

IVIS also implements the basics for the API layer. The API is built on the Express.js framework, which is commonly used for building RESTful APIs. Express Router enables separately creating a group of routes and their handlers [3] and integrating the group into the application, which leads to modular and reusable code.

An express router contains methods that enable the addition of a route based on the request type. To specify a group of routes, an instance of Express Router is created, and routes are added by calling a method based on the request type. Route specification includes match path, which may be regular expressions. Figure 5.5 shows an example of adding a route that will process requests to the executed workflow endpoint with a variable workflow ID.

The express router also offers a function that enables the addition of a route, which will process requests to the given path for any HTTP request type.

[3] Handlers are callback functions which are called when the server gets a request to the specified route.

```
'use strict';

const express = require("express");
const router = new express.Router();

router.get('/executed-workflows/:workflowId', async (req, res) => {
    const { workflowId } = req.params;
    res.status(200).send(`Request for workflow ${workflowId}`);
})
```

Figure 5.5 An example of adding a route that will process requests to the executed workflow endpoint with a variable workflow ID.

6 Data service technologies background

This chapter provides background on the technologies used for the data service. It includes a discussion of technologies for data storage and a description of REST API.

6.1 Database Management System Overview

When selecting a database, the first step must be to decide whether the database should be an SQL database or a NoSQL database. There is no universal solution for which database management system (DBMS) is better. The choice depends on the use case and requested properties. Database management systems differ in data model, performance, scalability properties, and query languages, which also affect the usability and performance of the database. In this section, SQL and NoSQL approaches will be described and compared.

6.1.1 SQL Databases

The first kind of database is SQL, which used to be a traditional approach. The data are stored in a collection of tables, where different attributes correspond to columns, and each record has its row.

The database is strongly typed, meaning that each column has its type and can contain only data of that type. Each column can also have specified further constraints, ensuring further data consistency. It may be specified that the values in the column must not be null or that each value has to be unique. The more complex constraints can be specified using check constraint, where an arbitrary condition can be specified. For example, the constraint that an experiment finish time must be greater than the experiment start time. Another constraint keyword is default, which specifies a value to be used when no other value is specified.

SQL databases are usually relational schemas containing multiple tables with relations between them. Each table represents a class of objects. The primary key uniquely identifies the record. Foreign keys are used as links between databases, providing relationships between the objects.

Data in SQL databases are accessed and modified via SQL statements - write, read, delete, and update. SQL was designed to resemble English and is quite natural. The statements can be grouped into transactions, which ensures the database's integrity. Each transaction ends with a commit or rollback. A transaction must follow four characteristics denoted by ACID - atomicity, consistency, isolation, and durability. SQL databases enable joining multiple tables to use the relations between objects and return all expected fields.

For the data service proposed and implemented by this thesis, scaling is important as the number of executed experiments is expected to grow immensely. SQL databases can be scaled only vertically, as they use one server and cannot be divided into multiple servers on different devices. So, if the SQL database is to be scaled, more resources need to be added to the single server.

6.1.2 NoSQL Databases

The second type is the NoSQL database management system. NoSQL differs from SQL as data is not stored in tabular storage but in other data structures, such as JSON documents. NoSQL databases do not require a schema, which enables the management of large and unstructured data and gives the user more flexibility. This kind of database is experiencing growth within cloud, mobile, and web applications.

NoSQL is a distributed system in which data can be stored locally or remotely. Because NoSQL databases are non-relational, they can be divided into shards, each containing part of the data and not relying on the rest of the system, enabling rapid scalability. NoSQL is typically scaled horizontally, as another server can be added to help with the traffic. Also, even if part of a database is not reachable, the rest can be obtained.

Sharding

By default, NoSQL databases support sharding. A shard is part of the data in the database or a search engine. Each shard can be stored on a different device, resulting in horizontal scaling. The shards do not share the same data.

NoSQL database types

NoSQL databases have multiple types. Data can be stored in key-value pairs, documents, wide column stores, graph stores, or even in memory.

Key-value pairs are the simplest way to store the data. Data are stored in a dictionary. Each key is mapped to its value, which can be an array of data. This approach is not ideal for fetching multiple records at a time as it optimizes simple key-to-value retrieval and not more complex queries such as advanced filtering, aggregations, or searching across multiple attributes. One of the use cases of the key-value pairs approach is storing user-session information. Redis is an example of such a database.

The second option is to store the data in documents. Various structures can be used, such as JSON or XML. This is especially useful for semi-structured data. Semi-structured data do not have a fixed schema but still contain some structure, which makes them easier to parse and analyze. The structure does not have to match across documents, but this comes at a risk of data corruption as there is no easy way to correct mistakes in field names. For example, one document can use the field executedExperiment and another executed_experiment. Querying across multiple attributes is possible; however, it can be inefficient if documents have different structures. One of the use cases of the document storage type is storing user profiles. MongoDB is an example of such a database.

The third type is a wide-column store (WCS). This approach is similar to the traditional SQL approach as it enables data storage in columns. The difference is that a wide column store enables each row to have its own set of columns. WCS is designed to store enormous amounts of data, as WCS systems like Apache Cassandra enable each row to have up to 2 billion columns [19]. WCS enables users to access the data without irrelevant data. WCS also solves problems of key-value pairs storage and document storage, such as querying across multiple attributes, which can be done efficiently using indexes on multiple columns. One of the use cases of the WCS is social networking websites. Apache Cassandra is an example of such a database.

The fourth type is a graph store. Data are stored in the knowledge graph. A node is an object, and an edge denotes a relationship. Each node or edge can have key-value pairs of properties. The joins are not needed, as relationships are already caught in the graph. The databases are optimized for exploring relationships. This approach is especially useful for highly connected data, such as supply chains. Neo4j is an example of such a database.

6.1.3 CAP theorem

Distributed data systems follow a fundamental principle called the CAP theorem. This theorem states that every distributed system can guarantee only two out of three of the following properties [20].

- Consistency - All nodes return the same data. When data are written to one node, they are propagated to all other nodes before the write is finished.
- Availability - Any request returns a valid response, even if some nodes are offline.
- Partition tolerance - The cluster works even if communication between some pairs of nodes is broken.

No distributed system can achieve all three properties, so a pair of properties needs to be picked [21].

- AP - Availability and partition tolerance are provided. This means that consistency is not guaranteed, and users may get inconsistent data. Apache Cassandra is an example of a database implementation providing availability and partition tolerance.
- CA - Consistency and availability are guaranteed. Partition between two nodes may break the system. Relational DBMSs such as MySQL are examples of databases with those properties.
- CP - Consistency and partition tolerance are guaranteed. If a node becomes offline, some data may not be available. MongoDB and Redis are examples of database implementations with these properties.

6.1.4 Emerging Trends

Various emerging trends also need to be taken into account.

The first trend is NewSQL, which is gaining popularity because it complies with ACID-like SQL and is comparable to NoSQL in scalability.

The second trend is the use of AI in databases. AI can replace humans when setting parameters and defining backup and recovery policies. AI can also improve database performance by data cleaning, indexing, and query optimizations.

The third trend is cloud-based DBMS. This approach provides additional scalability and flexibility. It is also a cost-effective solution as it lowers the infrastructure's cost. Cloud-based DBMS can use SQL, NoSQL, or newSQL. Cloud platforms also provide ways to optimize performance, including automatic caching and read-only replicas. One of the challenges of this approach is latency, which can be reduced by geographic distribution of data [22]. Other challenges include data security, vendor lock-in, and unexpected scaling, which can lead to a higher cost.

The fourth trend is decentralized DBMS using blockchain and Distributed Ledger Technology (DLT). The blockchain foundation makes the databases tamper-resistant and highly reliable [23]. This is because each block's hash is added to the subsequent block and used to compute its hash. This means that if a block is altered, its hash changes, and so on for each subsequent block. The further in the past, the change would be made, the more significant computational power would be needed to alter the subsequent blocks.

The fifth trend is in-memory DBMS, which stores data in the main memory instead of in disk storage. It provides significantly greater speeds [24] as the latency of communication with disk storage is much greater than computing in main memory. The disk storage may still be used for persistence or backup.

6.2 API protocol selection

API provides a way for other applications to communicate with the data service. For two applications to communicate with each other, a protocol needs to be established, containing a set of rules defining the expected communication. The structures of requests and responses need to be agreed upon. Communication can be facilitated by methods or endpoints. Each method or endpoint exposes a specific functionality of the system and has defined its expected inputs and outputs. The Clear choice for API is REST API as it is the most used API for web applications.

6.2.1 Rest API

The REST API protocol follows a Representational state transfer (REST) design. It is the most known API protocol, making it easier to use in projects like ExtremeXP.

