University of Groningen

Faculty of Science and Engineering

Tracking provenance of change in data science pipelines

Enabling reproducibility and provenance of results by tracking the evolution of data in (data science) pipelines

A thesis submitted in fulfillment of the requirements for degree of Master of Science in Computing Science

Author:
BSc. Mark Timmerman

First Supervisor:
prof. dr. Dimka Karastoyanova

Second Supervisor:
dr. George Azzopardi

July 2020

Abstract

One of the foundations for academic research is the ability to reproduce experiments. However, the data used for an experiment, or the computations performed during the experiment, might evolve over time, yielding different results for repeated experiments. It is therefore paramount that the versions of data and computations are tracked properly to ensure reproducible experiments.
Another foundation is that results of an experiment can be traced back to its origin. How data evolves during an experiment, and what elements need to be tracked to know this, are important factors in being able to achieve this provenance.
This research explores the different aspects related to reproducibility and tracking of provenance in data science pipelines. Existing research and tools related to these subjects are examined and summarized. The findings are used to design an architecture for a framework which assists users in tracking the provenance of data in data science pipelines. The framework is designed in such a way that it can track versions of data sets, source code, configurations of their pipelines and evolution of data throughout an experiment. From the designed architecture, a proof-of-concept implementation is introduced, named Iterum. The validity of the framework is evaluated by performing two experiments based on use cases from the domain of data science.
The experiments show that the framework achieves its goals, though it requires the provided abstractions to be used properly.

Acknowledgements

I would like to thank:

Dimka Karastoyanova for supervising this project, giving feedback, suggestions and generally being available for questions.

Edser Apperloo for the fun and constructive collaboration on this project.

Eline Timmerman-de Haan for supporting me, and for being a listening ear.

George Azzopardi for general assistance and being a second supervisor to this project.

Estefania Talavera for providing the *Egocentric photo-stream* use case, assisting with the implementation, and being available for questions.

Ahmad Alsahaf for providing the *FeatBoost* use case, assisting with the implementation, and being available for questions.

List of Tables

List of Figures

Glossary

DAG Directed Acyclic Graph - A graph structure consisting of nodes connected by directed edges, without forming cycles. v, vi, 5, 15

data lineage Tracking how data changes throughout a pipeline. This enables connecting output of a pipeline to various intermediate versions of the original data point(s) responsible for that output. Being able to retrace the origins of data using the data lineage is an important aspect of data provenance. 9

fragment A fragment is a layer of abstraction on top of actual data. It defines a chunk of the data, which can be a single file, multiple files or parts of a file. It serves two purposes: the first is that it allows the user to optimize for network traffic by grouping small pieces of data together in one bundle. Secondly, it allows the user to group data that belongs together in a way that is not limited by issues such as different file types (E.G. one fragment could contain both *some_image.jpg* and a corresponding *metadata.json*). v

fragment description A fragment description is the metadata part of a *fragment*, which does not contain the actual data of the fragment, but a reference to it. This is an abstraction used for the implementation of *Iterum*. 25

fragment streams Fragment streams are streams of *fragments* than flow through a pipeline and are (ideally) individually processable. 12

iterum The name given to the implementation of the framework introduced in this document. The word *iterum* is Latin for "again" or "a second time". v, 24

pipeline A set of data transformation steps designed to perform some computation, in which each step performs part of that computation before passing the transformed data on to the next step(s). In this research, pipelines can be structured as a *DAG*. Here each node is a transformation step and each (directed) edge denotes the direction of data flow. vi, 1, 5

pipeline configuration The configuration of a pipeline which consists of the different versions of the transformation steps, the version of the fragmenter, and any extra configuration the user has provided for the pipeline. In short, the (formal) specification regarding a pipeline excluding any execution details such as the data set version or additional user-specified parameters. v, 6, 21

pipeline run Also *pipeline execution*. The formal specification of a *pipeline run artifact*. A pipeline run combines a *pipeline configuration* with a versioned data set and any additional parameters based on this data set or other user intentions. By sharing this pipeline run configuration, another user should be able to deploy this pipeline run and reproduce the same results. vi, 6, 22

pipeline run artifact Also *pipeline artifact*. The artifacts produced by the (automated) deployment of a *pipeline run*. This artifact actually runs the computations and produces the results. The framework will track all relevant information and try to bring the execution of the associated components to completion. v, 6, 17, 22

provenance of change Information about the origin of changes. This includes what triggered this change, who is responsible for the change, and what the change entails (i.e. a previous and current version). Within the context of data science experiments this includes how both the code and data used by the experiment evolves over time and how it affects the experiment. 1

reproducibility The ability to recreate the results of an experiment by going through the same steps described by the definition of that experiment. This can imply reproducing the exact results from a predefined experiment, or in the case of stochastic processes results that are sufficiently similar in nature. An example of the latter would be accuracy scores very close to that of the original. 1

version tree A tree structure, were each node is a version of something (data, code, pipeline, etc.). Parents are the predecessors and so earlier versions. Children are the later versions that resulted from another version. 19

workflow A term used to describe a business or scientific flow which consists of a set of processes which is often structured in a *DAG*. A workflow is sometimes used interchangeably with *pipeline*, but there are some differences. These differences are described in detail in Section 2.6.1. In this thesis usage of this term is refrained in order to avoid confusion. 9

Contents

Chapter 1

Introduction

The quality of academic research can be quantified (partially) by the ability to reproduce its findings. Increasingly more experiments are performed through the use of simulations and other computational models. Often these cases are implemented in the form of a software system. In the cases that these systems process large quantities of data, they are often implemented as (distributed) data processing *pipelines*.

1.1 Problem Statement

At its core, computers and their software are based on pure mathematics and therefore, one would expect to be able to reproduce the same results consistently. However, especially in the realm of distributed computing, this proves difficult. Due to numerous factors, repeating experiments at a later stage may yield results deviating from the initial run. Knowing how and what changed in an experiment is key in order to explain those deviations. It also helps in determining whether these alternate results are acceptable. Information about where changes originate from and how they are created is called the *provenance of change*. Tracking this over the lifetime of an experiment helps in understanding how the current version of an experiment and its results came to be.

Factors that are subject to change or otherwise influence the reproducibility include the many different hard- and software environments, but also the rapid development of new technologies. Additionally, results depend on the models of the target phenomenon, system architecture, software design, and the underlying algorithms.

Besides the evolution of the computation environment, the data set, on which the computation is performed, is also subject to change. These changes originate from factors such as additional data gathering, data cleaning, and preprocessing.

Finally, the person reproducing the experiment is often not the one that originally defined it. This adds an additional layer of complexity to the *reproducibility* of (the results of) an experiment. Even when the original author of the experiment has documented the experiment extremely well, there is bound to be some tacit knowledge left undocumented, or a software specific dependency left unspecified. This can result in unwanted differences between two instances of an experiment.

All of the aforementioned factors make obtaining identical results from (approximately) the same experiments more difficult with time. In order to ensure the future reproducibility of those experiments it is essential to understand where, and how, changes were introduced. In other words, in order to keep consistent results it is crucial to track the provenance of change of an experiment.

1.2 Research Goals

In the field of computing science the importance of provenance of change has been noted not only within the context of academic research. Computing science has long been dealing with its different elements, such as proper versioning, roll-back and consistency across environments. To deal with the issues that accompany these elements, systems such as Git[1], Docker[2] and even Java[3] have been developed.

On the one hand, the field of computing science has developed state of the art tools that deal with (elements of) provenance of change in software settings. On the other hand, academic research struggles with similar issues related to change provenance. This research tries to bridge the gap between these two.

1.2.1 Research Questions

This thesis its the result of a joint research and therefore covers a subset of all aspects involved in the problem space. Pointers for results outside the scope of this thesis can be found in the next section. Given the perceived problems in academic research as described in Section 1.1, this thesis focuses on the following question:

How to track the evolution of data in (data science) pipelines in order to enable reproducibility and provenance of the results?

This question can be further divided into a set of sub-questions and topics. The following sub-questions will be addressed throughout this thesis in order to answer the main research question:

1. What are the various elements of change provenance related to data?
2. What tools and platforms exist that support tracking provenance of change in datasets?
3. How can an (adaptive) data science pipeline framework be designed, which is able to respond correctly to the different changes that can occur?
4. Which provenance elements can be tracked prior to the execution of a pipeline?
5. Which elements of provenance, that can be tracked, are specific to a pipeline execution?

1.2.2 Research Contributions

This thesis explores the reproducibility and tracking of provenance of change in data science pipelines, focusing on the tracking of data-related provenance. Another thesis written about this research focuses the aspects of pipeline and code provenance [2]. The combined findings are used to design an architecture for a framework which accommodates tracking the provenance of change in (data science) pipelines. From this architecture, a proof-of-concept implementation is introduced, to demonstrate the provenance tracking and reproducibility features. The validity of the framework is evaluated by re implementing two data science experiments from the domain.

The two theses describe insights from the joint research, and therefore the theses have a large overlap in content. The main contribution of the research is the architecture and

[1] git-scm.com
[2] docker.com
[3] java.com

implementation of the framework, which stem from the same requirements and design. These chapter are therefore common between the two theses. A complete overview of which chapters and sections overlap between the two theses can be found in Appendix A.

1.3 Methodology

This research was kick-started by evidence from the academic world that reproducing results of (computer-based) experiments can be challenging [19, 9]. These challenges were scoped down to reproducibility issues with cases of (distributed) data science pipelines. Distributed computing as well as processing of large quantities of data makes these problems especially apparent.

After this initial phase the standard research process began with a literature study of the available material. This included researching academic references as well as investigating existing tools, their capabilities, and their shortcomings. In an iterative process, these investigations led to a better understanding of the problem space and a well defined list of aspects to take into account. From this a list of requirements could be extrapolated. After concluding that most existing platforms have some shortcomings, a design phase was started. This led to a conceptual framework existing of components and interactions that aid in tracking the provenance of change. Due to time constraints and other resource constraints it was deemed most feasible to implement a custom proof of concept of this framework, rather than building on top of some existing alternatives. This will be covered in more depth in Chapter 2. This proof of concept was then evaluated by implementing two actual use cases within this framework.

Chapter 2

Background and Related Work

This chapter covers a mix of related academic work, background information on relevant topics, and an analysis of existing tools. It gives an overview of the various elements associated with tracking provenance of change, as well as a breakdown of various other challenging aspects induced by data processing, pipelines and distributed computing. It also provides an analysis of tools that currently already provide (part of) the envisioned solution.

2.1 Data processing models

Nowadays, data is collected at a massive scale. It can be gathered from server logs, activity data, sensors and more[3]. All of this data needs to be processed somehow. There are a couple of different models to process large amounts of data. In batch processing, data is collected over a time window, grouped together, and then processed as a whole. As opposed to batch processing, data can also be processed as a stream, where data is processed as it comes in. Both of these processing models have their appropriate use cases and are not mutually exclusive.

The Lambda Architecture is an example of an architecture where batch and stream processing are combined [12]. The stream of data coming in is processed via multiple paths of computation. First, data is processed as a stream to provide quick, slightly less accurate results. Simultaneously the data is stored so that it can be processed in a batch for more accurate, but slower results.
In data science experiments, scientists often work with a fixed data set, and thus a fixed "batch" of data. Even though this hints at the batch processing models, there may be situations where the batch of data is too large to fit in the storage of one machine. The Kappa architecture solves this problem by interpreting a batch of data as a bounded stream [14]. The batch is split into separate elements which are processed individually. This unified approach processes both batch data and streaming data as a stream. This reduces the complexity of data processing infrastructures. The first introduction of the Kappa architecture is based around the Kafka technology where data is stored in a distributed log [13]. Incoming streams of data are processed immediately. Then, when a batch of historic data needs to be processed, the log is "replayed" and processed by the stream processing components again.

2.2 Pipelines

A *pipeline* consists of a set of (independent) data processing steps configured such that these steps process an input data set, and send their output onwards as input for a next processing step. Due to processing steps transforming data from their input set to their output set they are often called transformation steps. This configuration of transformation steps together forms a graph structure through which the input data flows and gets transformed by the transformation steps. Pipelines can both be used to process batches of data and streams of data. However, for processing a batch of data each of the transformation steps has to finish before the next transformation step can start. Pipelines structured to process a stream, transformation steps can start processing as soon as the first pieces of output become available from the preceding transformation step.

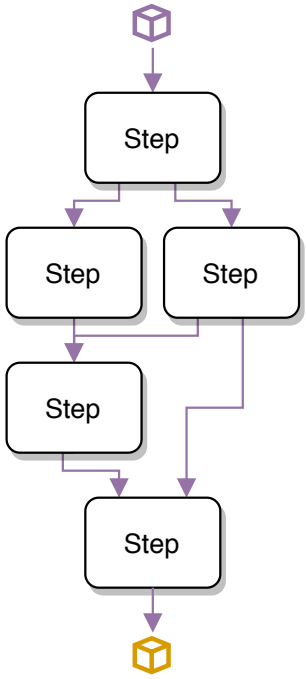


Figure 2.1: An example DAG describing a pipeline showing multiple inputs, outputs and multiple streams

Traditionally, a pipeline referred to a linear set of transformation steps chained in sequence. Nowadays, this definition has expanded to include any configuration of transformation steps structured as a *DAG*. Figure 2.1 shows an example of such a pipeline. This implies that transformation steps may have multiple inputs and outputs. This does not only cover the case where multiple (different) transformation steps post a single type of input to some transformation step. It also includes cases where a transformation step receives multiple input streams in which the data is not of the same shape and somehow needs to combine these streams. This creates a complicated problem of knowing how and when to combine which elements of which of the input streams to create a new output. Similarly, multiple output streams do not all have to send the same (shape of) data to their targets.

Pipelines appear in many forms and implementations. Pipelines can be implemented within one program in the form of modules or functions. To have the benefits of concur-

rent and parallel processing the different transformation steps can be split over multiple cores and threads. Take it one step further and each transformation step can be implemented as its own program distributed across multiple machines. All of the above can match the specifications of a pipeline, however, within this research the focus lies on the latter one. A distributed setting introduces more pitfalls and problems regarding provenance tracking. Assuming that the rate of growth of data sets will not slow down, the associated problems will only become more apparent. This is why this research focuses on distributed data pipelines. Their associated problems are covered in more detail in Section 2.3.

2.2.1 Within this Research

This research makes a distinction between the definition of a pipeline and its execution. A pipeline is defined by the set of transformation steps (code, execution environment, etc), and their configuration. This determines how they are connected, which input links to which output, and any other configurable elements of a pipeline. This pipeline definition is called the *pipeline configuration*. The configuration is defined via a formal structure describing all the necessary information. This could be the entire code that links the elements of the pipeline, but is more likely to consist of configuration files in a structured format such as JSON, or as output from some graphical interface. Those formats would then refer to explicit versions of the actual code of transformation steps. Chapters 3 and 4 cover these specifications in more detail.
The execution of a pipeline, or a *pipeline run*, is defined by the combination of a pipeline definition, a data set to be processed, and any additional parameters that can be set for a pipeline. This can be either pipeline-wide or specifically for a transformation step. Examples of the latter one include parameters of a transformation step which can be set dynamically based on the contents of the data set or the intention of the user. The term *pipeline run* can refer to both the (formal) specification of a *pipeline run artifact*, as well as the artifact itself. The pipeline artifact contains the actual programs executed in order to gain the results from the specified pipeline run.

2.2.2 Subjects of Change

As described in Chapter 1, the ability to reproduce an experiment is closely related to knowing the origin of changes. The focus on pipelines introduces more details that are subject to change besides the code for an experiment, the data set used and maybe the environment that it runs in. These additional subjects are the configuration of the pipeline, run time updates to the configuration of a pipeline, parameters of a pipeline execution, and intermediate data sets that result from the transformation steps. These are covered in more detail in Sections 2.4 and 2.5.

2.3 Distributed Computing

The size of the gathered data sets for experiments increases. Additionally, more complicated transformations, such as the training of a machine learning model, increase the workload of experiments. These larger experiments require more processing power in order to complete within reasonable time. To deal with this increased demand, large experiments are performed by distributing the workload over multiple computation nodes.

2.3.1 Cloud Computing

Large companies such as Amazon, Google, and Microsoft rent out their computing resources to users which do not want to, or can not host their own computing services. Resources are allocated and released dynamically by the cloud provider. Cloud services are attractive to use as they provide high availability guarantees, can be used on-demand, and allow users to only use what they need at that moment (i.e. a pay-per-use model). Compute clouds are able to provide the computational resources required for the increased workload of experiments. These clouds can be either private or rented via cloud providers. Usage of cloud platforms is increasing, making their problems all the more relevant to tackle [23]. Even though cloud computing has its merits, it also has some limitations. One of which originates from the fact that users of cloud providers are often subject to vendor lock-in. Meaning that once a provider is chosen it becomes hard to move away from it [15]. These kinds of limitations will need to be considered in order to design a solution to the problems posed in Section 1.1.

2.3.2 Provenance Tracking in the Distributed Setting

The addition of the distributed setting to academic research introduces many new problems that make the reproduction of results and provenance tracking especially hard. Distributed computing in itself introduces a couple of different problems inherent to its nature. These problems can be translated to complications for provenance tracking. Firstly, data needs to be communicated over a network between the different computational instances. Network failure occurs often and losing even a single piece of data can introduce different results for an experiment. Secondly, compute nodes often have ephemeral instances of the experiment running on them. As soon as these fail, data and progress inside is lost. If this information is not either recovered or tracked accordingly, later repetitions of the experiment will yield different results.

2.3.3 Orchestration

Many things can fail in a distributed setting. This requires systems designed for a distributed system to be fault-tolerant and able to recover. Orchestrators automate various processes related to managing distributed systems. Typically the orchestration software is installed on multiple computation nodes, which then form a pool of resources for the orchestrator to use. Users supply which services need to be active, and the orchestration software schedules the required services on the available nodes. This layer of abstraction allows the users to focus on the functionality of the system they are developing rather than the headaches of distributed deployment.

Kubernetes[1] being perhaps one of the most prominent examples of such a system [6]. Kubernetes is an open-source, widely used orchestration tool first developed by Google. It allows for the deployment and management of services over distributed computation nodes specified by YAML-structured configuration files.

2.4 Provenance of Data

In order to fully reproduce experiments it is critical to know how and when changes occur. These changes include changes to code, data, hardware, and execution environment for example. Each respective thesis (Section 1.2.2), identifies subjects of change related to its research scope. This includes a description of the subjects as well as what would be needed to track their evolution.

[1] https://kubernetes.io/

2.4.1 Data Sets

The provenance of the data sets themselves is also important, as a data set is not necessarily immutable. Similar to source code, data sets can evolve over time, so it is important to keep track of the changes to a data set. Data can be stored in CSV files, or as binary data such as images, or be stored in databases. The various types of data in data sets make a generic solution for version controlling data difficult. The techniques used to version control source code cannot be used to version control files which are not text files structured on newlines or line feeds. Even those files that are, such as CSV files, do not work well with source code versioning systems due to their size and the protocol behind those systems. It is however imperative that changes to data sets are tracked, as different input data obviously leads to different output data.

Data Versioning Tools

DVC stands for Data Version Control[2] and is an open-source platform which can be used to version control data. The user interacts with DVC using a command line interface, with commands that are very similar to Git. The idea of DVC is that the user maintains the references to the data versions used in the version control of the source code. In this way the version of the data set is coupled to the version of the source code. The data itself is stored separately in a back end of the user's choosing. The user is able to reproduce the original data set from the reference stored in the source code version control.
In addition to version control for data, DVC provides a pipeline functionality. This functionality allows users to have script files in their repository to run as steps in a pipeline. The results of each transformation step are then stored in the DVC back end. DVC is more focused on smaller experiments which can be run on a local machine. Since DVC couples the version of the data set with the version of the source code, it becomes more difficult to create a pipeline which consists of multiple transformation steps or to reuse transformation steps for other pipelines. Coupling the transformation steps, data and associated pipelines has inherent limitations. This will make reuse of components across pipelines harder. Furthermore, the pipeline functionality supplied by DVC seems insufficient, since running "plain" scripts without an abstraction such as containerization introduces many more environmental dependencies limiting the degree of reproducibility of such a pipeline.

Git Large File Support[3] (Git-LFS) is an open-source extension for Git. As the name implies it allows the user to use Git to version control large files. Even though Git-LFS has some benefits such as being able to use the same Git workflow that most coding scientists will be familiar with, Git-LFS has some limitations. The maximum file size for Git-LFS is 2GB, which is enough for some use-cases, but not enough for data sets consisting of larger video files for example. In order to support a wider range of experiments, let alone experiments with big data, a more robust solution is needed.

2.4.2 Intermediate Pipeline results

Another form a change provenance in data is the data that results from each transformation step. Instead of saying that a pipeline transforms a data set into an output data set, one can also think of transformation steps doing this on a smaller scale. Tracking how the input data changes throughout the pipeline enables connecting output data elements to their original input element(s) and by which transformation steps they were

[2] dvc.org
[3] git-lfs.github.com

produced. This kind of information is also called *data lineage* and is a key aspect of gaining insight in the origin of the data. Keeping track of the intermediate data also allows for increased performance in the run time when a pipeline is executed again. For instance, if transformation steps of a pipeline change with time, but the previous step(s) remain the same, the intermediate results of an earlier run of the previous step can be fed directly into the newly updated step. This removes redundant re-computation of all the prior results.

2.5 Provenance of Pipelines and Code

The results of an experiment are dependent on the data being used as an input, but also the computation performed on the data. Because of this, the versions of the transformations, and on a higher level, the whole pipeline which performs the experiment also need to tracked. These aspects are described other thesis which describes a different focus of the joint research as described in Section 1.2.2.

2.6 Existing Workflow and Pipeline Tools

Many tools, frameworks and programs provide (parts of) the functionality required to solve the problems of reproducible research and provenance tracking. There is particularly much to find on automated, distributed *workflow* deployment. None of the tools analyzed during this research seem to focus as much on reproducibility and provenance as desired, though many of them do invest in concepts such as data lineage. Some of these tools are investigated here.

2.6.1 Workflows versus Pipelines

Workflows and pipelines are terms that often have different meanings depending on their context. Pipelines have been defined in Section 2.2 as a graph of transformation steps through which data flows and gets transformed in order to achieve some common goal. When researching these topics the term workflow is often used as an alternative to pipeline. Though a workflow seems similar to a pipeline, there are differences. Within this research these terms are explicitly separated. A workflow is a combination of processes which describe some business- or scientific process. Similar to a pipeline, a workflow can be structured as a DAG of processes which interact with each other. Workflows are often higher level processes which can be more complex and might take a longer time to perform. This comes from the fact that processes are not necessarily performed by computers, but may also be some external process. The workflow can be triggered by some external event, or by another workflow. The data that flows between the different workflow processes is often metadata rather than actual data; in the end, the goal of a workflow is to perform some process, not to process data per se. An example of a workflow for a business is how an order for a product has to be handled. A more specific example within data processing is: pull data from a database, extract some information, produce a report, and send an email with the report, repeating the process every week. This is in contrast to the goal of a data science pipeline, which is to process a data set in order to gain new insights. It could very well be that a data processing pipeline is one of the processes in a larger workflow.

2.6.2 Pachyderm

Pachyderm[4] is an open-source data science platform with (enterprise) premium dashboard aimed at scalable, distributed and repeatable data science. Its goals seem very much in line with this research except for a few points. On the one hand it boasts open-source, but restricts necessary features such as a dashboard interface behind a paywall. It is also very resource intensive for local development. Running Pachyderm means first installing Kubernetes locally, followed by all of their abstractions. This makes even simple experiments slow, and more focused on setup rather than experiment development. Where they do provide a form of data lineage, it is not as extensive as this research argues it should be. You can track how data flows through a pipeline, however upon deletion of certain elements, all its information is lost. These are cases where, in order to be reproducible, at least preservation of the fact that this deletion happened is necessary. Ideally this also includes what information was deleted.

2.6.3 Apache Airflow

Apache has a fully open-source workflow manager called Apache Airflow[5] that allows workflows to be defined and configured in Python. Where Airflow shines in scheduling, automated deployment, extensibility and user friendliness, it does not provide all the features necessary for provenance tracking framework. Airflow is not made for data processing and passing data around. It (ideally) only passes around metadata and status messages. When actual data processing needs to happen, the average Airflow workflow launches a Spark[6] job which does the actual processing. Although Airflow supports a very experimental a form of data lineage, it can only track lineage on the level of the messages send between the nodes in the workflow. As these messages mostly consist of metadata and not of actual data itself, the tracking of data lineage is limited. In order to further stimulate reproducibility of experiments, more fine-grained level of data lineage is necessary.

2.6.4 Luigi

Luigi[7] is a tool very similar to Airflow. It was first developed by Spotify, is less mature and adapted by the community than Airflow, and runs into the same problems as described for Airflow in terms of processing.

2.6.5 Kubeflow (Pipelines)

Kubeflow[8] is a distributed, repeatable machine learning (ML) workflow deployment tool for Kubernetes [4]. It is one of the tools that seems to follow the ideas posed by this research, including the processing of data. A subsystem of Kubeflow called Kubeflow Pipelines can be used as a standalone and is more focused on the ML workflows and their deployment, whereas Kubeflow is a bit more generic. There are however three main concerns with Kubeflow Pipelines (in its current stage):

1. Much of Kubeflow (Pipelines) is in its alpha or beta stages. This means that much of it is subject to change and more error prone than desired.

[4] www.pachyderm.com
[5] airflow.apache.org
[6] spark.apache.org
[7] github.com/spotify/luigi
[8] www.kubeflow.org

2. Kubeflow is very resource intensive. Getting a local version running for Kubeflow proved troublesome due to a number of factors. Initially due to installation problems, which could be related to the alpha/beta status. Secondly, once set up, Kubeflow spun up over 60 pods on the local machine. This consumed all of the resources and made it slow down before it could even finish initializing.

3. Kubeflow has a focus on ML, rather than generic data science. It reasons from an ML perspective, having terms like training set and test set tightly integrated in the definition of Kubeflow workflows. Many data science experiments would not require structures like these.

In the future this Kubeflow may become more and more feasible as it matures, provided that the heavy weight local runs can be scaled down appropriately.

2.6.6 Flyte

Flyte[9] is an open-source cloud native machine learning and data processing platform developed by Lyft. It is similar to Kubeflow in its functionality, however it provides a more light weight approach (which is still quite resource intensive), has a lesser focus on ML, and seems to be more mature than Kubeflow. Flyte did however introduce complications with installing locally, and resource intensive non-distributed jobs. Nevertheless, The design of Flyte can be used as an inspiration for this research. Flyte allows users to create so-called "tasks" which are fundamental building blocks of their pipelines. These tasks can be functions which perform some computation on input data similar to what is done in functional programming when a function is mapped on an array or a stream. These tasks, which the user has to provide, can only be written in the Python language, using the provided SDK. Unfortunately, Flyte lacks the ability to keep track of lineage information in the pipelines it is able to run.

2.6.7 Remarks on Existing Tools

The existing tools and frameworks have their strengths, but unfortunately these do not completely align with the vision for this research, as provenance tracking is its focus. None of the aforementioned tools have this specific purpose. Additionally, all these platforms are very resource intensive, especially when run on a local development machine. Smaller experiments still exist today and a true solution should not differentiate between small and large experiments, but scale its required resources according to the experiment in question.

A project whose vision most aligns with that of this research, focusing on provenance tracking is *e-Science Central* [10]. This project requires the user to provide code for experiments in Java or Matlab. This is quite restrictive, as many data science experiments nowadays are also performed in languages like Python, Julia or R. Unfortunately, during more recent years this project seems to have been abandoned. The website has been taken down and its content is no longer referenced in academic works. This, once again, leaves a need for a framework which uses modern tools and practices without restricting users in what experiments they can perform. This research attempts to fill this need by designing and providing a proof-of-concept for this framework.

[9] flyte.org

Chapter 3

Architecture

This chapter describes a high level architecture of a conceptual framework designed to be capable of tracking provenance of change such that experiments remain reproducible. This chapter refrains from delving into any implementation details (which are covered in Chapter 4). Approaching the problem in this way ensures that the fundamentals of the framework and its components become apparent. An additional advantage is that when current tools become obsolete, this framework can be re-implemented using the newest state of the art tools, whilst relying on the same general architecture.

Additionally, note that this document is not meant to provide a full software architecture description as often done in software engineering (SE). The focus of this research is to provide an architecture for a framework, not a software product specification. This chapter will use terms that come from the SE domain in a less strict manner. Examples of this are functional requirements that are more general and at a higher level than expected from an architecture description in SE, as well as the absence of a 4+1-model.

3.1 Concepts and Ideas

3.1.1 Fragment Abstraction

In order to facilitate as many different kinds of experiments as possible, proper abstractions are necessary. This research defines a layer of abstraction over the data of an experiment, such that no assumptions are made about the underlying data. To deal with all the different shapes and sizes that data may be available in, this research states that pipelines operate on *fragment streams*, rather than on (batches of) data directly. A fragment is used to denote a subset of the data, not limited by files and their types. For example, a fragment can refer to a single line from a CSV, a single image file, or a set of files supposed to be processed as a single batch for example. As soon as any definitions change, the content of a fragment could change as well. This abstraction does not necessarily mean that each fragment has to be independently processable. Fragmenting can also be used in order to optimize for download size. One request and associated download for 1 row of CSV data uses much more bandwidth than necessary and can be countered by putting multiple rows/files into one fragment. Since fragments abstract away from the actual data, it becomes the responsibility of the users to open up a fragment and process its contents.

This abstraction changes the definition of a transformation step transforming one data set into another, to a transformation step transforming a set of fragments. By transitivity, this also changes the definition of a pipeline to process streams of fragments rather

than streams of data. Processing a large data set as a stream of data elements allows for changes to the pipeline whilst the pipeline is running. This fragment abstraction does not break this feature. The stream of fragments, just like a stream of data, can be redirected to a newer version of the transformation step for example.

It is important to note that there is no restriction or necessary correlation between the shapes of the input and output streams. This abstraction also captures any cases where data is supposed to be processed in batches, because any fragment can contain an arbitrary amount of data. In a special case this would mean that an entire data set can be treated as a single fragment. A similar technique is employed by Apache Flink[1] for processing batch data [22].