It enables uploading, modifying, reading, and deleting data using HTTP. Each request must specify the URL of the target endpoint, the HTTP method, and the HTTP header. The following HTTP methods can be used: GET, PUT, POST, and DELETE. Requests, except those with the GET request method [25], may also include a body with a payload.

The HTTP headers contain, for example, content type and content length and additional headers such as the authentication token.

Each request should result in a response from the server. Each response contains a body and headers, including the response code. The body may be plain text, JSON, HTML, YAML, or XML. Response codes have the following semantics:

- 1xx - Protocol level information
- 2xx - Success

- 3xx - Redirect
- 4xx - Client error
- 5xx - Server error

7 Data storage system technical realization

In the previous chapter, various databases were described. This chapter describes which database was selected to store executed experiments and their data, including metrics. Section 7.3 briefly summarizes what work was done to set up indices in the Elastic Search. Section 7.2 briefly discusses the storage of user authorization data.

In our case, a database management system that enables the storage of data with a schema that is not known beforehand is needed. The number of executed experiments is expected to grow significantly, and the database management system needs to support horizontal scaling. Also, we need to support the time series as metric results. Even though a database-as-a-service like TempoDB could be used [26] to store time series, this solution would obscure data storage infrastructure and complicate non-time series use cases.

After the analysis, the NoSQL DBMS was selected. As its implementation was selected Elastic search. Its capabilities and reasons for the selection are described in section 7.1. User authorization information was stored in the MariaDB, and after problem analysis, it was decided to keep it this way. The reasoning behind this is described in section 7.2.

7.1 Elastic search

Elastic search is a search engine based on the principles of NoSQL databases. It is an example of a distributed database, as the data do not have to be stored in a single node but rather in a cluster of nodes. A cluster may contain several nodes on different machines. Data in elastic search are stored in JSON files, which can be grouped together to form an index. An index may store data in shards across one or more nodes of a cluster. Figure 7.1 [1] shows an example of a cluster with three nodes containing an index with three data shards, each having one replica shard.

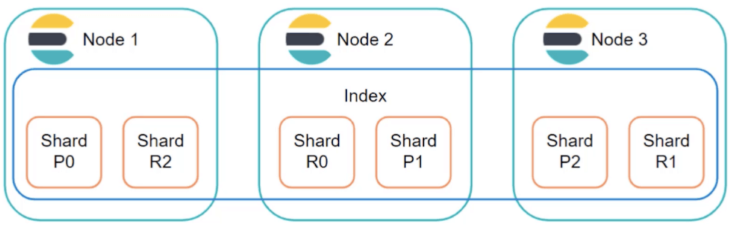

Figure 7.1 An example of a cluster with three nodes containing an index with three data shards, each having one replica shard.

Elasticsearch supports horizontal and vertical scaling. More shards can be added for horizontal scaling. Each shard can also run queries, enabling queries

[1]Figure 7.1 was taken from https://jovepater.com/article/es8-ep2-elasticsearch-cluster-installation/

to run in parallel. Replicas can be used to provide fault tolerance and further horizontal scale. Replicas are backup shards, and they provide fault tolerance by mirroring data in a different node [27]. Replicas increase disk size to store the data, so a tradeoff between infrastructure cost and fault tolerance needs to be made. Also, the number of replicas can be changed at runtime, but changing the number of shards requires the system to be off. Both approaches keep existing data.

Each node has its own limit for the stored data. A growing number of users will lead to growth in executed experiments, and sooner or later, adding a new shard to keep the queries fast will become necessary. But each shard has some overhead. This brings up the question of optimal shard count. One approach would be to keep the one shard per node, which significantly impacts the performance, as it limits parallelism capabilities. On the other end of the spectrum is creating an extreme number of shards. However, this leads to over-sharding as each shard consumes part of memory and CPU resources. A sharding strategy can be defined to keep the optimal number of shards. The strategy should take into account number of nodes, the volume of the data, and the available heap of each node. Developers of the elastic search state that a shard should contain less than 200 million documents, with a shard size from ten gigabytes to fifty gigabytes [28]. In previous versions of elastic search, there was also a limit of twenty shards per gigabyte of an allocated heap [29], but that recommendation has been deprecated starting version 8.3. Similarly to the shards, indices also have overhead. The developers of elastic search state that a node should provide at least one gigabyte of heap per three thousand indices [28].

Elastic search enables the automatic balancing of shards to use the available resources fully. It is also possible to divide nodes into data tiers, and data within the tier are rebalanced based on node additions or removals.

Elastic search uses Apache's Lucene as its core for searching and indexing. Each shard is a Lucene index [30].

The Elastic search was selected as the database for executed experiments and metrics because it provides great performance, scalability (including horizontal), support for fault tolerance, and a large community, ensuring support for the foreseeable future.

7.2 IVIS access control

Ivis contains different levels of authorization. The access control is fine-grained, and a user can be given access to a single entity or a whole namespace. An entity is, for example, a dataset, a visualization panel, or a workspace. A user can also be given access to an entire namespace. When providing access to a user, the permitted action is also specified. A user may be permitted to use an entity, update, or delete.

Each entity code specifies expected permissions. If a user tries to access the entity without sufficient permissions, an exception is thrown, and access is denied.

Authorization data are stored in MariaDB, an SQL database. MariaDB is a sufficient solution because the data are small and have a fixed schema. This is not the case with workflow data.

7.3 Data storage implementation

Create_indexes.sh script was created to set up the indices needed to store data about executed experiments. This script contains the URL of the Elastic search and the list of known indices. For each index, the script contains specifications of the index settings and mapping. Index settings specify the number of primary shards and replicas.

The script iterates through known indices and first checks whatever the index already exists. If it does, it is removed to ensure consistency. The index is then created based on its mapping and settings.

Currently, both primary shards and replicas for each index are set to one. During the release, the number of primary shards is supposed to change based on available resources and growth expectations. More shards support dividing the index into more nodes. Because replicas are set to one, each primary shard will have one backup shard, which is an optimal tradeoff between fault tolerance and efficient usage of available storage resources.

Known indices are the following:

- Executed_experiments
- Executed_workflows
- Executed_tasks
- Metrics
- Parameters
- Input_datasets
- Output_datasets
- Signal_sets

Several types are used in the mappings of the indices. Figure 7.2 shows the setup for the executed experiments index. Used types include:

- Text - fields with text type are analyzed by tokenizing the text into an array of words to support full-text search. This type can be replaced with match_only_text to optimize used space at the expense of positional queries. An example of such a field is the executed experiment's name.
- Keyword - fields with the keyword type are not analyzed and are stored as-is. This type is suitable for fields that require exact matches rather than partial ones. An example of such a field is the executed workflow's ID.
- Date - fields with Date type are automatically converted into milliseconds from the start of the epoch. Elastic search enables the definition of multiple date formats, in which case it accepts date strings of any specified type and converts them into number representation. This enables us to avoid dealing with different formats when querying. An example of such a field is the start of the executed experiment.

- Nested—fields with the Nested type enable storage of an array of objects with a specified schema. The type could be replaced with Object, but Nested enables querying the items in the array independently. An example of such a field is the metadata of an executed workflow. It defines the subfield's key and value.

```
executed_experiments_mapping='
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 1
  },
  "mappings": {
    "properties": {
      "id": { "type": "keyword" },
      "name": { "type": "text" },
      "start": { "type": "date" },
      "end": { "type": "date" },
      "metadata": {
        "type": "nested",
        "properties": {
          "key": { "type": "keyword" },
          "value": { "type": "text" }
        }
      },
      "comment": { "type": "text" },
      "workflowIds": {
        "type": "keyword"
      }
    }
  }
}'
```

Figure 7.2 The setup for the executed experiments index. Used types include text, keyword, date, and nested. Each type influences the storage of the field and querying capabilities.

8 API Technical Realization

This chapter describes the technical realization of the API that this thesis has implemented. API is used to integrate software components so they can communicate without the need to know their inner workings. Contrary to the database, an agreement with our partners on API protocol had to be reached. REST API protocol was selected for its popularity and simplicity. The description of REST API protocol is provided in section 6.2.

The following sections describe implemented endpoints, their functionality, expected request fields, their responses, and the server functionality to handle them. Each request must contain the header key “access-token”.The current access token can be found in the “account” tab under API.

8.1 Executed Experiments

The first group of implemented endpoints enables callees to add executed experiments, modify them, and query them.

Each executed experiment contains the following fields:

- Name
- Start
- End
- Intent
- Metadata - used to store an additional experiment data
- Comment
- WorkflowIds - used to get the list of associated executed workflows

8.1.1 Adding executed experiment

The PUT request method on the /executed-experiments endpoint can be used to add an executed experiment. The server first checks the access token to ensure that the callee is authorized to add executed experiments. Then the server validates the request’s body so it is ensured that the payload follows the designated schema. Validation is described in section 8.4. After validation, data are written in the executed_experiment index using the Elastic search index method. The executed experiment’s ID may be defined in the payload or may be left undefined. If the ID is omitted, an Elastic search generates a random unique ID, which is included in the return value of the index method. The server then returns a response from the Elastic search with code 200 or 400 based on the success of the operation.