3.1.2 Automating Features for Accessibility

To support many disciplines, the system needs to be easy to use. In order to take away as many pitfalls as possible, automation of processes is necessary. Such processes can be identified by analyzing the common parts between data science experiments. Simultaneously, the goal is to support any and all experiments that a scientist would like to run. Automating some processes would require assumptions about what a scientist is trying to achieve with his or her experiment. A few processes that can be automated include:

- Feeding the output of one transformation step as input to another. In the case of a distributed pipeline this means moving data across the network from one process to another. As explained in Section 2.3, it can be difficult to do so reliably. Taking this issue away for a non-computer scientist will increase the accessibility.
- Automatically rerunning the pipeline when either the data set or a version of a transformation step has been updated.
- Dynamically updating transformation step(s) of a running pipeline after a transformation step has been updated.

There are also a couple of processes which cannot be automated for the user:

- Defining the actual computations of transformation steps, only the scientist knows how his experiment tries to achieve its goal. However they can be reused.
- Tracking which fragments are used to produce which other fragments. This is due to the black box treatment of transformation steps. Many inputs could lead to one output, one input could lead to many outputs, and any other combination is an option. This is dependent on what the user is trying to achieve, which should be boundless.
- Translating the data of a specific data set version into a fragment stream as expected by a transformation step, such that it can be processed. Storing data in a versioned manner can be done, but it cannot cover all possible input formats that a transformation step may require.

3.2 Requirements

The following three key quality attributes are deemed most important: accessibility, portability and performance. These attributes are induced by the broad target that

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

the framework focuses on. Ideally any scientist using a piece of software for his or her research should be able to integrate with an implementation of the framework. Most scientists do not have a computing science background and therefore, for a framework like this to be adopted, usage should be as simple as possible. Any scientist able to write code for his or her research should be able to use this framework. Hence, the accessibility attribute. Additionally, scientists will use a very broad spectrum of environments. Running the same experiment years later for validation should yield the same results; this requires a high portability of the system with each current and future platform to arise as a target. Finally, the performance attribute comes from the fact that no scientist will use the framework if it takes exponentially more time as compared to running it without. Additionally, as described by Section 2.6, distributed platforms that currently supply some form of these features are often very resource intensive. This makes a performant, lightweight alternative desirable.

Using the quality attributes and insights gained from Section 3.1 and Chapter 2, a list of requirements can be determined. These requirements together describe a system that supports provenance tracking and reproducibility of data science pipelines. The identified requirements are categorized into 4 main functional requirements. These are data provenance, pipeline provenance, pipeline run time management, and user interaction. For each of these, a general description is given together with a list of specific requirements belonging to this main requirement. Each of the requirements is prepended with one of 4 labels referring to either Core, or to one of the quality attributes (denoted Acc., Perf, Port). The former means that this is a core functionality of the system that is not necessarily in support of any specific quality attribute. Following the set of functional requirements is a set of non-functional requirements grouped into the three key quality attributes. Some are annotated with another attribute indicating they are multi-purpose.

FR 1 Data Provenance

As stated in Section 2.4, it is important to keep track of the version of your data set in order to enable reproducibility. As is tracking how data is produced as the result of running a pipeline. Therefore, one of the requirements is that the system supports a data versioning element to use for versioning data sets as well as relating transformed results to those versions.

1. [**Core**] Define a storage format within which data sets are stored.
2. [**Core**] Track all of the different available versions of the data set.
3. [**Core**] Commit additions, removals and modifications to the data set in order to create new versions.
4. [**Core**] Support roll-backs to previously committed versions of a data set.
5. [**Core**] Store (intermediate) results of a pipeline execution related to a (versioned) data set.
6. [**Core**] Maintain lineage information about the data during the execution of a pipeline.
7. [**Acc.**] Be agnostic towards where versioned data sets are stored as to ensure vendor locked users can still use the framework (e.g. AWS-S3, local storage, private storage).

8. [**Acc.**] Store data sets remotely such that multiple users are able to work on the same data set simultaneously.

FR 2 Pipeline Provenance

This research focuses on provenance tracking of pipelines. As described in Section 2.5, in order to reproduce results it is critical to understand how a current version came to be, along with all changes that make up that version. Besides data provenance, this also requires that the system is able to track all subjects of change related to pipelines and their code. This includes transformation steps, pipeline configurations, and their executions.

1. [**Core**] Track the versions of all different aspects of a pipeline configuration. This includes the transformation steps, the version of the framework, and the DAG structure of the pipeline.
2. [**Core**] Track the versions of all different aspects of a pipeline run. This includes the version of the data set used, the pipeline configuration, and any additional user set parameters.
3. [**Acc.**] Accommodate the definition of pipelines consisting of N transformation steps defined as a DAG through a simple format (e.g. JSON or YAML).
4. [**Acc.**] Allow the passing of (parameter) configurations to transformation steps via the pipeline definition, in order to tweak the exact behaviour of a transformation step.
5. [**Perf**] Deploy updates to running pipelines dynamically and track these dynamic update events.

FR 3 Pipeline Runtime Management

It is important that a defined pipeline can be executed by the system in order to allow the execution of data science experiments. Therefore, it is necessary that there is a component which is able to execute the user-defined pipelines.

1. [**Core**] Deploy pipeline runs based on their definitions over available resources
2. [**Acc.**] Deliver data to relevant endpoints without additional user specification besides the pipeline.
3. [**Acc.**] Be agnostic towards the experiment performed in terms of programming language, shape of data, and execution environment.
4. [**Perf**] Scale individual transformation steps when indicated by the user.
5. [**Perf**] Given intermediate results and lineage information, allow restarts of a pipeline from an intermediate point, without having to re-execute all upstream nodes.
6. [**Port**] Function in both a distributed and single (computation) node setup.

FR 4 User Interaction

The user defining a data set or defining a pipeline should be able to interact with the system in a user-friendly manner.

1. [**Acc.**] Present one or more user interfaces which allows the user to interact with the various elements of the system.
2. [**Acc.**] Both results and lineage information of an executed pipeline should be both presented to, and retrievable by the user.
3. [**Acc.**] Notify the users about the progress of a pipeline run.

NFR 1 Non-Functional Accessibility Requirements

This requirement describes non-functional requirements that further support the accessibility quality attribute.

1. Try to explain from where inconsistencies originate in the case of inconsistent results that arise from non-user-defined behaviour such as crashes or failed requests that cannot be recovered from.
2. Redefining existing experiments within the scope of the system should require minimal effort for a user
3. Users with enough skill to program their experiments should be able to use this system

NFR 2 Non-Functional Performance Requirements

This requirement describes non-functional requirements that further support the performance quality attribute.

1. Experiments executed within the system should at most take 1.5 times the run time of the same experiment executed outside the context of the system.
2. Handle data sets larger than the memory capacity of a single node.
3. The system must be light weight such that it is able to run reliably on a single machine.

NFR 3 Non-Functional Portability Requirements

This requirement describes non-functional requirements that further support the portability quality attribute.

1. Interaction with versioned data sets do not require a user to download the entire data set. A client machine with insufficient disk space to fit the whole data set should still be able to use the system.

2. [**Acc.**] Data set development and manipulation works independently from pipeline definitions and their deployments such that users working on one of these do not need knowledge nor dependencies of the other.

3.3 Architecture Overview

This section describes the set of architectural components and their interactions on a high level. The components are divided into two categories: the infrastructure within which pipelines are created, deployed and managed (Section 3.3.1), and the components associated with such a *pipeline run artifact* (Section 3.3.2). Each of the components in these sections will be covered in more detail in Section 3.4. This section is only meant to give a general overview of the various components and their interactions.

3.3.1 Pipeline Infrastructure

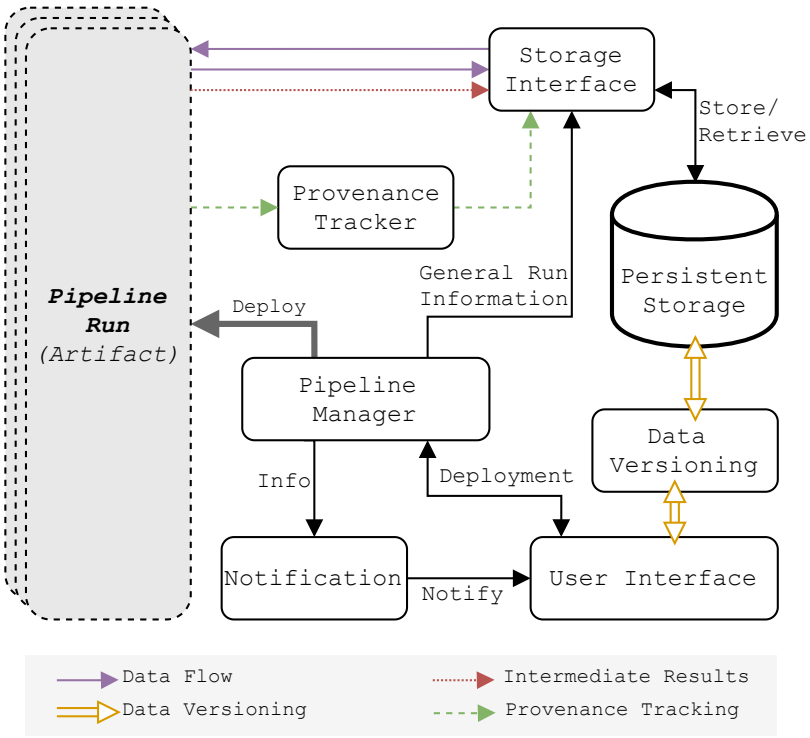


Figure 3.1: Component overview of the long-lived components

Figure 3.1 shows an overview of the longer lived components, together with a pipeline execution artifact. The user interface is the entry-point of the framework. Users will need to interact with the system and so, in order to make the system accessible to all users, a clear and well-presented interface is needed. In the diagram above, the user interface can be treated as the user as it is the only way to interact with the system. From here the pipelines can be deployed by submitting pipeline run specifications to the pipeline manager. Besides pipeline management and deployment, the user interface also grants access to the data versioning features of the framework. The data versioning

component is responsible for handling all requests regarding creation, updating and versioning of data sets. The pipeline manager component is responsible for deploying, tracking and further managing pipeline artifacts that a user wants to execute. It is also responsible for keeping the user updated by passing on information to a notifications component, which will then inform the user via the interface. Furthermore, in order to allow for decoupled data set development and pipeline executions, there is a dedicated storage interface for other components in the system such as the pipeline manager and the transformation steps. Finally, there is a provenance tracker component which is responsible for tracking provenance data regarding a pipeline execution. This includes, but is not limited to, information such as data lineage on the fragment level.

3.3.2 Pipeline Artifacts

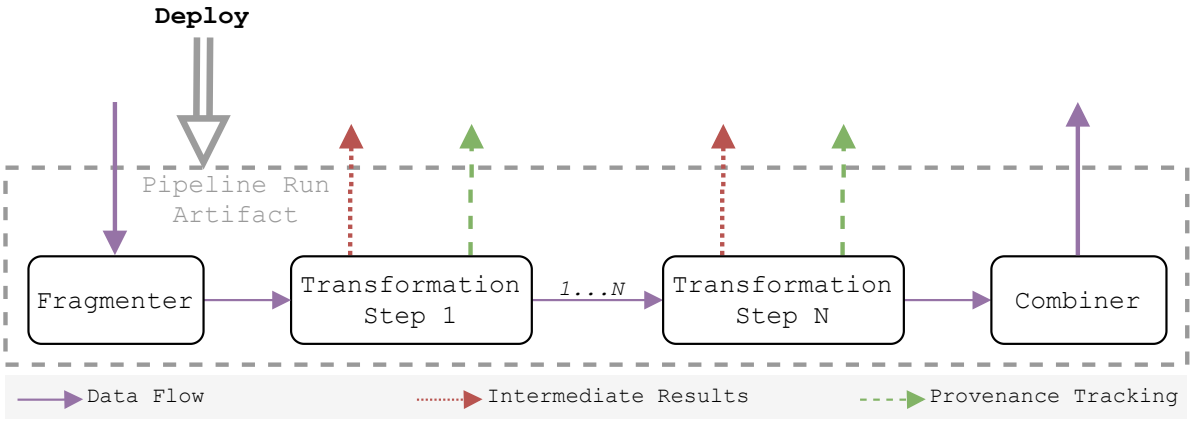


Figure 3.2: Component overview of the run time components

After a user has submitted a pipeline run configuration to the pipeline manager, a pipeline artifact is deployed. This artifact consists of three different types of components. These types are fragmenters, transformation steps, and combiners. An example pipeline is shown in Figure 3.2. The components of the artifact are created when a pipeline artifact is deployed, and removed after it has completed.
The fragmenter acts as the source of data within a pipeline run. With that, it is also the root of its associated DAG. This component is responsible for transforming the files associated with a data set, into a stream of fragments. Since the goal of a pipeline is defined by the user, it is important that the framework makes no assumptions about a data set. Therefore, users can define how a fragmenter generates fragments from a data set.
The transformation steps are the user defined operations that process fragments from the fragment stream. Furthermore, each transformation step can save its intermediate results. They also send provenance information about the fragments they process in order to track their progress.
The combiner acts as the final step of the pipeline and is responsible for transforming a fragment stream back into just a data set and storing it along the original dataset. This means that a combiner typically acts as the sink of the DAG of a pipeline. The combiner is created automatically without specifics provided by the user. This is possible because both the format of fragments and storage format of datasets are defined by the framework. So, in order to help users, this process can be automated.

3.4 Architectural Components

3.4.1 Persistent Storage

In order to enable reproducibility over a larger time span, it is necessary that data sets, their versions, the pipeline versions etc. are stored persistently. The persistent storage component is a storage service where the data required by the framework can be stored. The goal of this component is to allow users to re-run pipelines even years later, because the data is still stored in the same manner.
In order to support as many users as possible, the framework needs to support many different storage backends. Users may already be accustomed or vendor locked in to one provider and so, supporting as many as possible furthers the accessibility of the framework. The storage backend can be provided by either a cloud storage provider, such as Amazon S3[2] or Google Cloud Storage[3], or by some other persistent storage connected to the framework. For testing and development purposes a storage medium of a local machine can also be used.

It is important to note that the only way through which stored data should be accessible is via either the storage interface component or through the data versioning component. This is due to the fact that these two dictate the format and structure of the stored data. This allows for a clean decoupling of the interface and the storage. The data might not be needed for years, and can sit in archival storage without needing any CPU resources to manage it. This saves resources and, especially for cloud instances, money. The moment this data set is needed again, an instance of the storage interface or the data versioning component can be started. This instance will mount the storage backend, which makes the data available again.

3.4.2 Storage Interface

Each of the persistent storage providers might have different interfaces. This component provides a common, easily extendable interface over these persistent storage providers. This enables the users of the system to use any storage provider they want, whilst using the framework. The storage interface is used to grant access to the stored data in the format of the framework to the other components in the framework. These other components include the pipeline artifacts, pipeline manager and the provenance tracker. Each time these components need to retrieve or store data, they do so through this storage interface.
The storage access of the data versioning component (Section 3.4.3) is separate from the rest of the framework. This decoupling allows scientist to work with a data set without having to use the rest of the framework and its dependencies.