Request

[PUT] http://localhost:8445/api/executed-experiments

Payload


```
{
  "name": "Experiment1",
  "intent": "Test the new approach for Increased Cybersecurity
  situation awareness for efficient threat mitigation"
}
```


Response

201 Created


```
{
    "id": "kV-cNpQBR7hBR5gtxTfk"
}
```


8.1.2 Modifying executed experiment

The POST request method on the /executed-experiments/:experimentId endpoint can be used to modify an executed experiment. “:experimentId” is a placeholder for the executed experiment’s ID. The ID is used to check whatever the database contains the executed experiment. If no executed experiment is associated with the given ID, the response’s code is set to 404, and the response payload informs that the executed experiment with the specified ID does not exist. Otherwise, the request’s payload will be validated against the schema. The update method of Elastic search is used to modify the experiment, and a response with code 200 or 400 is returned based on the operation’s success.

Request

[POST] http://localhost:8445/api/executed-experiments/kV-cNpQBR7hBR5gtxTfk
(The last part of the URL is the ID returned in the response to the previous PUT request)

Payload


```
{
  "metadata": {
    "experiment_type": "ML learning"
  },
  "workflowIds": [
    "VSses5AB4eIuD9qm4hd7"
  ]
}
```


Response

200 OK

```
{
    "message": "Executed experiment updated successfully"
}
```


8.1.3 Querying executed experiment

The GET request method on the /executed-experiments/:experimentId endpoint can be used to retrieve an executed experiment based on the experiment ID. Elastic search's get method is used to find the record in the database. In case of success, the record is returned with response code 200. Otherwise, the response informs that the executed experiment was not found, and the response code is set to 404.

Request

[GET] http://localhost:8445/api/executed-experiments/kV-cNpQBR7hBR5gtxTfk

Response


```
{
    "experiment": {
        "name": "Experiment1",
        "intent": "Test the new approach for Increased Cybersecurity
        situation awareness for efficient threat mitigation",
        "workflowIds": [
            "VSses5AB4eIuD9qm4hd7"
        ],
        "metadata": {
            "experiment_type": "ML learning"
        }
    }
}
```


8.1.4 Advanced querying executed experiments

The POST request method on the /executed-experiments-query endpoint can be used for the more complex queries based on the executed experiment's intent and metadata. The query's payload is validated and parsed. Elastic search's search method is used to find all the executed experiments conforming to the given criteria. The array of found executed experiments is then returned with response code 200.

The request results with the success code 200 even if no record corresponds to the criteria. This is different from the previous endpoint. The previous endpoint returned 404 if no experiment with the specified experiment ID was found because it meant that the request contained the wrong ID, which is considered a client error. The callee asked for a specific executed experiment that does not exist. In this case, the callee asks for a set of executed experiments with specified properties, and an empty set is a valid response.

Even though this endpoint is used for retrieving data and not storing them, the POST request method was used rather than the GET request method. This is because metadata is a collection of key-value pairs, and its length is unbound. If the GET request method were used, metadata would need to be passed as an array in the query part of the URL. POST request method enables to passing of the metadata as an object in the JSON payload. The object's keys correspond to the metadata keys using javascript's dynamic nature.

Request

[POST] http://localhost:8445/api/executed-experiments-query

Payload


```
{
    "metadata": {
        "experiment_type": "ML learning"
    },
    "intent": "Test the new approach for Increased Cybersecurity
    situation awareness for efficient threat mitigation"
}
```


Response


```
[
    {
        "name": "Experiment1",
        "intent": "Test the new approach for Increased Cybersecurity
        situation awareness for efficient threat mitigation",
        "workflowIds": [
            "VSses5AB4eIuD9qm4hd7"
        ],
        "metadata": {
            "experiment_type": "ML learning"
        },
        "id": "kV-cNpQBR7hBR5gtxTfk"
    }
]
```


Remark: Partial matches are also returned, as it is sufficient that one token ("ML" or "learning") is matched. Text is divided into tokens using whitespaces as separators.

8.2 Executed Workflows

The second group of implemented endpoints enables callees to add executed workflows, modify them, and query them.

Each executed workflow contains the following fields:

- Name
- deployedWorkflow - the model snapshot
- Experiment id
- Start
- End
- Metadata
- Comment
- Parameters
- Input datasets
- Metrics
- Output datasets
- Executed tasks

8.2.1 Adding executed workflows

The PUT request method on the /executed-workflows endpoint can be used to add an executed workflow. Similarly to the executed experiments, the server first checks the access token to make sure that the user is authorized to add executed workflows. The payload is then validated, and Elastic Search's index method is used to add the record to the database. Response code 201 or 400 is returned based on the method's return value.

Request

[PUT] http://localhost:8445/api/executed-workflows

Payload


```
{
    "name": "Real-time Log Extraction",
    "experimentId": "VCsVs5AB4eIuD9qm4her",
    "deployedWorkflow": "workflow DataPreprocessing {  define task
    SplitData;  START -> SplitData -> END; configure task SplitData
    { implementation \"tasks/primitives/split_data/split_data.py\";
    } define input data XXX;  define output data YYY;",
    "start": "2024-01-01T00:00:00Z",
    "end": "2024-01-01T12:00:00Z",
    "metadata": {
        "Approach": "1.0"
    },
    "comment": "Initial run of the real-time log extraction workflow",
    "parameters": [
```

```
    {
        "name": "buffer_size",
        "value": "2048",
        "type": "number",
        "usedByExecutedTasks": "log_extraction_task"
    }
],
"input_datasets": [
    {
        "name": "Log Dataset",
        "uri": "uri/log_dataset.csv",
        "usedByTasks": "log_extraction_task",
        "date": "2024-01-01T00:00:00Z",
        "checksum": "0x12345ABC"
    }
],
"metrics": [
    {
        "name": "Extraction Accuracy",
        "semanticType": "accuracy",
        "type": "single-value",
        "value": "0.98",
        "producedByTask": "log_extraction_task",
        "date": "2024-01-01T12:00:00Z"
    },
    {
        "name": "System Load",
        "metadata": {
            "RAM": "32"
        },
        "semanticType": "CPU utilization",
        "type": "timeseries",
        "value": "metric_system_load",
        "producedByTask": "log_extraction_task",
        "date": "2024-01-01T12:00:00Z"
    }
],
"output_datasets": [
    {
        "uri": "uri/processed_logs.csv",
        "type": "csv",
        "producedByTask": "log_extraction_task",
        "name": "Processed Log Dataset",
        "date": "2024-01-01T12:00:00Z",
        "checksum": "0x67890DEF",
        "description": "Processed log data from the cybersecurity
        experiment"
    }
],
"executedTasks": [
    {
```

```
            "id": "log_extraction_task",
            "name": "Log Extraction Task",
            "start": "2024-01-01T00:00:00Z",
            "end": "2024-01-01T12:00:00Z",
            "metadata": {
                "task_type": "log_extraction"
            },
            "comment": "Extracted and processed logs from input dataset",
            "executedWorkflow": "Real-time Log Extraction",
            "source_code": "extraction_code_here",
            "parameters": [
                {
                    "name": "buffer_size",
                    "type": "number",
                    "value": "2048"
                }
            ],
            "input_datasets": [
                {
                    "name": "Log Dataset",
                    "uri": "uri/log_dataset.csv",
                    "date": "2024-01-01T00:00:00Z",
                    "checksum": "0x12345ABC"
                }
            ],
            "metrics": [
                {
                    "name": "Execution Time",
                    "type": "single-value",
                    "value": "6s",
                    "date": "2024-01-01T12:00:00Z"
                }
            ],
            "output_datasets": [
                {
                    "type": "csv",
                    "uri": "uri/processed_logs.csv",
                    "name": "Processed Log Dataset",
                    "date": "2024-01-01T12:00:00Z",
                    "checksum": "0x67890DEF",
                    "description": "Processed log data from the
                    cybersecurity experiment"
                }
            ]
        }
    ]
}
```

Response

Status - 201 Created

```
{
    "id": "kl-iNpQBR7hBR5gtVzcn"
}
```


8.2.2 Modifying Executed Workflows

The POST request method on the /executed-workflows/:workflowId endpoint can be used to modify an executed workflow. Any field from the previous endpoint can be changed, omitted fields are left unchanged.

Request

[POST] http://localhost:8445/api/executed-workflows/kl-iNpQBR7hBR5gtVzcn

Payload


```
{
    "metadata": {
        "version": "3.0",
        "priority": "high",
        "Approach": "1.0"
    }
}
```


Response

200 OK


```
{
    "message": "Executed workflow updated successfully"
}
```


8.2.3 Querying Executed Workflows

The GET request method on the /executed-workflows/:workflowId endpoint can be used to get one executed workflow based on its ID.