3.4.3 Data Versioning

The data versioning component is responsible for managing the versioning of data sets. A data set can have many different versions, each of these are related to their previous versions. Newer versions are usually similar to an older one, but with some additions or removals. This results in a tree like structure of versions, called a *version tree*. It is important that all versions and the version tree itself are kept valid, since actions such as rolling back to a previous version of a data set, or changing to another branch in the version tree needs to be possible. The data versioning component performs this validation and ensures these rollbacks and version changes are possible.

[2] https://aws.amazon.com/s3/
[3] https://cloud.google.com/storage

After a new version of a data set has been submitted, the actual files of the new version are stored in the persistent storage component. Other components in the framework are then able to access the files of the different data set versions by requesting the files from the storage interface component.

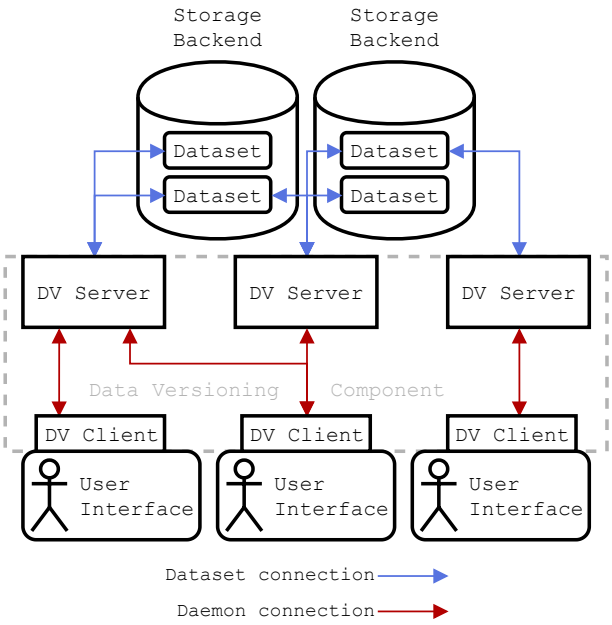


Figure 3.3: Component architecture of the data versioning component

The data versioning component functions similar to version control software such as Git[4] in the following ways: there is a remote version, which is the ground truth. It has a tree of versions and branches, each of which is immutable. Locally, users can pull the latest version, stage any changes to the data set and then commit the results back to the system. This gives the data versioning 3 layers as portrayed in Figure 3.3. One difference with version control software for source code is that it is not necessary to download the whole data set in order to make changes to a data set. This is necessary because data sets might be too large in size to fit on a single machine. The storage is handled by the persistent storage component. The local functionality is performed by the *DV client*, which can be exposed through the user interface component. Validation, committing, and pulling commits is handled by a subcomponent called the *Data versioning server*. This server is a separate program that can run independently and the DV clients connect to them. It can run on the same machine and be presented as one. This extra layer of indirection however, allows for multiple users to connect to the same data versioning server. This would enable special access control restrictions on data sets or storage backends at this server level, which is useful in cases where data is classified. The fact that different users can access data both via their own instances of a data versioning server, or by connecting to the same server also ensures that scientists working remotely on a joined project can more easily share (access to) data sets. This multiple users and multiple providers pattern is also shown in Figure 3.3.

[4]https://git-scm.com/

3.4.4 Pipeline Manager

The pipeline manager is the component responsible for managing the different aspects of a pipeline run. The user provides a configuration for a pipeline run, which includes the versions of the transformation steps, the version of the data set and other configurable settings of a pipeline in a structured format. The pipeline manager is then able to deploy the different steps of the pipeline and makes sure that the correct processes are scheduled on the available processing nodes. It keeps track of the status of the pipeline and ensures the individual steps of the pipeline complete successfully. If something goes wrong in the execution of the pipeline, the manager is responsible for either restarting the failed pipeline run, or reporting an error to other components in the framework. The pipeline manager also makes sure that the resources used by the pipeline artifact are freed after finishing. This component would also be the one to orchestrate any changes to the pipelines, such as applying dynamic updates and ensuring these would result in a new valid state of the pipeline.

3.4.5 Notification

Whenever a pipeline starts, the pipeline manager keeps track of the status of this pipeline. The notification component is responsible for notifying the user about the status of the pipelines and of any additional notifications that may prove necessary. The user can be notified when a specific transformation step has finished, or when the whole pipeline has finished running. The user can also be notified if there is something wrong with the framework. For instance, if there is an error in the source code of a transformation step, or if the persistent storage is not accessible. Depending on the implementation of the user interface (e.g. CLI, web-interface, etc), these notifications can then be visualized and displayed accordingly. The notification component is introduced as a filter and aggregator such that, for larger experiments, the user is not flooded with information. It is meant to present all this information in a concise way.

3.4.6 Provenance Tracker

The framework should be able to track the origins of the results of a pipeline. As stated in Section 3.1.1, the data sets are processed as a stream of fragments. Because of this it is possible to keep track of the lineage of each of the fragments. Each transformation step keeps track of which input fragments are used to produce which output fragments. The exact details cannot be automated, as only the user knows how many fragments are used to create a new one. So the framework should present handles for this. The information is sent to the provenance tracker component, which collects all of the fragment lineage information of the pipeline run produced by the artifact. This information includes which fragments are processed by which transformation steps, which fragments are descendants of other fragments, and so, how data evolves throughout a pipeline. After a pipeline has finished running, the lineage data is redirected to the storage interface to be properly and persistently stored. This information differs from the notifications send to the user, as this information is the raw provenance data that the notification component at most aggregates over. Thanks to this component, an entire lineage path is available for each fragment.

3.4.7 User Interface

The user interface is the main component through which the user can interact with the framework. The user interface allows users to interact with the data versioning component. Through this interface users can also submit a *pipeline configuration* together

with a data set version to form a *pipeline run*. From here the "deployment-request" is submitted to the pipeline manager. The user interface also shows the user whether the submitted request is valid, presents the user with information about its progress, and receives other updates from the notification component. Important to note is that this user interface could consist of multiple subcomponents. This component means to envelop all user interface parts of the framework. For example, if in the implementation the data versioning would be accessed via a CLI and the pipeline manager via a web-interface, these two would still be covered under this component. This component is responsible for handling user interaction, how it does that, in which format, and in how many parts this interface is presented, is an implementation detail.

3.4.8 Ephemeral Components

The ephemeral components, that make up a *pipeline run artifact*, are created by the pipeline manager as the user submits a pipeline run configuration.

Transformation Steps

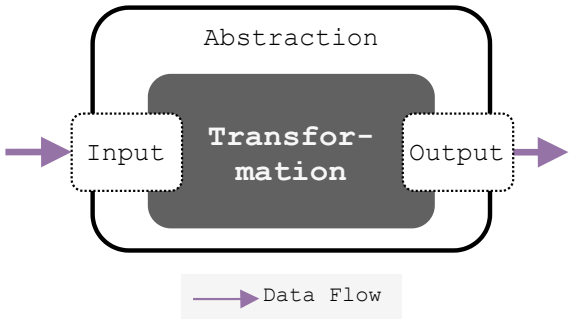


Figure 3.4: Abstract view of a transformation step

The user provides the code for the actual transformations. It is important that the framework is agnostic towards as many aspects of the experiment, so the choice of programming language and other dependencies should not be limited. This implies that some kind of abstraction is needed to refrain from making assumptions about such dependencies. A concrete example of such an abstraction is given in Chapter 4 using modern pre-existing tools. In order to make it as accessible as possible, the framework should take care of getting input data to the transformation step and moving the output data out. Therefore, some kind of interface is needed to connect the framework to the actual computations of the user. The abstraction layer, interfaces and the actual transformation are all depicted in Figure 3.4.

Fragmenter

Data sets are not stored as a fragment stream, as the data set may be useful in different situations or for different experiments. However, within this framework, pipelines operate on streams of fragments. From this a need arises for some component that transforms data from a version of a data set into a fragment stream to be processed by the transformation steps of the pipeline. The fragmenter is this component. This makes the structure of a fragmenter a special case of a transformation step. It processes a single element fragment stream, i.e. the entire data set, and produces N fragments. These fragments are the actual fragment stream of this pipeline. Due to the size of data

nowadays, no machine could reasonably process entire (large) data sets as a single fragment. This is why the fragmenter is still identified as a special component, because it needs to deal with the entire data set at once, irregardless of its size. Ways to deal with this can be to fragment based on metadata about the data set (files, types, etc.), then downloading and attaching the data associated with only that fragment before passing it on. The actual implementation is not covered here, but tackled in Chapter 4. Nevertheless, this problem should be considered and any implementation should carefully design an elegant way to deal with this.

Combiner

As mentioned in Section 3.3.2, the combiner acts as sink of a pipeline execution. It is introduced for two reasons. The first being the fact that a user does not have to deal with storing its output data back into the persistent storage. Both the fragment abstraction and the storage interface are defined by the framework, and so this can be taken care of for the user. Secondly, it functions as a component that can produce "new" data sets from the output of a pipeline. It can be useful for example to store the output of one pipeline, not only as a transformed version of the original data set, but to store it as an entirely new data set.

Chapter 4

Implementation

In this section a proof-of-concept implementation is described. Note that the architecture itself can be implemented it multiple ways. This section describes one possibility using the tools currently available. Additionally, the time allocated for this research did not allow for the full-fledged implementation of a framework of this size. This causes the implementation that is described here to focus on core elements that demonstrate the capabilities and limitations of the designed architecture. This includes the ability to track provenance and to reproduce results, as well as pitfalls and complications discovered during the implementation phase. The internal workings of the implemented services is discussed conceptually, but the fine-grained details of the internal workings of the services is left to the documentation found in the according code repositories.

The proof of concept framework introduced in this section is called *Iterum*. Iterum is open source[1], licensed under the open MIT license. It focuses on two systems which together provide most of the functionalities described in Section 3.2. The first system consists of a data versioning platform, and the second is a pipeline deployment framework. The data versioning system can be used without the user having to install the whole pipeline deployment framework, whereas the pipeline deployment framework requires access to a version controlled data set created by the data versioning system. The versioning of source code has long been handled by version control software such as Git, which has proven to be an invaluable asset in the software development cycle. Therefore the focus of this implementation lies not in version controlling source code and configuration files (Dockerfiles, Kubernetes configurations, etc.), but rather on defining the challenges faced by a provenance tracking framework in terms of such proven tools.

4.1 Implementation Overview

In Section 3.3 an overview of the different architectural components is given. In this section, the mapping of these components to implemented software artifacts is described.

4.1.1 General implementation philosophy

The software artifacts are mostly implemented as microservices. This fits the paradigm of distributed computing, whilst remaining useable in a single node setup. The software artifacts expose their functionality over a web interface, which allows them to interact with each other using predefined interface contracts.

[1] github.com/iterum-provenance

Kubernetes

The framework makes extensive use of Kubernetes in order to manage (distributed) cluster resources. Kubernetes provides a layer of abstraction on top of actual infrastructure resources, which automates the deployment and the management of a set of distributed nodes. This massively reduces the cognitive load on developers. The following abstractions provided by Kubernetes affect the design of the software artifacts:
A *Pod*[2] is the smallest logical unit of computing that Kubernetes can schedule. A pod consists of a set of (Docker) containers which function in conjunction with each other. This means that the containers in a pod are always scheduled on the same physical node in a cluster. Containers in a pod are able to share storage volumes and networks, which is a concept that is used when scheduling pipeline run artifacts.
A *Deployment*[3], is a long-lived workload of pod(s). A deployment usually remains running until the user removes the deployment. Kubernetes tries to maintain the desired state that is described by the deployment. This is done by restarting pods which have crashed for example.
A *Job*[4] is a short-lived variant of a deployment. The pods in the job run until the pod exits either successfully or with an error. After the job has finished running, the job is considered completed and the resources used are cleaned up. Iterum uses more concepts provided by Kubernetes, but these are less relevant in the discussion of the implementation.

Fragment Implementation

Part of the asynchronicity of Iterum is achieved by using a message queueing service to communicate between the different steps of a pipeline artifact. Section 3.1.1 explains that the concept of a fragment is used to process a data set as a stream. However, the data in a fragment might be large and cannot be enveloped by message queue appropriate messages. In order to accompany this, the fragments are split into *fragment descriptions* and fragment data. These fragment descriptions contain references to the data associated with that fragment. In this way, the fragment descriptions are small and can be passed around via the message queue, but the actual data is stored in a distributed storage and can be retrieved when needed.

Language Features

The performance quality attribute is one of the key drivers of the design of the framework. The implementation should therefore carefully consider decisions that may impact this. Picking a programming language is an example of such a decision. Personal preference should arguably not influence the choice of programming language, but rather how well the qualities of a language match the need of the project. This framework extensively uses both the Rust and Go programming languages. Language features such as Goroutines and so-called channels from the Go language are used in order to keep communication asynchronous and workloads concurrent. The borrow-checker from Rust is used in order to keep data access consistent. Both languages are relatively young and low-level, enabling high performance, whilst offering twenty-first century language features, such as proper package management and tooling.

[2] kubernetes.io/docs/concepts/workloads/pods/pod
[3] kubernetes.io/docs/concepts/workloads/controllers/deployment
[4] kubernetes.io/docs/concepts/workloads/controllers/job

4.1.2 Mapping of architectural components to software artifacts

Some of the architectural components are combined into single software artifacts. This is due to time-constraints and the evolution of the architecture throughout the project. This does not have large effects on the functioning of the framework, as the communication pathways between the combined components are similar to the ones described by the original architecture. For example, the *Provenance tracker* component is combined with the *Pipeline run manager* to form the larger "Manager" component. Both of these components communicate with the storage backend, and are therefore combined. Figure 4.1 shows an overview of how the different architectural components map to the software artifacts.

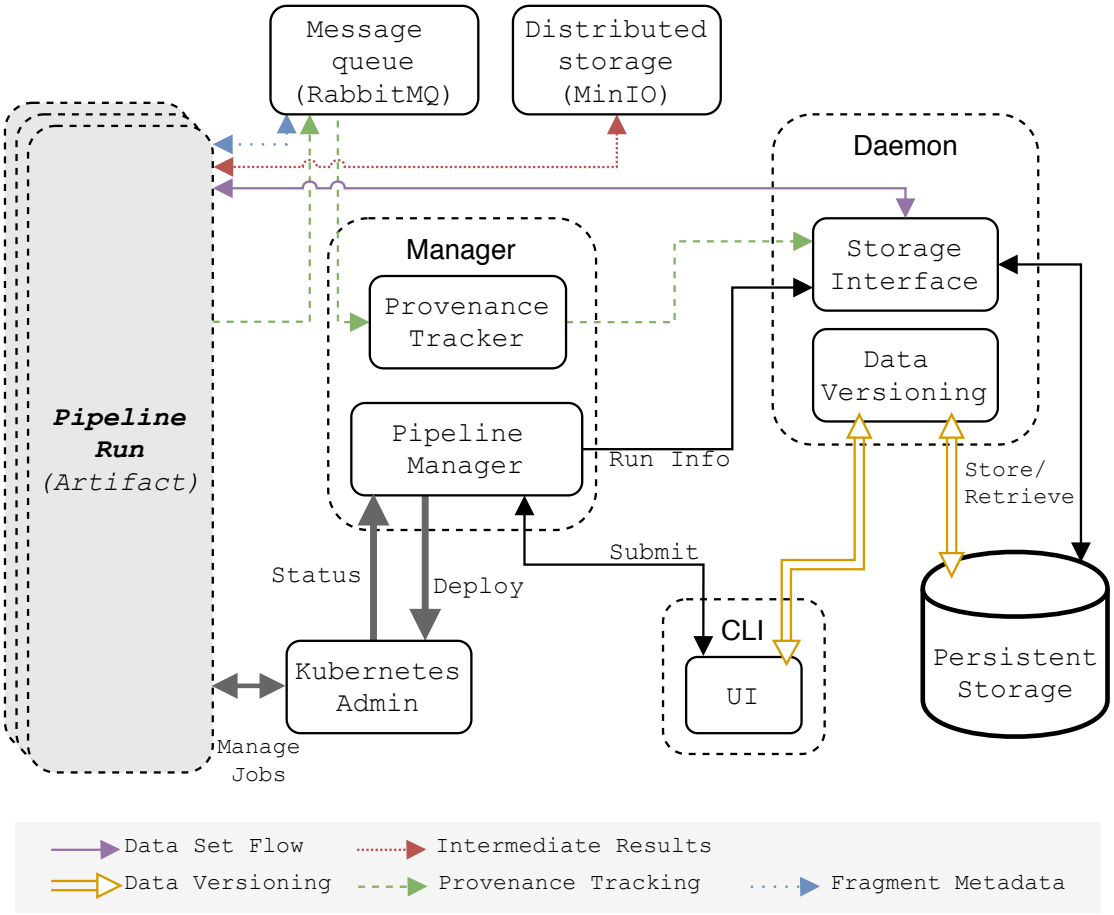


Figure 4.1: An overview of the architectural components and how they relate to the implemented software artifacts

The dashed boxes enveloping one or more architectural components imply a single software artifact fulfilling the function of those architectural components. Furthermore, the notification component has been omitted due to time constraints. This component does not directly contribute to the provenance tracking capabilities, but more to the accessibility. The choice was made to spend more time on the other components, since the notification information can be retrieved instead of delivered. Finally, three additional software artifacts are introduced: a message queueing service (RabbitMQ[5]), distributed

[5] www.rabbitmq.com

storage (MinIO[6]), and the Kubernetes Admin which is part of Kubernetes.

4.2 Component Analysis

In this section, the software artifacts of Iterum are discussed and it is described how they implement the features of the architectural components.

4.2.1 Infrastructural components

Manager

The manager combines the functionality of the architectural components of the *Pipeline manager* and the *Provenance tracker* into one component. The manager communicates with the Kubernetes API whenever a pipeline job needs to be scheduled, hence the Kubernetes Admin.

A pipeline run artifact consists of $2 + N$ subcomponents: a fragmenter, a combiner and N transformation steps; where N depends on the pipeline configuration. Each of these subcomponents are packaged in a pod, and then scheduled as Kubernetes jobs. This means that for each of the pipeline runs, the manager has to schedule $2 + N$ jobs. Scheduling each of the subcomponents in a separate job, instead of one large job containing all of the subsystems, allows for fine-grained control over how the subcomponents have to be scheduled. For instance, specifying the amount of pods per job, and therefore per subcomponent, rather than pipeline-wide. This can then be used to parallelize part of the workload where possible.

The transformation steps of a deployed pipeline produce lineage information for each processed fragment. This lineage information is published on a lineage information queue on the message queuing service. The provenance tracking component consumes this queue, and ensures that the provenance information is properly stored on the storage back end, together with the rest of the information regarding the pipeline.

Daemon

The daemon[7] combines the architectural components of the *Storage interface* and the *Data versioning server*. The users are able to interact with the data versioning daemon via the user interface. Here they specify how new versions of a data set have to be added to the data set. The data versioning component checks and manages new versions of a data set. Valid updates to a data set are then acknowledged and stored in the persistent storage associated with this data set. How the versioning of data is implemented is further explained in the code documentation[8]. Since the storage interface only provides access to the data structured by the data versioning server, these group together naturally into one software artifact.

For this proof of concept (PoC), only a local storage back end is supported. However, interfaces are provided in the source code; if implemented, these will allow the usage of other storage providers such as Amazon S3 and Google Cloud Storage. These extra features would be necessary when using Iterum only for its data versioning features, without running a Kubernetes cluster. Kubernetes uses another abstraction to provide persistent storage called Persistent Volumes[9]. These persistent volumes can be provisioned by many storage providers, but mounted as if they were local storage by pods in

[6] min.io
[7] this artifact was intended as a background process for system, hence the name
[8] https://github.com/iterum-provenance
[9] kubernetes.io/docs/concepts/storage/persistent-volumes

the cluster. This means that a daemon, mounting a persistent volume provisioned by Amazon S3 for example, can use this storage as if it is a local storage (back end). This is why the extra storage back ends for daemon instances outside a Kubernetes cluster were left less explored.

User interface

In general, each of the infrastructural components expose a REST API through which requests can be made. All of the functionality of the framework is exposed through these REST APIs. This allows for the creation of different user interfaces consuming these APIs. The usage of a REST API allows for decoupling of the the user interface and the rest of the application. In terms of extensibility of the current implementation, this means that the creation of other user interfaces such as a web interface can be achieved without additional effort in terms of integration.

For Iterum, a CLI is created which allows users to interact with the system[10]. This CLI consumes the REST APIs of the *Manager* and the *Daemon*. The CLI currently consists of two main subsystems, one to manage Iterum data sets, and another to manage (the deployment of) pipelines. The data set management subsystem allows the user to create a new data set, commit new versions, branch off, and remove data sets. In order to use the versioned data set as an input for a pipeline run, the data set needs to synchronized with a *Data versioning server*, which stores the data in the specified storage back end. The CLI also provides the tools necessary to do so.
The pipeline subsystem of the CLI allows users to submit pipeline runs. The CLI passes this specification on to the manager which is then able to schedule the pipeline. The user can request the status of pipelines currently deployed using the CLI.

4.2.2 Additional Services

Three services have been added to the system in order to support the workings of the pipeline run artifacts. All of these services are added in order to support the sending and retrieving of fragments between the different subcomponents of a pipeline run.

Message Queue and Distributed Storage

As described before, a fragment is split into a fragment description and fragment data. The fragment description is small enough to be passed around via channels on the message queue, but the data belonging to the fragments is not. This data is stored on the distributed storage. This is done such that it can be retrieved again by the consumer of the associated fragment description that came off of the message queue.

Note that it is also possible to send the fragments between the different nodes in a peer-to-peer fashion, without the help of the described services. For this PoC the decision was made to add these services, as the usage of a centralized message queue simplifies the communication between the subsystems of a pipeline run. Additionally, message queuing systems provide fault-tolerance and persistence guarantees which are harder to accomplish when the software artifacts communicate in a peer-to-peer fashion. The same argument holds for the additional in-cluster distributed storage.

Kubernetes Admin

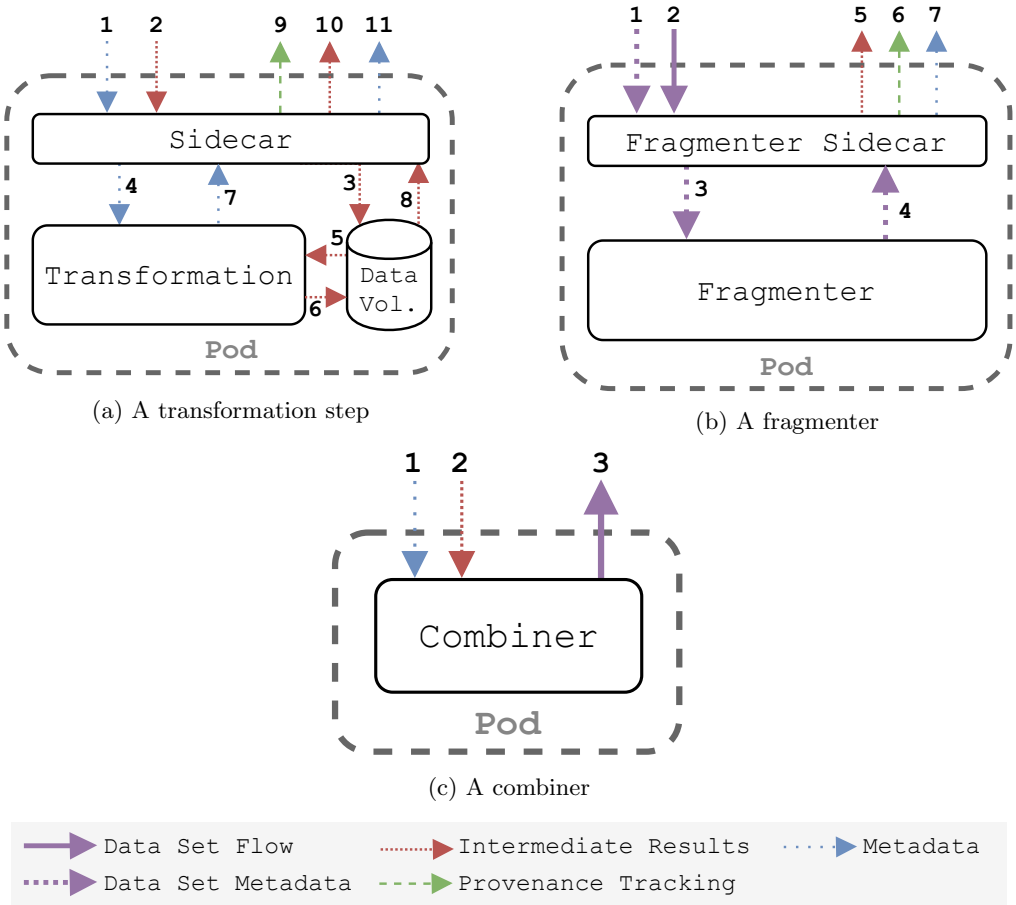
As stated before, the pipeline run artifacts are deployed as Kubernetes jobs. This is done using the Kubernetes Admin. In a Kubernetes cluster, any user or service

[10] github.com/iterum-provenance/cli

account with the correct permissions is able to manage resources on a cluster[11]. The Kubernetes API can be used to create, modify or delete resources, provided that the user is authorized[12]. For the Iterum framework these functionalities are used to control the creation and the management of the pipeline run artifacts. The software artifact of the manager is deployed such that it is able to deploy new jobs, but also request the status of the deployed jobs.

4.2.3 Ephemeral components

Every time a pipeline run is submitted to the *Manager*, some artifacts are scheduled to run on the cluster. As described in 4.2.1, the different components are scheduled as Kubernetes jobs. The lifetime of each of these artifacts is limited. These ephemeral artifacts start when they are scheduled, and complete whenever there is no data left to process. Figure 4.2 shows an implementation overview of these components. These diagrams are explained in more detail throughout the rest of this section.



(a) A transformation step

(b) A fragmenter

(c) A combiner

Figure 4.2: Implementation of the ephemeral components with numbered interactions

[11] kubectl.docs.kubernetes.io/pages/kubectl_book/resources_and_controllers
[12] kubernetes.io/docs/reference/access-authn-authz/rbac/

Sidecar Concept

In this implementation of the framework, the user is able to provide any Docker image containing the computational part of a transformation step. In Iterum, two containers are combined in each of the pods of the Kubernetes job. One container being an instance of the transformation step image the user provides, and the other container is a "sidecar". This sidecar is responsible for communicating between the user-provided transformation step and the other components of the pipeline run. The sidecar is the implementation of both the abstraction and input/output interfaces described in Figure 3.4. Since the transformation only needs to interact with the sidecar, the implementation of the transformation can remain minimal, only focusing on the actual computation.

Even though the sidecar abstracts away from all communication logic between a user-provided transformation and the rest of Iterum, the transformation still has to communicate with the sidecar. Theoretically, the user can write this logic since common methods are used to communicate (network sockets) using common formats (JSON). However, to lower the entry barrier for the users, client libraries can be created which abstracts this communication away as well. This PoC is accompanied by a Python library called Pyterum[13] which provides this functionality for the Python language. This language was chosen as it is one of the main tools used by data scientists today. The concept of a sidecar also helps in keeping these languages libraries as small as possible, since the sidecar takes care of a lot of the work otherwise included in the language library. An example of how a user-provided transformation step using Pyterum looks like can be found in Appendix B.1.

Fragmenter

The fragmenter component consists of a so-called fragmenter sidecar combined with the user-provided fragmenter. The interaction between these containers is shown in Figure 4.2b. The numbers in this figure correspond to the following descriptions:

1. As the fragmenter sidecar is started, it retrieves the list of filenames associated with the provided version of the data set from the storage interface.
2. The actual data from the files is retrieved in a one by one fashion.
3. The list of filenames is send to the user-provided fragmenter which uses this list to fragment the file list as described by the image.
4. The fragmenter sends subsets of the provided list of files back to the sidecar. Each subset indicates one fragment.
5. The fragmenter sidecar simultaneously uploads all files one by one to the distributed storage whilst the fragmenter is producing file lists.
6. Once all files associated with a fragment are uploaded a fragment description is created using the file list generated by the fragmenter. The fragmenter sidecar uploads the provenance information of the produced fragment (description) to the provenance tracker py publishing it to a specialized message queue.
7. The fragmenter sidecar sends the fragment description to the message queue so it can be processed by the first transformation step.

[13] github.com/iterum-provenance/pyterum

Transformation step

A transformation step consists of a sidecar and the user-provided transformation. The communication between the sidecar and the transformation step container uses a shared volume and a shared network between the two images. The shared volume is used by the sidecar to store downloaded files in order to make them accessible to the transformation step provided by the user. The shared network is used to communicate from the sidecar to the transformation step which fragment is ready to be processed. The transformation step then lets the sidecar know which fragment has been processed. This allows for a decoupled structure in which the sidecar coordinates with the framework and downloads and stores the work to be done by the transformation step. Then, as the work becomes available on the shared data volume of the pod, the sidecar informs the transformation step that the workload is ready to be processed. Storing files on the shared volume allows the sidecar to buffer the next few fragments and corresponding files to be processed, improving the performance. Finally, it also increases the accessibility of the framework as users can write their transformations as if they read in data from the disk. It is very likely that they are familiar with these kinds of operations, making working within the framework more natural. In Figure 4.2a the interaction between the containers is shown. Similar to the description of the fragmenter sidecar, the numbers in this figure correspond to the following descriptions:

1. The sidecar retrieves a fragment description from the message queue.
2. The sidecar downloads the file(s) associated with the description from the distributed storage.
3. The sidecar stores the downloaded file(s) in the volume shared by the containers in this pod.
4. The sidecar passes the fragment description to the transformation step, indicating its data is available for processing.
5. The transformation step can now retrieve the file(s) described by the fragment description and process them in any way.
6. The transformation step stores the transformed data on the shared volume.
7. The transformation step informs the sidecar that a fragment has been processed by sending a new fragment description.
8. The sidecar retrieves the new data associated with the new fragment description from the shared volume
9. The sidecar sends provenance information regarding this new fragment to the provenance tracker via the message queue
10. The sidecar then uploads the new data to the distributed storage
11. Finally, the sidecar publishes the new fragment description on its output channel as input for the next step.