Request

[GET] http://localhost:8445/api/executed-workflows/kl-iNpQBR7hBR5gtVzcn

Response


```
{
    "workflow": {
        "name": "Real-time Log Extraction",
        "experimentId": "VCsVs5AB4eIuD9qm4her",
        "deployedWorkflow": "workflow DataPreprocessing {  define task
        SplitData;  START -> SplitData -> END; configure task SplitData
```

```
{ implementation \"tasks/primitives/split_data/split_data.py\";
} define input data XXX;  define output data YYY;",
"start": "2024-01-01T00:00:00Z",
"end": "2024-01-01T12:00:00Z",
"metadata": {
    "Approach": "1.0",
    "priority": "high",
    "version": "3.0"
},
"comment": "Initial run of the real-time log extraction workflow",
"parameters": [
    {
        "name": "buffer_size",
        "value": "2048",
        "type": "number",
        "usedByExecutedTasks": "log_extraction_task"
    }
],
"input_datasets": [
    {
        "name": "Log Dataset",
        "uri": "uri/log_dataset.csv",
        "usedByTasks": "log_extraction_task",
        "date": "2024-01-01T00:00:00Z",
        "checksum": "0x12345ABC"
    }
],
"metrics": [
    {
        "name": "Extraction Accuracy",
        "semanticType": "accuracy",
        "type": "single-value",
        "value": "0.98",
        "producedByTask": "log_extraction_task",
        "date": "2024-01-01T12:00:00Z"
    },
    {
        "name": "System Load",
        "metadata": {
            "RAM": "32"
        },
        "semanticType": "CPU utilization",
        "type": "timeseries",
        "value": "metric_system_load",
        "producedByTask": "log_extraction_task",
        "date": "2024-01-01T12:00:00Z"
    }
],
"output_datasets": [
    {
        "uri": "uri/processed_logs.csv",
```

```
            "type": "csv",
            "producedByTask": "log_extraction_task",
            "name": "Processed Log Dataset",
            "date": "2024-01-01T12:00:00Z",
            "checksum": "0x67890DEF",
            "description": "Processed log data from the
            cybersecurity experiment"
        }
    ],
    "executedTasks": [
        {
            "id": "log_extraction_task",
            "name": "Log Extraction Task",
            "start": "2024-01-01T00:00:00Z",
            "end": "2024-01-01T12:00:00Z",
            "metadata": {
                "task_type": "log_extraction"
            },
            "comment": "Extracted and processed logs from input
            dataset",
            "executedWorkflow": "Real-time Log Extraction",
            "source_code": "extraction_code_here",
            "parameters": [
                {
                    "name": "buffer_size",
                    "type": "number",
                    "value": "2048"
                }
            ],
            "input_datasets": [
                {
                    "name": "Log Dataset",
                    "uri": "uri/log_dataset.csv",
                    "date": "2024-01-01T00:00:00Z",
                    "checksum": "0x12345ABC"
                }
            ],
            "metrics": [
                {
                    "name": "Execution Time",
                    "type": "single-value",
                    "value": "6s",
                    "date": "2024-01-01T12:00:00Z"
                }
            ],
            "output_datasets": [
                {
                    "type": "csv",
                    "uri": "uri/processed_logs.csv",
                    "name": "Processed Log Dataset",
                    "date": "2024-01-01T12:00:00Z",
```

```
                        "checksum": "0x67890DEF",
                        "description": "Processed log data from the
                        cybersecurity experiment"
                    }
                ]
            }
        ]
    }
}
```


8.2.4 Advanced querying executed workflows

The POST request method on the /executed-workflows-query endpoint can be used for more complex queries. Executed workflows can be queried based on the parent executed experiment's ID, a single executed workflow ID, or a range constraint on executed workflow's IDs. The endpoint supports specifying date range constraints and metadata queries. All the constraints are parsed, and a query for Elastic search is created and run. The resulting set of executed workflows conforming to the criteria is returned. The set may be empty.

Request

[POST] http://localhost:8445/api/executed-workflows-query

Payload


```
{
    "experimentId": "VCsVs5AB4eIuD9qm4her",
    "metadata": {
        "version": "3.0",
        "priority": "high",
        "Approach": "1.0"
    }
}
```


Response

200 OK


```
[
    {
        "name": "Real-time Log Extraction",
        "experimentId": "VCsVs5AB4eIuD9qm4her",
        "deployedWorkflow": "workflow DataPreprocessing {  define task
        SplitData;  START -> SplitData -> END; configure task SplitData
        { implementation \"tasks/primitives/split_data/split_data.py\";
        } define input data XXX;  define output data YYY;",
        "start": "2024-01-01T00:00:00Z",
        "end": "2024-01-01T12:00:00Z",
        "metadata": {
```

```
        "Approach": "1.0",
        "priority": "high",
        "version": "3.0"
    },
    "comment": "Initial run of the real-time log extraction workflow",
    "parameters": [
        {
            "name": "buffer_size",
            "value": "2048",
            "type": "number",
            "usedByExecutedTasks": "log_extraction_task"
        }
    ],
    "input_datasets": [
        {
            "name": "Log Dataset",
            "uri": "uri/log_dataset.csv",
            "usedByTasks": "log_extraction_task",
            "date": "2024-01-01T00:00:00Z",
            "checksum": "0x12345ABC"
        }
    ],
    "metrics": [
        {
            "name": "Extraction Accuracy",
            "semanticType": "accuracy",
            "type": "single-value",
            "value": "0.98",
            "producedByTask": "log_extraction_task",
            "date": "2024-01-01T12:00:00Z"
        },
        {
            "name": "System Load",
            "metadata": {
                "RAM": "32"
            },
            "semanticType": "CPU utilization",
            "type": "timeseries",
            "value": "metric_system_load",
            "producedByTask": "log_extraction_task",
            "date": "2024-01-01T12:00:00Z"
        }
    ],
    "output_datasets": [
        {
            "uri": "uri/processed_logs.csv",
            "type": "csv",
            "producedByTask": "log_extraction_task",
            "name": "Processed Log Dataset",
            "date": "2024-01-01T12:00:00Z",
            "checksum": "0x67890DEF",
```

```
            "description": "Processed log data from the
            cybersecurity experiment"
        }
    ],
    "executedTasks": [
        {
            "id": "log_extraction_task",
            "name": "Log Extraction Task",
            "start": "2024-01-01T00:00:00Z",
            "end": "2024-01-01T12:00:00Z",
            "metadata": {
                "task_type": "log_extraction"
            },
            "comment": "Extracted and processed logs from input dataset",
            "executedWorkflow": "Real-time Log Extraction",
            "source_code": "extraction_code_here",
            "parameters": [
                {
                    "name": "buffer_size",
                    "type": "number",
                    "value": "2048"
                }
            ],
            "input_datasets": [
                {
                    "name": "Log Dataset",
                    "uri": "uri/log_dataset.csv",
                    "date": "2024-01-01T00:00:00Z",
                    "checksum": "0x12345ABC"
                }
            ],
            "metrics": [
                {
                    "name": "Execution Time",
                    "type": "single-value",
                    "value": "6s",
                    "date": "2024-01-01T12:00:00Z"
                }
            ],
            "output_datasets": [
                {
                    "type": "csv",
                    "uri": "uri/processed_logs.csv",
                    "name": "Processed Log Dataset",
                    "date": "2024-01-01T12:00:00Z",
                    "checksum": "0x67890DEF",
                    "description": "Processed log data from the
                    cybersecurity experiment"
                }
            ]
        }
```

```
        ],
        "id": "kl-iNpQBR7hBR5gtVzcn"
    }
]
```

8.3 Metrics

The third group of implemented API endpoints are metric endpoints. Endpoints in this group can be used to add a time series metric definition and data to time series metrics. It also provides means to query the time series metric definitions based on the metric ID. Metrics of any type can be queried based on more complex queries. The last functionality of this group is to query data of a metric, making it possible to get only relevant records.

8.3.1 Adding metric with Schema

The PUT request method on the /metrics endpoint can be used to add a time series metric definition. The definition should contain the name of the metric and its schema. Schema specifies the time-series structure using IVIS's signal sets. A schema is an object where keys are different signals and values are their types, such as a keyword or a float. This endpoint is not used to upload the time series data, and the request will result in error response code 400, the same as if the metric name or schema is not specified. Other metric fields including metadata and id of the task which produced the metric are stored in the executed workflow which created it. IDs of the executed experiment and the specific executed workflow which created it are returned by advanced queries described in the section 8.3.3.

Request

[PUT] http://localhost:8445/api/metrics

Payload

```
{
    "name": "metric\_system\_load",
    "schema": {
        "time": "string",
        "utilization": "float"
    }
}
```

Response

200 OK

```
"Metric created successfully!"
```

8.3.2 Querying the Metric Definitions

The GET request on the /metrics/:metricId endpoint can be used to query metric definitions. The last part of the endpoint's URL is the metric name used when adding the metric. If the metric is not found, error response code 400 is returned. Otherwise, the metric name and its schema are returned.

Request

[GET] http://localhost:8445/api/metrics/metric_system_load

Response

200 OK


```
{
    "name": "metric_system_load",
    "schema": {
        "time": "keyword",
        "utilization": "float"
    }
}
```


8.3.3 Advanced metric querying

The POST request on the /metrics-query endpoint can be used for advanced metric queries. Following query attributes, and their arbitrary composition, are supported:

- ExperimentId
- WorkflowId
- Metadata
- Type - e.g., timeseries or single-value
- SemanticType - e.g., accuracy, corresponds to metric group
- Value - based on the type may be its value or a reference to time series data

Executed workflows are queried and metric passing the criteria are returned. Following fields are returned:

- Name
- SemanticType
- Type
- Value
- ProducedByTask

- Date
- WorkflowId
- ExperimentId

Request

[POST] http://localhost:8445/api/metrics-query

Payload


```
{
    "semanticType": "CPU utilization",
    "type": "timeseries",
    "value": "metric_system_load",
    "workflowId": "kl-iNpQBR7hBR5gtVzcn",
    "experimentId": "VCsVs5AB4eIuD9qm4her",
    "metadata": {
        "RAM": "32"
    }
}
```


Response

200 OK


```
[
    {
        "name": "System Load",
        "metadata": {
            "RAM": "32"
        },
        "semanticType": "CPU utilization",
        "type": "timeseries",
        "value": "metric_system_load",
        "producedByTask": "log_extraction_task",
        "date": "2024-01-01T12:00:00Z",
        "workflowId": "kl-iNpQBR7hBR5gtVzcn",
        "experimentId": "VCsVs5AB4eIuD9qm4her"
    }
]
```


8.3.4 Adding data to time series metrics

The PUT request on the /metrics/:metricId endpoint can be used to upload data points to the specified time-series metric. Data points can be uploaded in batches by passing an array of objects. Each object must contain its unique ID and may contain any fields from previously specified schema. If an entry contains additional fields not specified in the schema, error response code 400 is returned, and callee is let known.

Request

[PUT] http://localhost:8445/api/metrics-data/metric_system_load

Payload


```
{
    "data": [
        {
            "id": "1",
            "time": "2024-06-01T12:27:05.284Z",
            "utilization": 0.75
        },
        {
            "id": "2",
            "time": "2024-06-01T12:27:06.284Z",
            "utilization": 0.1
        }
    ]
}
```


Response

200 OK

```
"Records added successfully!"
```

8.3.5 Querying time series metric data

The POST request method on the /metrics-query/:metricId endpoint can be used to query time-series metric data. The payload of each request must contain an array specifying which attributes (a.k.a signals) the callee is interested in from each record. A count can be specified to specify the maximal number of data points. The start date and the time attribute can be specified to request records that have the attribute's value higher than the specified date. The records are expected to be uploaded sequentially, so the first record with a higher than or equal to the given date is selected, not the record with the lowest date higher than or equal to the given date.

Request

[POST] http://localhost:8445/api/metrics-data/metric_system_load

Payload


```
{
    "startDate": "2024-06-01T12:27:06.284Z",
    "count": 1,
    "timeAttribute": "time",
    "attribute": ["time", "utilization"]
}
```

Response

200 OK


```
[
    {
        "_id": "2",
        "time": "2024-06-01T12:27:06.284Z",
        "utilization": 0.1
    }
]
```


8.4 Validation Algorithm

The validation algorithm traverses through the payload, and for each field, it checks that it is expected by the schema. Each field's type is checked against the type defined in the schema. If a field's value is an object or array, the value is recursively checked as well, providing in-depth schema validation. Because of the design, a payload may miss some fields, but it is forbidden to contain unexpected fields. In case of unexpected fields or wrong types, the response's code is set to 400, and the response's payload contains the field that caused it and a description of the schema breach. Allowing to omit fields can be used to update only part of the entity (e.g, only metadata of executed workflow).

9 Deployment and Evaluation

This chapter discusses three deployment possibilities, reasons why Docker was selected, and which work was done by this thesis to enable deployment of the data service using Docker. Section 9.5 provides evaluation.

After designing and implementing the data service, an optimal deployment method had to be selected. The main criteria were performance and scalability, as the system will have to adapt to an increasing number of users, and executed experiments. Other factors, such as portability, start time, and future support outlook, also had to be taken into account.

Several deployment methods were considered, and their advantages and disadvantages were compared. In the end, the Docker deployment method was deemed to be the best fit for our purpose and chosen for its performance and scalability.

9.1 Deployment Possibilities

In this section, various deployment possibilities are introduced and compared based on their advantages and disadvantages. Then, the best deployment approach is selected to provide high performance, scalability, portability, and long-lasting support.

9.1.1 Bare-metal deployment

The most intuitive approach is bare-metal deployment. Bare-metal servers do not use an additional layer with a hypervisor or other virtualization software. The user has complete control over the server's infrastructure. This option only supports one tenant, and resources cannot be shared, resulting in smaller overall utilization and is a cost-ineffective solution [31].

However, this approach also has advantages, such as reduced latency, as there is no additional layer between an application and the underlying infrastructure. Another advantage is added security because of physical isolation. Physical isolation also removes The Noisy neighbor problem, leading to more consistent performance.

9.1.2 Virtual Machines

The second deployment variant is to use a virtual machine. Virtual machines emulate a computer system. Applications do not communicate directly with the physical computer but rather with its simulation, which is provided by a hypervisor. The hypervisor may manage multiple VMs on a single physical machine.

This approach has multiple benefits. It provides isolation to the application, which leads to more robust security and resolves issues with various versions of dependencies of different applications. Also, each virtual machine has its own copy of an operating system, which can differ across the virtual machines. Virtual machines provide compatibility as the host operating system (OS) and guest OS can differ. Each virtual machine can be tailored for the application and contain its dependencies with the requested versions. A big advantage of VMs is portability, enabling to move them easily between various hosts. This enables enhanced scalability, as more VMs can be deployed to handle higher traffic.

The disadvantages of this approach include higher overhead for memory and storage operations. Also, the VMs are large, as they contain an operating system. Consequently, the VMs have a long start time. Virtual machines are useful for legacy applications, testing, and development. They are also useful when complete OS-level isolation is required for compatibility or security. Virtualization softwares include VMWare Workstation and Hyper-V.

9.1.3 Containerization

The third deployment approach is containerization. Containerization is a modern and lightweight approach. Similarly to traditional virtual machines, containerization ensures that the application is independent of the underlying platform. The difference between virtual machines and containers is that containers do not contain their operating system but rather use the host OS abstracted by the containerization engine, which ensures that the applications run independently on the underlying infrastructure. Figure 9.1 [1] shows a comparison of the Virtual machine's approach and Containerization.

A container contains an application and its dependencies. As containers do not contain an OS, they are lightweight, resulting in faster start times than VMs. However, containers provide weaker isolation compared with previous approaches.

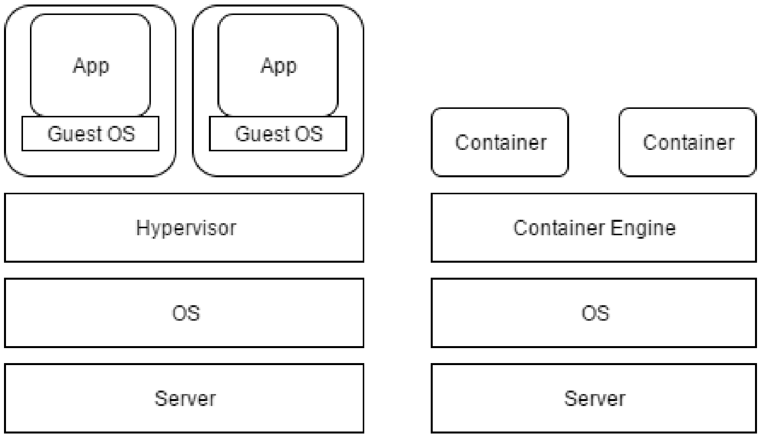

Figure 9.1 A comparison of Virtual Machines Approach (left) and Containerization (right).

9.2 Deployment method selection

An example of a platform enabling containerization is Docker [32], which was selected as the optimal deployment method. Docker has better CPU performance and load speed than virtual machines [33] and promises long-lasting support in the foreseeable future. It also provides all the advantages of containerization. It

[1]Figure 9.1 was taken from https://www.researchgate.net/figure/rtual-Machine-vs-Container_fig4_311426878

enables the separation of an application from an infrastructure, resulting in faster software delivery.