Combiner

After all of the fragments have been processed by the user-provided transformation steps, the results need to be stored. The combiner is similar to sidecars described in previous section. However, instead of communicating with the transformation step, the combiner uploads the data to the storage interface. Figure 4.2c describes the interactions between

the combiner and the rest of the framework. The numbers in this figure correspond to the following descriptions:

1. The combiner retrieves fragment descriptions from the message queue.
2. The combiner downloads the file(s) associated with the description from the distributed storage.
3. The combiner uploads the data to the storage interface, associating it with the original data set and the corresponding pipeline run.

4.3 How Iterum enables provenance tracking

The overarching goal of Iterum is to track the provenance of change in data science pipelines. In order to do so, Iterum tracks the versions of various aspects of a pipeline. Between these versions, differences can be identified and used to explain deviating results.
Iterum is strict in its specifications, only allowing pipelines to be executed if all its elements are defined within its scope. For each transformation, an explicit Docker image has to be specified along with a tag. This ensures that the exact same version of a transformation is used every time. The pipeline configuration itself, is stored alongside the results of the execution in the versioned data storage. This connects the data set, output of the pipeline, and the definition of the pipeline run to each other. The input data of a pipeline is specified by a commit hash, which defines a specific version of a data set that will be used. The only data sets that are allowed as input for Iterum are data sets versioned through Iterum and accessible via the *Daemon*.
Throughout the run of the pipeline, all intermediate results are stored on the distributed storage component; ensuring that this information remains accessible. For each fragment that passes through the pipeline, lineage information is gathered and stored in the persistent storage as well. This allows the tracing of each and every output fragment and file back through the pipeline, all the way to the original input data.

Chapter 5

Evaluation

This chapter evaluates the results of this research in three parts. The first is an evaluation of Iterum through two use cases. Secondly, it evaluates whether the implementation and subsequently the architecture meet the requirements posed in Section 3.2. This is done in order to understand whether the implementation is a proper representation of the architecture and the architecture of the requirements. Finally, the reproducibility and provenance tracking capabilities of the architecture are analyzed in more detail. Pitfalls and limitations are identified and analyzed in order to understand where these come from and what their impact is.

5.1 Use Case 1: FeatBoost

In order to assess the viability of the framework, two actual data science experiments have been ported and re-implemented within the Iterum framework. The first use case is an experiment involving a novel feature selection algorithm called FeatBoost [1]. The algorithm can be used in conjunction with another classifier or predictor. Feature selection is an essential part of data science as the size and complexity of data increases. These algorithms often function as part of a larger system in which features are first selected after which the resulting data set is processed by subsequent steps.

Iterum is better suited to work with data sets which can be processed as a stream, but this use case demonstrates that batch jobs with less, but (possibly) larger, fragments are also a valid approaches.

5.1.1 Experiment Definition

Input data set

A subset of the Isolet data set is used as the input for the pipeline [21]. This is one of the data sets used to evaluate the FeatBoost algorithm. This data set is first versioned using the data versioning component of Iterum in order for it to be used as an input for the pipeline. The entire data set consists of a single file with a '*.mat*' extension. It contains 1560 records with 617 features each, totalling roughly 5MB in size.

Pipeline definition

For the purpose of using FeatBoost for Iterum, a basic pipeline consisting of two transformation steps is used. The fragmenter of this pipeline fragments the file as a single

fragment, producing a single element fragment stream. This is viable due to the relatively small size of the data and the dependency of the algorithm on all data at once. The first step of the pipeline is a preprocessing step, it parses the data from the file into the correct Numpy structures[1]. The second step performs the actual FeatBoost algorithm on the preprocessed data, which consists of first training the classifier on the data and subsequently transforming the data into the set of selected features. This transformed data is then send to the combiner, which is responsible for storing the data back into the persistent storage along with the original data set.

5.1.2 Results

The results of the various editions of the experiment are displayed in Table 5.1. This table shows run times for each of the transformation steps as well as the overall run time. In the cases of Iterum, the sum of processing steps does not add up to the total. This is due to the parallelization in Kubernetes and the preprocessing step finishing up after the FeatBoost step is already running. The true time that the preprocessing pod is online in Iterum is therefore also longer than the value depicted here. Due to randomness of which pods initialize first it may have to wait a little before starting and finalizing. These times do not scale, but are simply latency. The table shows six different runs: A baseline python script containing the pipeline which is run on a local machine, the same script run within a docker container, this docker container deployed as a job in Microk8s, the implementation in Iterum run on Microk8s, the docker container deployed as job on a Google Cloud cluster, and finally the Iterum implementation run on this cluster. The final column shows whether the results of the experiment were identical to the ones observed in the baseline experiment.

Lineage data on the fragment level is tracked. This is not explicitly presented here as the data generate by this experiment is trivial. This is due to the simple chain of transformation steps and the single fragment that is passed through it. A more interesting example is presented in Figure 5.1 for the other experiment. For this experiment it would just be a simple 3 element chain.

run times →	**Total**	**Preprocess**	**FeatBoost**	**Fragmenter**	*results*
Local	00:19:18	< 00:00:01	00:19:18	N.A.	baseline
Docker	00:19:49	< 00:00:01	00:19:49	N.A.	identical
Microk8s	00:30:23	< 00:00:01	00:30:23	N.A.	identical
Iterum	00:30:20	< 00:00:01	00:30:19	< 00:00:01	identical
GCloud	00:25:48	< 00:00:01	00:25:48	N.A.	identical
Iterum	00:26:00	< 00:00:01	00:26:00	< 00:00:01	identical

Table 5.1: Results of various editions of the FeatBoost experiment and their run times

5.1.3 Evaluation

As described, this experiment revolves around a single element fragment stream running through two transformation steps and a fragmenter and combiner in a sequential fashion. Since the structure of the experiment is very simple, one would not expect Iterum to

[1] numpy.org

impact the performance by much. This is verified by the performance seen in rows three through six of the table. The slowdown clearly comes from running the experiment on Kubernetes (running on top of an operating system), rather than the abstractions introduced by Iterum. This slowdown due to Kubernetes is expected in these cases as the pipeline is not very complicated and not very parallelizable, meaning that the benefits of Kubernetes are used minimally. Measuring the Iterum performance impact is not really possible with this experiment due to its simplistic nature, but will become more apparent by analyzing results from the other experiment.

What this experiment does show is the ability to run batch jobs using Iterum, as well as its ability to properly (re)create experiments. Each of the experiments shown in Table 5.1 produces identical results, showing that Iterum does not interfere with the possibilities. The single chain of lineage information is still very valuable. It shows for each step of the pipeline, the version of this step, what data set goes in, and what results it produces. This allows for reconstruction and analysis of what happened even after the experiment finishes. All in all, Iterum does not limit the performance of this experiment, but enables a better reproducibility thanks to the versioning of the pipeline run configuration and the intermediate results combined with a lineage path of each fragment.

5.2 Use Case 2: Egocentric Photo Streams

The second use case comes from a paper authored by Talavera et al[18]. In this paper, a tool is introduced for automatic discovery of routine days of an individual based on egocentric photos[2]. This tool consists of a pipeline made up of computational steps which is able to classify the egocentric photos of one day into either a routine day or a non-routine day. In addition to the tool, the paper introduces a data set consisting of egocentric photos of users which is used to evaluate the tool.

5.2.1 Experiment Definition

The tool introduced in the paper consists of a couple of separate computation steps, which can be mapped well to the transformation step abstraction of Iterum. Additionally, some of the steps in the pipeline can start the computation without requiring the whole data set to be present, enabling the data to be processed as a stream of fragments. The pipeline presented in the paper can be described as follows:

1. Image semantics extraction using convolutional neural networks (CNNs) to generate labels per image.
2. Temporal document construction by aggregating all generated labels over certain periods of time.
3. Generate topics per time slot using all temporal documents of a user (using LDA [5]).
4. Unsupervised routine discovery using clustering methods per user. This results in the prediction of a label per day.

(5.) Evaluate the results using the ground truth labels.

[2]Photos taken from the point of view of an individual

Input data set

The paper introduces a data set called EgoRoutine, which consists of egocentric photos of 7 users for a total of 104 days. This results in the distribution of pictures as depicted in Table 5.2 totalling a size of roughly 69 GB. These images can be used to determine what a user was doing at a certain time of the day.

User	*1*	*2*	*3*	*4*	*5*	*6*	*7*	**Total**
Num. days	14	10	16	20	13	18	13	104
Num. imgs	20521	9583	21606	19152	17046	16592	10957	115430

Table 5.2: The distribution of egocentric images per user

Pipeline definition

The pipeline is fairly straightforward to implement. The next step depends on the previous step and so the resulting graph is simply a chain. The initial artifact is the fragmenter, which generates one fragment per image and attaches some meta data about the image such as the (time of) day, user, ground truth, and time slot. As the fragments become available the first transformation step transforms the images into a list of labels.

In the original paper this labeling is done using three different types of CNNs. Each of these detect a different aspect of a photo: object detection (Yolov3, Xception), scene recognition (VGG16) and activity recognition (Cartas et al.) [16, 8, 17, 7]. In the paper, an ablation study is performed to find out which CNN results in the best results. It is concluded that a combination of multiple CNNs yields the best results. For this implementation only the CNN used for object detection (Yolov3) was ported. The label generation is by far the most resource intensive step and processing nearly 70 GB of data on a single machine takes a very long time. Increasing the complexity of this step by repeating this process two more times for the other labels was therefore omitted. The results of the pipeline using only this object detection CNN are also included in the original paper, so evaluating results in Iterum is still possible. In order to deal with the still heavy workload of this step, Iterum provides the ability to spin up multiple instances processing the input stream simultaneously. The implementation of this pipeline heavily uses this fact by spinning up multiple of these "labeler" transformation steps. This is possible thanks to the way Iterum uses the Kubernetes abstractions. This step allows for this approach since each fragment is processable independently of all the others.

1. As described, this transformation transforms the stream of images into a stream of lists of labels per image.

2. This second step aggregates elements of this stream per user and per time slot. All these labels are then converted into a document per time slot concatenating all label lists of a user within that time slot. The result of this is a stream of lists of labels aggregated per user per day per time slot.

3. The third transformation step aggregates all data per user in order to vectorize the document as well as train the LDA model on these vectors. Then, for each time slot a vector of probabilities for each topic is generated by the LDA model. This results in a stream of topic probability vectors per user per day per time slot.

4. The next transformation again aggregates results per user day in order to compare each day of a user with each other day. This comparison is performed using DTW

[11]. Afterwards, the days are clustered into either the non-routine or routine category.

5. The list of predicted labels is then send to the final transformation step, which evaluates the performance of the pipeline by comparing the predicted labels with the ground truth labels.

6. The results are then send to the combiner which stores the results back into the Persistent storage.

5.2.2 Results

The predictions made by the pipeline are compared with the ground truth values, and various metrics are computed using these results.

Comparison to the original experiment

Evaluated predictions of the pipeline can be found in Table 5.3. These results are produced when running the experiment in Iterum on the k3d implementation of Kubernetes. The actual results of the experiment, as stated in the paper introducing the pipeline [18], can be found in Table 5.4. Note that this table shows results per user, and these results are retrieved by using multiple different CNNs for the labeler step. The complete results per user when only using the Yolov3 CNN are not presented in the original paper, though the average results over the users are. These are added as an extra column in this table. Both versions of the experiment show similar results, though there are minor variations.

User	*1*	*2*	*3*	*4*	*5*	*6*	*7*	**Avg.**
Acc.	0.79	0.60	0.63	0.70	0.92	0.50	0.85	0.71
F-1	0.75	0.58	0.50	0.70	0.92	0.46	0.84	0.68
Prec.	0.75	0.60	0.50	0.77	0.93	0.54	0.89	0.71
Rec.	0.86	0.62	0.50	0.77	0.93	0.57	0.83	0.73

Table 5.3: The accuracy, F1-score, precision, and recall respectively per user for the experiment ran within the Iterum framework using only the Yolov3 CNN.

User	*1*	*2*	*3*	*4*	*5*	*6*	*7*	**Avg.**	**Avg. Yolov3**
Acc.	0.79	0.74	0.75	0.90	0.92	0.56	0.92	0.80	0.71
F-1	0.75	0.70	0.71	0.89	0.92	0.50	0.92	0.77	0.68
Prec.	0.75	0.75	0.70	0.89	0.93	0.56	0.94	0.79	0.72
Rec.	0.86	0.79	0.75	0.89	0.93	0.60	0.92	0.82	0.74

Table 5.4: The accuracy, F1-score, precision, and recall respectively per user as presented in the paper. The averages for running the experiment only using the Yolov3 CNN are appended

Comparison between different run-time environments

In order to analyze both the performance impact and reproducibility of Iterum, the experiment is run in various run time environments. Baseline values were produced by re-implementing a version of the original experiment. The average results per user can be found in Table 5.5. For each of the editions of the experiment, the run times of each transformation job are stated in Table 5.6. Note that for the Iterum variants of the experiment, the data set is processed as a stream, and therefore steps further in the pipeline start processing data as soon as it becomes available. The duration for these jobs are therefore not a true representation of the total run time as they have been online for a longer time and may have processed some data. The times represent the amount of time it took the step to finish after the previous step completed. This is still deemed a valid comparison since the other experiments show that the total run time is dominated by the "labeler" step anyway. The increased times for the timeslot aggregation step for the Iterum experiments in comparison to the baseline are further discussed in the evaluation.

Results	**Accuracy**	**F-1**	**Precision**	**Recall**
Local	0.69	0.66	0.72	0.72
Docker	0.69	0.66	0.72	0.72
k3d	0.69	0.66	0.72	0.72
Iterum	0.70	0.66	0.70	0.70
GCloud	0.69	0.66	0.72	0.72
Iterum	0.71	0.68	0.71	0.73

Table 5.5: The accuracy, F1-score, precision, and recall for the experiment ran in different environments

Durations	**Fragmenter**	**Labeler**	**Time Aggr.**	**Topics**	**Clustering**
Local	N.A.	6:06:47	0:00:00	0:00:05	0:00:02
Docker	N.A.	5:55:28	0:00:00	0:00:04	0:00:01
k3d	N.A.	5:41:10	0:00:00	0:00:04	0:00:01
Iterum	0:22:05	6:48:43	0:19:10	0:00:08	0:00:11
GCloud	N.A.	11:04:12	0:00:00	0:00:07	0:00:02
Iterum	0:43:46	10:18:09	0:50:31	0:00:15	0:00:14

Table 5.6: The duration of each transformation step per variant of the experiment. The times required to compute the evaluation step are omitted, as all values were less than 1 second.

Distributed computing

To evaluate the distributed functionality of Iterum, the experiment was also performed on a cluster of 3 computation nodes (using Google Compute Instances) rather than 1

computational node. The results and the duration of these experiments can be found in Table 5.7 and Table 5.8 respectively. Note that the single computational node still uses 3 labelers, but these are all scheduled on one computational instance, therefore still restricted by the resources available on this single instance. The experiment with 3 computational nodes uses 64 labelers, scheduled over 3 different nodes, also being able to access the resources of these 3 nodes.

Results	**Accuracy**	**F-1**	**Precision**	**Recall**
1 node	0.71	0.68	0.71	0.73
3 nodes	0.71	0.68	0.71	0.73

Table 5.7: The accuracy, F1-score, precision, and recall for the experiment ran on either 1 computational node with 3 instances of labelers, or 3 computational node with 64 instances of labelers.