9.3 Deployment with Docker

To deploy an application using Docker, the Docker image with the application first needs to be created. Then, the application is deployed by running a Docker container based on the Docker image.

9.3.1 Docker Images

Docker images do not have to be created from scratch, but a base image can be used as a starting point. The Docker hub contains various base images, which may contain a language runtime, minimal operating system, or be tailored for a specific framework. A custom image can be created by creating a docker file, which is a file containing a specification of a base image, a list of commands for the image creation, a list of files to be copied, and a command to be executed after a container based on the image is run. Also, a docker file may contain a declaration of a working directory.

A container could also be created directly from the base image and modified to the desired state. However, that is poor practice, as it removes repeatability, one of Docker's main advantages.

9.3.2 Docker Compose

Docker Compose is a container orchestration tool for running an application with multiple containers. IVIS, for example, uses a container for the server, a container for elastic search, a container for Redis, and a container for MariaDB. The deployment process is automized with docker-compose, which runs containers for all three parts and defines their relationship.

9.3.3 Docker Engine Architecture

The Docker engine uses a client-server architecture. The client enables interactions with a user using the command line interface (CLI). The client then communicates via REST API with the Docker host, which is running Docker Deamon. Docker Deamon enables the creation of docker objects (e.g., images, containers) and their modifications. Docker Deamon also communicates with Docker Hub to resolve base images [34].

9.3.4 Data persistency

When working with Docker, data persistency needs to be resolved. By default, data are stored only on the container's writable layer. Data are stored only as long as the container exists. The following scenario illustrates this.

1. A container is created based on an image

2. The container runs and gathers data

3. The container is removed

4. Another container based on the same image is created

In this scenario, data would not persist as the second container is an entirely separate container from the first one, even though both are based on the same image. Docker solves this problem by introducing volumes [35]. Volumes enable the data to be stored on the Docker host. When running a container, the volume to be used can be specified. Multiple containers can use the same volume, thus enabling data sharing. The volumes need to be pruned to remove unused ones, as they are not removed automatically if no active container uses them due to keeping the persistence. Volumes also enable the storage of data remotely. Figure 9.2 shows how the volumes are defined in Docker Compose and bind with the service's containers.

```yaml
services:
  mariadb:
    image: mariadb:latest
    volumes:
      - mariadb_data:/var/lib/mysql

  redis:
    image: redis:latest
    volumes:
      - redis_data:/data

  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:8.14.3
    volumes:
      - elasticsearch_data:/usr/share/elasticsearch/data

volumes:
  mariadb_data:
  redis_data:
  elasticsearch_data:
```

Figure 9.2 An example of defining volumes and binding them with containers using the Docker compose orchestration tool.

9.4 Deployment Approaches Conclusion

In this chapter, three approaches to deployment were introduced.

The first possibility was to use the bare-metal approach, where the application runs directly on physical hardware. This leads to the best performance as there is no virtualization overhead. It also gives full access to the hardware. Security is strengthened as bare metal accepts only a single tenant, leading to physical isolation. However, the disadvantages are difficult portability and scalability, as manual effort is required.

The second approach was to use virtual machines. VMs communicate with a hypervisor, which simulates physical hardware. Because of that, each operation introduces overhead. However, resource utilization is higher because a single hypervisor can host multiple VMs. Each virtual machine contains its own operating system, which leads to better portability, but it also increases its size to gigabytes and its startup time to minutes.

The third approach was to use containerization, which also uses virtualization. The difference from VMs is that containers do not contain an instance of an operating system but rather communicate with the host OS via the containerization engine, which provides the virtualization. Containers are lightweight, and their start time is only a few seconds. Containers offer high scalability. Docker is a containerization technology that enables the building and running of containers. Docker uses client-server architecture. The user communicates with the Docker client via CLI, and the client sends REST API requests to the Docker server, which uses the Docker daemon to create and manage docker objects. The drawback of using containerization is weaker isolation from other applications.

After this analysis, Docker was selected as the deployment method because it provides scalability and portability and has an active community that ensures support in the foreseeable future.

9.5 Evaluation

The purpose of this thesis was to gather requirements from our project partners, propose the data service to store data about executed experiments, and implement the data service. The data service includes the API to store and query the executed experiments and metrics.

The work done on this thesis led to a deep understanding of the experiment data and their relations. The important analysis steps done by this thesis are:

- Data categories
- Data model
- API use case categories

Following the analysis, this thesis provided a discussion about various data storage possibilities, and Elastic search was selected as the optimal database implementation. A script was created that configures the indices to store executed experiments data, including computed metrics.

The thesis provides the implementation of the REST API, which contains 13 endpoints enabling to add executed experiments and executed workflows, modify them and query them based on their id or based on advanced queries, specifying metadata and their values. Metrics with simple type results can be stored directly in the executed workflows. For time series metrics, this thesis implemented endpoints to define the time series metric's schema and add data. Data points can be queried to get only the relevant data points. An endpoint enabling to query all metrics based on various fields was also implemented.

This thesis also prepared necessary Docker-related files, such as Dockerfile and docker-compose, and created a container on the designated server. This provides project partners with access to the API.

The research team from Charles University then extended the API done by this thesis. The extended API is currently used by the following partners.

- Vrije Universiteit Amsterdam (VU) uses the API to store, query, and retrieve experiments, workflows, and metrics through their graphical editor.
- Athena Research Center (Athenarc) uses the API to retrieve experiments, workflows, and metrics for visualization
- Interactive 4D uses the API to store and retrieve metrics
- National Technical University of Athens (NTUA) uses the API to retrieve experiments

10 Related work

This chapter introduces related work. This thesis designs and implements a data service for data about executed experiments for the ExtremeXP experimentation framework. Other experimentation frameworks and their experimentation data services were examined to compare the data service with related work.

The first experimentation tool is MLflow, which is designed to simplify machine learning algorithm development. A Python API is provided to be simply used in a Python code [36]. It also offers its Rest API, providing means to store experiments, runs, metrics, datasets, and artifacts. MLflow's main element is the experiment, which is used to group multiple runs. Each experiment contains its ID, name, tags (metadata), creation and last update timestamps, lifecycle stage, and artifact location. An ML algorithm execution is called a run and is linked with parameters, metrics, and datasets [37].

The second experimentation tool is ClearML, which is also designed to simplify machine learning algorithm development. The REST API is provided to manage projects linked with tasks, models, and datasets [38]. Endpoints to manage worker machines are also provided. Events endpoints can be used to retrieve metrics and debug samples.

Other tools for machine learning include Neptune.ai, Comet ML, and W&B.

10.1 Differences in data service purpose

The data service designed and implemented by this thesis differs from the data services of the mentioned frameworks in that it aims to support storing data from any kind of experiments, not just machine learning experiments. Also, some of these frameworks are designed to run locally with a small number of users and executed experiments. The ExtremeXP's data service, on the other hand, aims to provide an efficient solution even for a higher number of users and executed experiments.

10.2 Used technologies comparison

The ExtremeXP's data service relies on Elastic Search, a NoSQL database search engine. This means that the data storage can be distributed amongst multiple nodes and scaled horizontally. MLflow uses a tracking server, which uses SQLAlchemy, a Python library containing an object-oriented mapper for interaction with databases, to store data in an SQL database such as PostgreSQL. While SQLAlchemy supports horizontal sharding, it may lead to degraded query performance. This thesis puts emphasis on scalability and performance, both of which Elastic Search provides. By storing data in Elastic Search instead of PostgreSQL, we achieve significantly higher query performance [39].

ClearML stores experiment metadata and metrics in a MongoDB database. MongoDB is a NoSQL database, which provides CRUD[1] operations. Even though

[1] CRUD is an abbreviation for four data management operations - create, read, update, and delete.

MongoDB is faster for CRUD operations, Elastic search is faster for full-text queries [40].

10.3 Data model and endpoint comparison with MLflow

ExtremeXP's data service differs from MLflow in that it allows workflows to be decomposed into tasks. This allowed for more concrete information about where parameters our datasets were used, where output datasets were produced, and where metrics were computed.

We also do not allow the deletion of stored data. MLflow enables deleting executed experiments. While the executed experiments may be restored, deleting experiments interferes with traceability, as the experiments may contain important metadata.

MLflow's metrics are key-value pairs, where the key is a string and the value is a float value. They also contain date and step attributes. This means that no metadata can be directly linked with a metric. As metrics are linked with runs, and a run can contain metadata, a metric's metadata can be stored there, but this means that it will not be retrieved when asking for metrics.

In comparison, metrics in the data service, designed and implemented by this thesis, are more complex and designed for general usage. Each metric contains a specification of its type, enabling also non-float values. This enables metrics such as confusion matrices, confidence intervals, or storing the units with values, e.g., 15s or 85%. Metric metadata is a direct part of metrics and is retrieved during requests on the metrics-query endpoint leading to optimizing the number of necessary requests.

MLflow enables getting a list of all values of a metric, which can be beneficial, for example, for plotting accuracy changes. Our data service also provides means to achieve this behavior either by declaring the metric's type as time-series and storing the values or by storing the previous values in metadata.

10.4 Related work conclusion

in conclusion, while multiple experimentation frameworks with data services for management experiments and related data, including metadata, exist, they are designed solely for machine learning and smaller usages, so they are not fit for ExtremeXP purpose, as the data service designed by this thesis provides support for more general cases and also the data storage was selected to be efficient even with higher traffic of users and experiments.

11 Conclusion

The task of this thesis was to participate in the ExtremeXP project, which aims to provide tools to help with the experimentation process. The framework supports designing an experiment, running it, and analyzing results. Designing an experiment creates its model, which is stored in the EMF-based model repository. The experimentation engine is used to run the experiments, and Oracle is consulted to propose experiment modifications such as parameter values. ExtremeXP uses its human-in-the-loop approach, where the user is not only shown the results but may participate during experimentation, increasing the trustworthiness and value of its results to the user. Each experiment may produce output datasets and compute metrics.

Traceability also supports the system's trustworthiness. To achieve traceability, each produced output dataset and computed metric must contain a trace to the experiment specification.

This thesis's main task was to design and implement the data service for storing data about executed experiments. The framework context needed to be analyzed, and stakeholder requirements were gathered. The analysis yielded the following entities. Executed workflow is the main process, which takes input datasets and parameters. The executed workflow results are a set of output datasets and computed metrics. The second entity is the executed task, which can be used to divide the process into smaller parts, supporting reusability. More complex tasks can be modeled as a workflow for further decomposition. A set of workflows produces an experiment.

The experiment specification was established during the system analysis. Workflow models, inputs, and parameter values must be stored to recreate the experiment. It is important that modifications to the model in the model repository would break the traceability. The second option would be for the model repository to enable versioning. But as this would bring complications, it was decided that each executed workflow would contain a snapshot of its model. This brings higher spatial requirements but ensures that the model's necessary condition for traceability is achieved. There are more necessary conditions. Traceability is achieved as long as the datasets, which may be externally managed, do not change. The checksum of each dataset is stored with each dataset reference. If a checksum does not match the dataset, the traceability is broken as the dataset was changed. An important remark is that traceability does not equal reproducibility, as heuristics in the experiments may lead to different results. However, the metrics to which the law of large numbers and the Central theorem apply, such as accuracy. will give approximately the same results.

The analysis included the study of the metrics. Metrics may be task-independent or task-dependent. Task-independent metrics do not need access to the inner workings of a task. Examples of such metrics are used hardware resources. Task-dependent metrics can be further divided into output-independent and output-dependent. An example of a task-dependent output-independent metric is the time of one training cycle. An example of a task-dependent and output-dependent metric is precision.

This division of examples leads to the analysis of the possible types of metric results. A metric may produce one scalar value, a vector of values, a time series, or an uninterpreted BLOB. Time series can easily lead to large data volumes, which need to be taken into account during the database selection. The API needed to support data series queries to allow only relevant data points to be collected, as the time series can contain a large number of entries.

The analysis resulted in a data model later used as a base for database indices and their schema.

Various scenarios were collected during the API design and divided into the following categories:

- Experiments overview
- Data overview
- Experiment result analysis
- Experiment description
- Reliability or reproducibility of an approach
- Comparison of multiple approaches
- Traceability

The API was designed to be efficient. Relevant data might be obtained in as few queries as possible, but the queries would also not return much irrelevant data. The API design also supports traceability queries.

The IVIS framework, developed at Charles University, was used as a starting point for implementation. IVIS is a framework for creating web applications with data visualizations and analytics. It is a server-client application. The client uses the Javascript library React, and the server is built on NodeJS, allowing Javascript to run on the server side. IVIS was not updated for three years, and its dependencies were deprecated. The first task of creating the data service was to update the dependencies. Various dependencies brought breaking changes.

Especially big changes were introduced by the React router, which introduced hooks that enable functional components to keep their state and access their lifecycle methods. Route matching is used for rendering correct components. Hooks were introduced for obtaining routing properties, and the withRouter HOC was removed. This resulted in changes in recommended programming approaches, as the hooks are only callable from functional components, whereas class components were more used previously.

Data storage implementation started with the selection of the DBMS. The traditional SQL approach was compared with the newer NoSQL approach. SQL stores data in a tabular manner where attributes correspond to the columns and entries to the rows. SQL supports vertical scaling, where the system can adapt to higher traffic by giving more resources to the server. On the other hand, NoSQL uses different structures such as key-value pairs, WCS, graphs, or documents. NoSQL systems may be distributed and can use the benefits of horizontal scaling. The system needs to be prepared for the rise in the number of users and the number of executed experiments. For these reasons, the Elastic search was selected to store data about executed experiments and computed metrics. Various indices were then implemented, including Executed experiments, executed workflows, executed tasks, and metrics.

The data access layer needed to be implemented on top of the data storage to provide project partners with a way to easily store and retrieve data without knowing the inner workings of the data service. The data storage was hidden from the partners, so they did not need to weigh in on the selected DBMS. In the case of the API, they were part of the discussion, and a clear choice for the API's protocol was the REST API protocol.

A collection of endpoints to show the API design was implemented, including endpoints for executed experiments, executed workflows, metric definitions, and data of time-series metrics. During a request to any endpoint, the server first checks the authentication token and ensures that the callee is authorized for the requested operation. Then it checks the request fields, and if the request conforms to the expected schema, the server continues by communicating with the Elasticsearch database and returns information whatever the operation was successful.

After the implementation, the deployment method was selected. Possible deployment methods included bare-metal deployment, virtual machines, and containers. The advantages and disadvantages of each approach were discussed. The bare-metal approach provides physical isolation but does not provide needed scalability and portability. Virtual machines use virtualization to abstract communication with the physical machine. Each virtual machine contains its own OS, making the application independent of the host OS. The drawback of virtual machines is their size and start time caused by the containment of the operating system. Containerization is an approach similar to virtual machines, but a container does not contain an instance of OS; rather, it uses a containerization engine to virtualize communication with the host OS. Therefore, the containers are lightweight and have a significantly faster start time. Docker, which was selected as the optimal deployment method, is an example of a platform enabling to use containerization principles. Deploying the application can be automized by using Dockerfiles to create a docker image and orchestrated using Docker Compose or Kubernetes. Docker compose was used for the data service to start Elastic search, mariaDB, and Redis containers based on the images from the Docker hub.

In conclusion, this thesis successfully gathered requirements from stakeholders, analyzed which data needs to be stored by the data service, designed the API based on an analysis of use cases, and selected the optimal data storage system. The IVIS framework was updated as the dependencies were deprecated. Thirteen endpoints from the API were also implemented as part of this thesis. As the update of IVIS took longer than expected, other designed endpoints were implemented by other ExtremeXP team members to meet deliverable deadlines. Docker was selected as the optimal deployment method, based on scalability, portability, and start time. The deployment was automated with the necessary files (dockerfile and docker-compose.yml).

Bibliography

1. THORNTON, Chris; HUTTER, Frank; HOOS, Holger H; LEYTON-BROWN, Kevin. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2013, pp. 847–855.
2. [https://i2cat.net/projects/extremexp/]. [N.d.]. Accessed: 10.11.2024.
3. [https://d3s.mff.cuni.cz/projects/extremexp/]. [N.d.]. Accessed: 10.11.2024.
4. RAJENTHIRAM, Keerthiga. Optimizing Data Analytics Workflows through User-driven Experimentation. In: *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering-Software Engineering for AI*. 2024, pp. 253–255.
5. LARNIER, Kévin; COVES, Jacques; STÉPHAN, Gwendoline; DELPORTE, Pauline; DUMAS, Loïc; BOULFANI, Yasmine. Urban Flash Flood Prediction through High-Resolution Hydrodynamic Modeling and Machine Learning Approaches. In: *IGARSS 2024-2024 IEEE International Geoscience and Remote Sensing Symposium*. IEEE, 2024, pp. 3757–3760.
6. LAGO, Patricia; MUCCINI, Henry; VAN VLIET, Hans. A scoped approach to traceability management. *Journal of Systems and Software*. 2009, vol. 82, no. 1, pp. 168–182.
7. WANG, Fusheng; LIU, Peiya; PEARSON, John; AZAR, Fred; MADLMAYR, Gerald. Experiment management with metadata-based integration for collaborative scientific research. In: *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 2006, pp. 96–96.
8. ZHENG, Zibin; XIE, Shaoan; DAI, Hong-Ning; CHEN, Xiangping; WANG, Huaimin. Blockchain challenges and opportunities: A survey. *International journal of web and grid services*. 2018, vol. 14, no. 4, pp. 352–375.
9. SIRIGNANO, Justin; SPILIOPOULOS, Konstantinos. Mean field analysis of neural networks: A law of large numbers. *SIAM Journal on Applied Mathematics*. 2020, vol. 80, no. 2, pp. 725–752.
10. LOVAS, Attila; RÁSONYI, Miklós. Functional Central Limit Theorem and Strong Law of Large Numbers for Stochastic Gradient Langevin Dynamics. *Applied Mathematics & Optimization*. 2023, vol. 88, no. 3, p. 78.
11. [https://v8.dev/docs]. [N.d.]. Accessed: 7.12.2024.
12. PASQUALI, Sandro; FAABORG, Kevin. *Mastering Node. js: build robust and scalable real-time server-side web applications efficiently*. Packt Publishing Ltd, 2017.
13. [https://www.npmjs.com/]. [N.d.]. Accessed: 7.12.2024.
14. FINK, Gil; FLATOW, Ido; FINK, Gil; FLATOW, Ido. Introducing single page applications. *Pro single page application development: Using backbone. js and asp. net*. 2014, pp. 3–13.