Durations	**Fragmenter**	**Labeler**	**Time Aggr.**	**Topics**	**Clustering**
1 node	0:43:46	10:18:09	0:50:31	0:00:15	0:00:14
3 nodes	0:58:08	2:38:56	0:50:04	0:00:13	0:00:16

Table 5.8: The duration of each transformation step for the experiment ran on either 1 computational node with 3 instances of labelers, or 3 computational node with 64 instances of labelers.

Data Lineage

Iterum collects multiple types of provenance information; one of which is the data lineage information which links input fragments to output fragments throughout the pipeline. An example of lineage information produced for the resulting fragment of a single user is shown in Figure 5.1. It shows how lineage information is tracked at every transformation step, for each fragment passing through it. The tree construction comes from the multiple aggregation steps performed for this experiment, eventually resulting in one evaluation output for a user. The evaluation output is generated by aggregating over multiple outputs of the previous step, as indicated by the horizontal dots. The vertical dots imply that these sub-trees grow similarly to the one to the left of it. This structure is repeated on the other levels as well. Every fragment is given an identifier, which transformation step produced it and which data files it contains. By linking this information to its predecessors and descendants a lineage tree can be constructed.

5.2.3 Evaluation

Table 5.3 and Table 5.4 show approximately the same results, though there are minor differences between the two. First of all, the experiment performed in the original paper produced results by using multiple different CNNs to extract semantics of images, whereas the editions in this research only use the Yolov3 CNN. Both the full results and Yolov3 averages from the original paper only deviate slightly from the results produced by this research. One possible explanation for this is that Iterum does not guarantee the ordering of the stream. The topic modeling step in the pipeline trains in batches, and is therefore affected by this, yielding slightly different outcomes for the experiment.

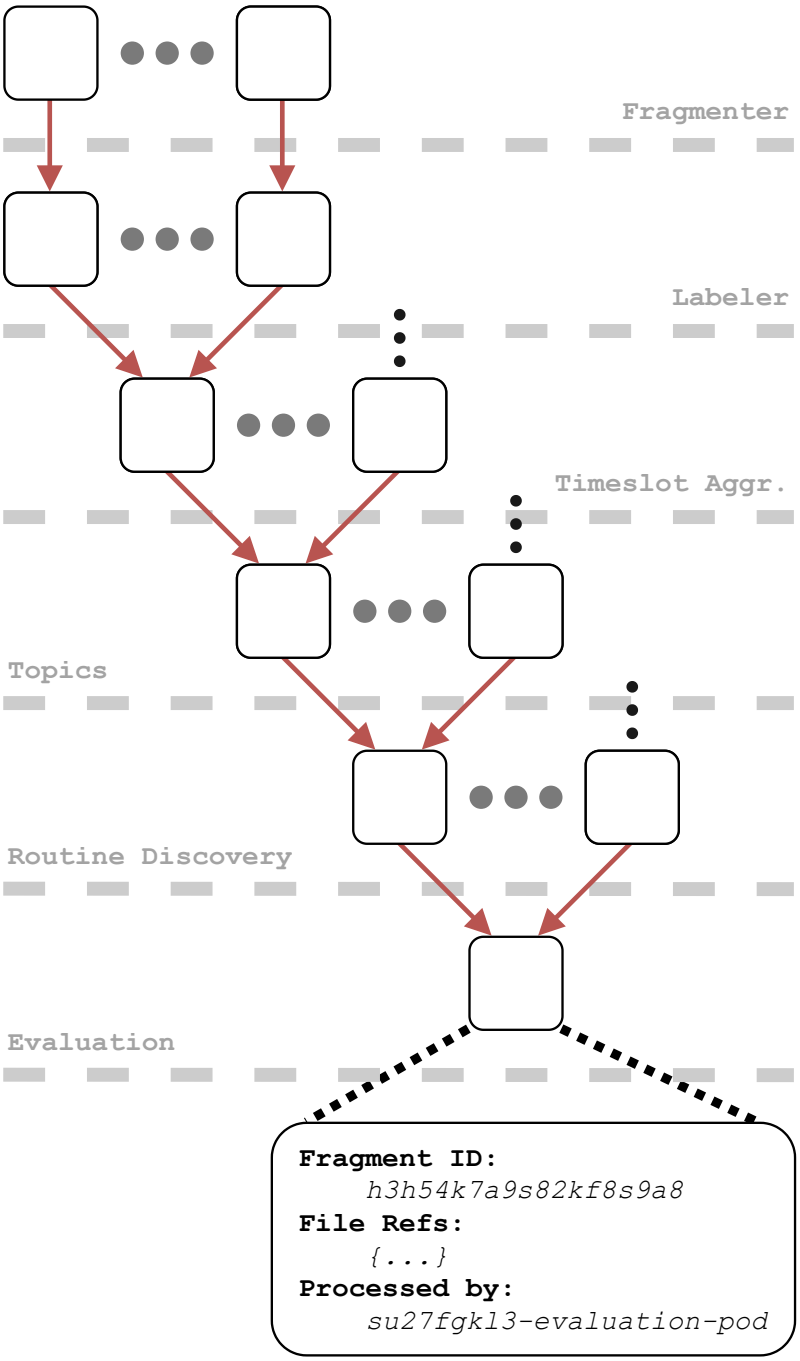


Figure 5.1: An example of the kind of lineage information produced by Iterum

Another possibility for the differences in results is the usage of a different seed for the topic modeling step.

Table 5.5 shows results of runs using different run-time environments. These results give insight into how much overhead Iterum introduces. The results of each non-Iterum run are the same, as the ordering of data points and the random seed were kept the same. The experiments run with Iterum produce slightly different results; this is due to the

lack of ordering of a stream.
In Table 5.6, the duration of each of the transformation steps is shown. The experiment which ran on Google Cloud took more hours to complete than the experiment ran on k3d. This difference is largely explained by the different amount of computational resources available and the additional network overhead introduced by communicating with remote nodes and disks (different hardware means different performance).

Iterum does however seem to introduce some overhead. The time it takes to perform the timeslot aggregation is higher than expected, the most likely explanation for this is the high IO overhead introduced by Iterum. Whereas the single job pipeline reads in the pictures for step one and can continue with all results (string labels) in memory, Iterum reads and writes each result to a file multiple times for each transformation step. Additionally, the network overhead of serving 115k small files can also have an impact on the GCloud cases, both with and without Iterum. The reason the GCloud experiment takes such a long time is that it possibly had to retrieve its data from a different node or non-local disk. This introduces a delay. Also minimal paralellization will have had an effect.
These two reasons would also explain why the other steps do not show this performance dip, as results are aggregated and the 115k file set is reduced to less than 1000. This effect is amplified in the GCloud case, due to possible absence of data locality. For example, resources such as persistent disks can be in physically different locations than the CPUs for example.

In Table 5.8 the time it takes to run the experiment on either 1 node or 3 nodes is shown. Using more nodes yields clear speed ups of the experiment; yet using 64 labelers is somewhat excessive. The speed increase is limited by two factors: speed at which the fragmenter produces fragments, and the processing time of a fragment for a labeler. The latter is in turn impacted by the computational resources available to the labeler. When more labelers are created, the resources are divided, but after some threshold all labelers slow down due to low availability of CPU resources. However, there are more labelers working in parallel, so the overall performance remains about the same. Having many instances guarantees that the workload would be properly distributed over all available computational resources[20].

5.3 Requirements Evaluation

This section evaluates how well the different requirements described in Section 3.2 are met by Iterum. It serves as an analysis of how representative the implementation is of the designed architecture and, in turn, how well the architecture represents the defined requirements.

5.3.1 Data Provenance

Tracking the data provenance is an important aspect of reproducibility, as described in Section 2.4.1. However, existing version control tools such as Git are not sufficient for a data versioning system, as the files stored in a data set often are of a different nature and are larger in size. Iterum introduces a data versioning system which allows users to keep track of versions of a data set. The data versioning system integrates with the rest of the framework, but can also be used without the pipeline system.

1. *Define a storage format within which data sets are stored*
   Iterum introduces a storage format which allows data sets to be stored in a persistent manner. The format is designed in such a way that it can be stored on a

file-system, where metadata and configuration files are stored using common file formats such as JSON. These files contain the metadata of a data set: information about the data set in general, a list of versions, how versions relate to each other, and any other information necessary for maintaining proper versions. The data itself is also stored along the metadata files on a file-system. Additionally, information about a pipeline run, and results of a pipeline run execution can be stored in this data set format as well, allowing for retrieval and reproducibility of experiments at a later time.

2. *Track all of the different available versions of the data set*
   The data versioning system stores all of the required information related to a data set in the storage back end. The different versions of the data set that have been added by users compound to a version tree. This version tree can be retrieved by the user to gain insight into the evolution of the data set over time.

3. *Commit additions, removals and modifications to the data set in order to create new versions*
   The user is able to create new versions of a data set via the CLI provided by Iterum. Files can be added, removed or modified, and subsequently be committed to the data versioning server. This data versioning server stores the files and makes sure that the new version information is stored in the storage back end of the data set.

4. *Support roll-backs to previously committed versions of a data set*
   The user is able to roll-back to a previously committed version of a data set by checking out to this version via the CLI. This is conceptually similar to how the Git versioning model works, and should therefore be familiar to users accustomed to using Git.

5. *Store (intermediate) results of a pipeline execution related to a (versioned) data set*
   The results of a pipeline execution are stored in the storage back end where they are associated with the original data set that was used as input of the pipeline. Currently, Iterum does not support the persistent storage of intermediate results, though they are stored in the distributed storage. The infrastructure for storing the intermediate results persistently is present in the implementation, so adding this functionality in the future should be relatively simple.

6. *Maintain lineage information about the data during the execution of a pipeline*
   The provenance tracker component, which in the current implementation is handled by the *Manager*, receives the lineage information about each fragment during the pipeline execution. After the pipeline execution has completed, this information is send to the *Daemon* which stores this information in the storage back end, where it is associated with the data set. Users can then retrieve this information about the specific fragments and gain insight into the origin of a fragment. It also shows how it was transformed over the course of the pipeline execution.

7. *Be agnostic towards where versioned data sets are stored as to ensure vendor locked users can still use the framework (e.g. AWS-S3, local storage, private storage)*
   The *Daemon* fulfills the role of the storage interface. Currently, Iterum only provides storage bindings for volumes mounted by the *Daemon*. Persistent storage needs in a Kubernetes cluster are met using the PersistentVolume abstraction. Since the *Daemon* is often deployed on a Kubernetes cluster, the storage types that can be provided via this abstraction. These volumes can therefore be used as a storage back end for a data set and in that sense Iterum fulfills this requirement.

However, in the situation where the *Daemon* is used without a cluster (when a user only wants to use the data versioning component), the requirement is not supported. Iterum does not provide the implementation to connect to third party storage services. The necessary abstractions and interfaces are present in the source code so that this can be added at a later point. Using the Daemon on a local machine, with a local storage mounted, is still possible.

8. *Store data sets remotely such that multiple users are able to work on the same data set simultaneously*
   Data sets can be stored on a remote storage back end such that multiple *Daemons* can point to the same storage. This allows multiple users to work on the same data set simultaneously, each using their own instance of a daemon. It is also possible for multiple users to connect to the same daemon. Multiple daemons and one storage however, currently only works for a volume which is mounted by each daemon at the same time (for example, a NAS connected to multiple machines on which a daemon is hosted). The functionality can be supported in full when the implementation for connecting to an external storage provider is provided by Iterum rather than the Kubernetes PersistentVolume abstraction.

5.3.2 Pipeline Provenance

This evaluation is left to the other thesis [2]

5.3.3 Pipeline Runtime Management

The requirements described by Func. Requirement 3 focus on the pipeline deployment functionalities of the designed framework. Iterum focuses on this aspect, since it is considered one of the two core systems for which it is developed. In the deployment and execution of pipeline artifacts performance and accessibility play a large role, which is why most requirements focus on these aspects.

1. *Deploy pipeline runs based on their definitions over available resources*
   This requirement is one of the core features that the Iterum framework provides. Defined pipeline runs can be submitted to the Manager component which deploys the associated artifacts by making calls to the Kubernetes API. Kubernetes is then responsible for distributing the workload over the available resources.

2. *Deliver data to relevant endpoints without additional user specification besides the pipeline*
   As described by Section 4.2.3, this is one of the reasons that the sidecar is introduced. The sidecars for both fragmenters and transformation steps are used in order to fulfill this requirement. Based on the pipeline run specification the sidecar retrieves fragment descriptions from the message queues. Based on this description, data is then downloaded from the distributed data storage. Results from the transformation are uploaded and posted as description accordingly. This abstracts sending and retrieving data such that the user can focus on writing the actual transformation.

3. *Be agnostic towards the experiment performed in terms of programming language, shape of data, and execution environment*
   This requirement in the strictest sense is not entirely met by Iterum. The fragment abstraction covers the shape and type of data, such that is free to be chosen by the user. Thanks to tools like Docker and Kubernetes, the environment is abstracted away from the execution. A major focus of these tools is to create reproducible

execution environments. The programming language is the element that is not entirely met by Iterum. Due to some processes that cannot be automated as described in Section 3.1.2, an interface between the user-defined transformation and the sidecar is needed. This so-called language client library can be kept tiny thanks to all the work done by the predefined sidecars. In the extreme case this library can be implemented on a per case basis as it uses three easy to parse message types in a JSON format over (network) sockets. These are elementary concepts supported by every major language. Moreover, this language client is provided for the Python language as it is one of the major languages in data science today.

The fact that users can provide Docker images for transformation steps, and that what happens within the transformation steps is considered to be a black box, ensure that a user is also able to communicate with other services outside the context of Iterum or its architecture. This opens the door towards, for example, performing a batch processing job using a Spark cluster which is accessible from within the Iterum cluster. Users can then still use the provenance tracking capabilities of Iterum, which Spark lacks, whilst also reaping the benefits of performing computations using a Spark cluster. Note that by using services outside the control of Iterum some of the guarantees regarding reproducibility are lost, as the applications are theoretically possible to connect to services with an changing internal state. A user creating such a pipeline will need to take these dire consequences into account, by choosing non-stateful applications for example.

4. *Scale individual transformation steps when indicated by the user*
   Iterum provides this functionality by allowing the specification of how many instances of a transformation step should be initialized. The manager then invokes the Kubernetes API to spin up multiple copies to complete the associated job in parallel. It is important to note that users should themselves realize whether this is a valid option as ordering and which message goes to which step can not be guaranteed. This comes from the fact that all copies created in this way consume from the same input queue(s) and send to the same output queue(s). This feature is exploited by the implementation of the use case described in Section 5.2, by initializing multiple labeler transformations. The processing of images can be done in an independent fashion, so this is a prime example where this is a valid approach.

5. *Given intermediate results and lineage information, allow restarts of a pipeline from an intermediate point, without having to re-execute all upstream nodes*
   This requirement is not entirely met by Iterum, however all its prerequisites are. Intermediate results of each transformation step are saved, as well as lineage paths of every fragment. This information together allows the reconstruction of a fragment stream by moving up through the lineage hierarchy up until the point from where to restart. Each fragment at this level has both a description and data saved in the intermediate results. So, when fed to the new starting point, the pipeline run can be restarted from that intermediate point. Iterum does not support this feature, but based on the prerequisites that have been met, its implementation should be fairly straightforward.