15. Boduch, Adam. *React and react native*. Packt Publishing Ltd, 2017.
16. [https://reactrouter.com/6.28.0/upgrading/v5]. [N.d.]. Accessed: 30.11.2024.
17. [https://gist.github.com/mjackson/d54b40a094277b7afdd6b81f51a0393f]. [N.d.]. Accessed: 24.11.2024.
18. [https://gist.github.com/mjackson/b5748add2795ce7448a366ae8f8ae3bb]. [N.d.]. Accessed: 23.11.2024.
19. Dieu, Guillaume; Lerat, Jean-Sébastien; Cremer, Samuel. Relational Wide Column Store: a new Big Data paradigm. In: *Proceedings of the 20th ACM International Conference on Computing Frontiers*. 2023, pp. 211–212.
20. [https://www.ibm.com/topics/cap-theorem]. [N.d.]. Accessed: 24.11.2024.
21. Jowan, Salah A; Swese, Ramadan Faraj; Aldabrzi, Antisar Yousf; Shertil, Mahmood Saad. Traditional RDBMS to NoSQL database: new era of databases for big data. *Journal of Basic Sciences*. 2016, vol. 29, pp. 83–102.
22. Januzaj, Ylber; Ajdari, Jaumin; Selimi, Besnik. DBMS as a Cloud service: Advantages and Disadvantages. *Procedia-Social and Behavioral Sciences*. 2015, vol. 195, pp. 1851–1859.
23. Nathan, Senthil; Govindarajan, Chander; Saraf, Adarsh; Sethi, Manish; Jayachandran, Praveen. Blockchain meets database: Design and implementation of a blockchain relational database. *arXiv preprint arXiv:1903.01919*. 2019.
24. Faerber, Franz; Kemper, Alfons; Larson, Per-Åke; Levandoski, Justin; Neumann, Thomas; Pavlo, Andrew, et al. Main memory database systems. *Foundations and Trends® in Databases*. 2017, vol. 8, no. 1-2, pp. 1–130.
25. Masse, Mark. *REST API design rulebook*. " O'Reilly Media, Inc.", 2011.
26. Wlodarczyk, Tomasz Wiktor. Overview of time series storage and processing in a cloud environment. In: *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*. IEEE, 2012, pp. 625–628.
27. Kuć, Rafal; Rogoziński, Marek. *Mastering elasticsearch*. Packt Pub., 2013.
28. [https://www.elastic.co/guide/en/elasticsearch/reference/current/size-your-shards.html]. [N.d.]. Accessed: 5.12.2024.
29. [https://www.elastic.co/blog/how-many-shards-should-i-have-in-my-elasticsearch-cluster]. [N.d.]. Accessed: 5.12.2024.
30. Kuć, Rafał; Rogozinski, Marek. *ElasticSearch server*. Packt Publishing Ltd, 2016.
31. Kominos, Charalampos Gavriil; Seyvet, Nicolas; Vandikas, Konstantinos. Bare-metal, virtual machines and containers in OpenStack. In: *2017 20th conference on innovations in clouds, internet and networks (icin)*. IEEE, 2017, pp. 36–43.
32. [https://docs.docker.com/]. [N.d.]. Accessed: 17.11.2024.
33. Potdar, Amit M; Narayan, DG; Kengond, Shivaraj; Mulla, Mohammed Moin. Performance evaluation of docker container and virtual machine. *Procedia Computer Science*. 2020, vol. 171, pp. 1419–1428.

34. Patra, Manoj Kumar; Sahoo, Bibhudatta; Turuk, Ashok Kumar. Containerization in cloud computing for OS-level virtualization. In: *Advances in Cyber Security and Intelligent Analytics*. CRC Press, 2022, pp. 261–276.

35. Smith, Randall. *Docker Orchestration*. Packt Publishing Ltd, 2017.

36. Zaharia, Matei; Chen, Andrew; Davidson, Aaron; Ghodsi, Ali; Hong, Sue Ann; Konwinski, Andy; Murching, Siddharth; Nykodym, Tomas; Ogilvie, Paul; Parkhe, Mani, et al. Accelerating the machine learning lifecycle with MLflow. *IEEE Data Eng. Bull.* 2018, vol. 41, no. 4, pp. 39–45.

37. [`https://mlflow.org/docs/latest/rest-api.html`]. [N.d.]. Accessed: 6.1.2025.

38. [`https://clear.ml/`]. [N.d.]. Accessed: 7.1.2025.

39. Zhang, Hongping; Huang, Wei; Ding, Lei; Tang, Dejin; Wang, Cong; Xu, Yong; Wang, Zhen. Construction and Application of Place Name and Address Management System Based on Elasticsearch. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*. 2024, vol. 48, pp. 571–576.

40. Kathare, Nikita; Reddy, O Vinati; Prabhu, Vishalakshi. A comprehensive study of Elasticsearch. *International Journal of Science and Research (IJSR)*. 2020.

List of Figures

List of Abbreviations

API Application programming interface

ABAC Attribute-based access control

DBMS Database management systems

BLOB Binary large object

SGLD Stochastic Gradient Langevin Dynamics

DOM Document object model

NPM Node package manager

SPA Single-page application

HOC Higher-order component

WCS Wide-column store

DLT Distributed Ledger Technology

CLI Command line interface

OS Operating system

I/O, IO Input and output

REST Representational state transfer

URI Uniform Resource Identifier

HTTP Hypertext Transfer Protocol

A Attachments

A.1 Application deployement

The following steps are needed to deploy the application on Ubuntu to test it locally. In case of any issue please consult the paragraph ”possible issues” which is provided bellow.

1. Clone the repository from https://gitlab.mff.cuni.cz/d3s/ivis/extivis
2. Initialize ivis-core git submodule
3. Make sure that server/config contains development.yaml file
4. Get wait-for-it.sh script from https://raw.githubusercontent.com/vishnubob/wait-for-it/master/wait-for-it.sh
5. Build the Docker image
   (a) Update the EXPVIS_REPO_PATH variable in the path.env file
       i. It must be an absolute path
   (b) sudo docker compose –env-file path.env build –no-cache
6. Create the containers
   (a) sudo docker compose –env-file path.env up
7. Create the database indices
   (a) Enter the container with sudo docker exec -it extivis-expvis-1 bash
   (b) (If changed) Modify ES_HOST variable in /setup/create_indexes.sh
   (c) Run the script with ./setup/create_indexes.sh
8. Create authentication token
   (a) Open http://localhost:8443/login in the browser
   (b) The username is “admin”, and the password is “test”
   (c) Click on “Account”
   (d) Click on “API”
   (e) Click on “Generate access token”.
   (f) Applications like Postman may be used to test the API
   (g) The access token must be put in the request’s header ”access-token”

Possible issues

1. Building Docker image gets stuck on command cd ivis-core/server && npm install
    - Cause: submodule is not fully initialized
    - Solution: Please run npm install in ivis-core/server directory
2. Wait-for-it script timeouts before establishing a connection with Elasticsearch or MariaDB
    - Cause: This may be caused by a change of container names.
    - Solution: Docker-compose.yml needs to be updated in such cases.
    - Container names are used in Expvis -> environment and Expvis > Command
3. Logging in leads to network problems
    - The developer console shows "Access has been blocked by CORS policy."
    - Cause: This is caused by replacing localhost with 127.0.0.1
    - Solution: http://localhost:8443/login
4. POST request to Executed-experiments-query return response code 500
    - Solution: Step 6 prevents these errors, such as the following, thrown by metadata queries