6. *Function in both a distributed and single (computation) node setup*
   Iterum enables this thanks to the usage of Kubernetes and its many implementations. If Kubernetes runs on the associated resources, so will Iterum. On clouds

and clusters Kubernetes can be installed, for local development light weight alternatives can be used such as Microk8s[3] and K3s[4].

5.3.4 User Interaction

User interaction has been less of a focus for Iterum. This is due to the fact that the time constraints of this research did not allow for a full-fledged implementation of the architecture. In order to show the provenance tracking capabilities of the system, the allotted time was spent on developing the provenance tracking features of Iterum. From the perspective of this research, these features are more interesting than the user experience, even though for a production-ready version these features are critical in order to improve the accessibility. As discussed in Section 6.3 and Appendix C, proper UI design is critical and need investigation in the future.

1. *Present one or more user interfaces which allows the user to interact with the various elements of the system*
   The CLI presented in Section 4.2.1 is responsible for handling the user interaction. As described, it consists of two parts, each of which is responsible for interacting with one of the major subsystems (data versioning and pipeline run deployment). This is done by consuming the REST APIs exposed by the other software artifacts. Presenting additional user interfaces can be achieved by consuming these REST APIs, similar to the CLI.

2. *Both results and lineage information of an executed pipeline should be both presented to, and retrievable by the user*
   The CLI allows the user to retrieve lineage information of specific fragments. The user is also able to retrieve an overview of the fragments connected to a pipeline run. For each of these fragments, the user can retrieve information about how this fragment relates to other fragments in the pipeline.

3. *Notify the users about the progress of a pipeline run* This feature has not been implemented, as the notification component was given a low priority. Due to the available tools it is however possible to inspect the progress of the pipeline in multiple ways. This is an active action by the user, rather than the user getting informed. Getting the status of pods is a functionality that the Kubernetes CLI provides and can be used to inspect which parts of the pipeline have finished running or resulted in an error. By inspecting the logs of the containers in those pods, users can gain insight into potential problems. This process could easily be automated for users such that they can view this information at any time, rather than retrieving it themselves. In short, Iterum does not provide an interface to check this information, nevertheless it is possible for the user to retrieve it.

5.3.5 Non-Functional Requirements

Accessibility

User experience is critical to the success of a framework that aims at such a diverse set of users. Scientists able to write code to express their experiments should be able to use this framework, making accessibility one of the most important non-functional requirements. As Iterum is not in a production-ready state, this requirement could not be evaluated with actual users. Instead the use cases were developed by the developers of the framework, this makes fully evaluating this requirement impossible. Nevertheless,

[3] microk8s.io
[4] k3s.io

many choices in the design of Iterum have been made from the perspective of a user, these choices are outlined here.

1. *Try to explain from where inconsistencies originate in the case of inconsistent results that arise from non-user-defined behaviour such as crashes or failed requests that cannot be recovered from*
   Fulfilling this requirements is very complex, since it implies catching all possible errors and fixing them if possible, if not report them. This is a feature that every software strives for. Many possible complications arise from bad communication, incomplete messaging, and network failure. Many of these functionalities can be checked. To minimize the effort Iterum relies on proven tools such as Kubernetes, RabbitMQ and MinIO. If an error does occur in one of the Iterum components, such as the sidecars, these are all logged and made available to the user. Iterum tries to complete the sunny-day scenario first, then retrying multiple attempts on failure if possible, and eventually, when there is no recovering from the error, informing the user of what went wrong.

2. *Redefining existing experiments within the scope of the system should require minimal effort for a user*
   Changing an existing experiment which has already been run can be changed with minimal effort. The user is able to retrieve the pipeline run configuration in its JSON format. The user is able to change any configurable parameters in this specification. This includes available versions of transformation steps[5], changing the DAG structure, using a different (version of a) data set, and adjusting configurable parameters. After the user has adjusted the experiment, it can be resubmitted and Iterum will deploy the generated artifacts.

3. *Users with enough skill to program their experiments should be able to use this system*
   Programming a transformation on some data set will likely include the parsing of that data set into the context of the program, followed by the actual transformation, before finally saving the results. Iterum keeps this flow more or less intact such that, given that a user is able to code this for his experiment, the user is also able to do that within Iterum. The only additional steps the user has to take are Dockerizing the application, defining the pipeline in a JSON format, and using the Iterum language library which will provide abstractions that feed the data per fragment to the program. Whilst users only have to learn these relatively easy steps, Iterum takes care of challenges such as moving data over the network, deploying the artifacts, publishing and consuming messages and persistent distributed storage.

Performance

Performance is very important for a framework such as this, since it will have a large impact on whether such a system is adopted. If the run times increase exponentially, or certain actions or experiments are unavailable due to performance limitations, users will not use the framework. Iterum boasts the usage of distributed tools and paralellization, but also makes some less performant choices.

1. *Experiments executed within the system should at most take* 1.5 *times the run time of the same experiment executed outside the context of the system*
   The framework does introduce overhead, as data needs to be transmitted over a

[5]Dockerized, using an Iterum language library, and pushed to a container registry available to Kubernetes

network between the different steps of the computation. This would not necessarily have been present when performing the experiment outside Iterum. Additionally, the many IO operations that Iterum performs when moving data around the pipeline also poses an inherent limitation. However, as shown by the use cases, Iterum still performs really well in the single node setup that the experiments were tested in. Performance decrease is mostly due to the extra abstractions of Kubernetes rather than Iterum. Based off of these experiments, Iterum manages to stay under the performance cap.

2. *Handle data sets larger than the memory capacity of a single node*
   Due to the fragment abstraction pipelines process streams of fragments. Each fragment is downloaded file by file and streamed directly to the storage of the pod. This allows for both uploading and downloading of the files associated with a fragment description irregardless of the size. As long as user defined transformation steps do not attempt to read files that are larger than the memory resources available to the Kubernetes pod, this problem will not occur.

3. *The system must be light weight such that it is able to run reliably on a single machine*
   The data versioning component of Iterum is able to run on a single machine. However, as datasets become larger, connecting to an external storage back end such as a cloud storage provider becomes necessary, rather than local storage on the machine of the user. The pipeline component of Iterum requires a Kubernetes cluster to run, but there are light weight implementations which allow for this on a single machine. Experiments with either a small amount of data or light computational load can be run on a single machine. For larger, or more complex experiments, a Kubernetes cluster consisting of multiple (more powerful) nodes can be used. This is not a limitation of Iterum as these cases would often already require larger, more elaborate setups in order to complete, even outside the context of Iterum.

Portability

1. *Interaction with versioned data sets do not require a user to download the entire data set. A client machine with insufficient disk space to fit the whole data set should still be able to use the system*
   The CLI enables users to add or remove data from a data set. Files added to a new version of the data set are uploaded to the data versioning server, allowing users to remove the files from their local machine. Removing files does not require any data as references to the remote data are enough. Users can therefore add files to, and modify or remove files from a data set without actually having the (entire) data set downloaded on the client machine.

2. *Data set development and manipulation works independently from pipeline definitions and their deployments such that users working on one of these do not need knowledge nor dependencies of the other*
   Iterum is split into two subsystems. One focuses on data versioning, the other on pipeline management. This split appears in nearly every component. The CLI has subcommands for these two options and the Daemon software artifact implements the architectural components used for the *Data versioning component* and for the *Storage interface*. Both *Daemon* and CLI can run outside the scope of Kubernetes and therefore the data versioning can be decoupled from the pipeline management. They can however both run within the context of the Kubernetes cluster. Users

can therefore create new versions of data sets, and at the same time other users can submit pipelines using existing versions of that same data set.

5.4 Architecture Evaluation

This section evaluates the architecture by assessing both the provenance tracking capabilities and the ability to reproduce experiments. This combines the design of the framework, its implementation, and its performance on the presented use cases into an overall evaluation of the framework.

5.4.1 Limitations

Iterum attempts to implement the architecture described in Chapter 3. Section 5.3 and Sections 5.1 and 5.2 show the merits, but also some limitations of this implementation. In this section, the limitations that have not yet been covered (in detail), are discussed. Due to the architecture being very generic, it does not show many inherent limitations; due to which most of the limitations arise from the implementation.

Dependency on Kubernetes

One of the important dependencies of this project is Kubernetes. Kubernetes provides the abstractions used for scheduling and deploying the different software artifacts used in Iterum. However, depending on one software dependency for a significant amount of functionality is also a liability for a project. The API of Kubernetes might change with time and older features might not be supported anymore. Even though this may seem a limitation, Kubernetes is open-source and does not seem likely to change drastically as many users depend on it. The benefits of using Kubernetes easily outweigh the (potential) drawbacks. In addition to this, the architecture as described in 3 is not dependent on Kubernetes, only Iterum is. The architecture can be re-implemented using other software dependencies when current tools become outdated or otherwise impractical to use.

IO Operation Count

As explained before, Iterum splits the fragments into a fragment description and fragment data. The data of a fragment is presented to a transformation step as a file written on a shared volume, but is also retrieved from the transformation step as a file written to that shared volume. Using files to send and receive data from a transformation step allows Iterum to handle larger files which might not fit in memory of a single computation node, and it allows the user to process any kind of file present in a data set. However, reading and writing this many files to disk introduces IO overhead. Parts of which could be avoided when processing a data set outside Iterum. Additionally, the data is not messaged directly between transformation steps, but rather passed around via a remote distributed storage. This is yet another additional read-write operation. In total this means three read and three write operations and sending the data over the network twice per fragment and per transformation step.

Restrictive Sidecars

Sidecars help in keeping individual language libraries small whilst simultaneously greatly increasing the accessibility of the framework by abstracting away from communication and data flow between transformation steps. However, the use of these sidecars also comes with a downside; they also restrict access to these abstracted complexities. This

disallows a user to implement efficient selective consumers for example by not being allowed to communicate with the message queues directly. Giving more advanced users the option to optimize for their use case by omitting (the restrictions posed by) sidecars would be a way to deal with this limitation.

Streams with Significant Order

Some experiments produce different results depending on the order of how the data is processed. One example of where this occurs is in the training of a neural network, where the weights of the network can be updated each time a data element is presented. If data items are presented in a random order, the weights of the network get updated slightly differently. A small change in the weights can lead to (larger) differences in results in the end. This problem can be reduced by using *batch* training rather than *on-line* training, this would require the user to adapt the experiment. The severity of this limitation changes based on the definition of reproducibility that one has. In exact experiments that perform some calculation reproducible means to be able to get the exact same results. However, in data science, many experiments involve training some kind of model in order to predict future values. Evaluation of these models is usually done by measuring performance scores such as accuracy, recall and f1-scores. Even though a trained model may be slightly different because of the order it sees the data samples in, its performance may still be very close to that of the original one. In many cases this is seen as reproducible, and sometimes even as a feature. The model is then robust enough not to care about the order of the input whilst performing (approximately) the same.

Ideally, Iterum should provide an option in which ordering of the fragment stream can be guaranteed for some transformation steps. Especially for cases where the order is important beyond yielding an exact copy of a trained model. In this way, more types of experiments can be fully reproduced.

Data versioning: Handling of files other than binary files

The data versioning component was initially designed to work well with data sets that consist of files where the files themselves do not change, but the composition of files does. One example is a data set consisting of photos, where photos are added or removed in newer versions of the data set, but in general the photos themselves are not modified. This is because the data versioning components stores both the complete original file and the modified file in the storage backend, even though 99% may be the same.

To illustrate this limitation: there might be a data set consisting of one large CSV file, 10MB in size. Changing one value in this CSV file results in a different file. Iterum stores a new version of the data set with this modification as a completely different file, storing almost 10MB of the data redundantly. To circumvent this, modified files can also be stored as the differences in the file, and the reference to the original file. This would reduce the amount of data storage required, but it require a system to be created which is able to reconstruct files from a list of differences and the original file.

Work required from the user

Many of the provenance tracking and reproducibility features provided by Iterum are contingent on the users using the provided abstractions properly. Iterum provides the handles necessary for the user to version their data, create reproducible experiments, and gain insight into their results, but this requires users to actually version their data, code and pipelines. For instance, a user might have a pipeline with some transformation

steps. One of the transformation steps uses a *latest* version of a code repository, and submits the pipelines using this *latest* version. However, this *latest* version changes over time. At this point, Iterum is unable to tell the user that this will not yield reproducible experiments. A similar thing holds for the tracking of lineage information. Iterum can only track the lineage of the data if the user supplies the predecessors for each fragment they produce properly.

5.4.2 Architecture Representation by Iterum

As described in 4.1.2, each of the architectural components is mapped to a software artifact. However, some of the architectural components are combined. Since the communication pathways are similar, combining them does not affect the functionality of implementation. However, combining the source code for two different components does introduce some coupling, which is best avoided. One impractical repercussion is that the *Daemon* always needs to be available to the software artifacts of a pipeline run, whereas they only require the *Storage interface* component implemented by it. The same is true in reverse for the *Data versioning* component not needing to be present within the cluster.

Other than the aforementioned deviations, the architectural components are mapped clearly onto the design of the implementation. Iterum introduces some additional components, such as the sidecars, message queue, and distributed storage; these are used in the implementation of the communication pathways between the other components. Iterum currently lacks some features described by the architecture, but this is due to time constraints and prioritization. Iterum is in a pre-alpha stage and functions as a proof-of-concept implementation; its design still carefully considers the envisioned requirements, as described throughout this research. From this it can concluded that Iterum is in fact a valid representation of the (most relevant components of the) architecture, implying that Iterum can be used to validate the conceptual architecture.

5.4.3 Reproducibility

The architecture as well as Iterum enable a user to track all kinds of changes throughout their experiment. From data to code, to pipelines configurations. However, none of this will work as long as the user does not work with the suggested tools and versions his structures using appropriate tools. Given that the user has defined the pipeline run configuration properly, Iterum allows the user to reproduce the experiment. This means proper usage of source code versioning tools such as Git (e.g. tagging releases), but also pushing Docker images to registries, and also tagging these with versions. The same goes for the pipeline configurations. Iterum enables pipeline definitions through a simple format, which can be versioned using tools such as Git again. If all of this is done properly, users can easily run experiments of others, whilst being assured that the transformations, data, and environment are identical to that of the original user. Most, if not all, of these processes could be automated for Iterum in a future version as well.

5.5 Motivation for scientists to use Iterum

The ability to reproduce experiments is not only useful for scientists trying to verify work of others, but also for the creators of the experiment itself. Since these creators will be the ones to implement their experiment within Iterum, there should be a benefit for them as well, not only such that others can recreate them. Being able to track all the described types of provenance and presenting this information in a clean manner can be

of tremendous help to the original scientist as well. Understanding how his experiment evolved over time, seeing, in hindsight, which parameters produced those interesting results that one time, and being able to conclusively see exactly which data flowed where is all very relevant information to scientists creating experiments. Conclusively knowing and having all this information is not the only benefit though, it also opens doors towards even more interesting features such as sensitivity analysis, parameter tuning, and skipping parts of a pipeline because results up till there have already been computed some time in the past. These are not features that were within the scope of this research, and were there left largely unexplored. However these types of features can all be powered by the type of information that Iterum and its architecture track.

Chapter 6

Conclusions

This chapter reflects on the research questions posed in Section 1.2.1. It correlates the presented framework, its implementation, and their evaluations in order to answer all questions generated by the identified challenges of this research. The chapter first delves into the sub-questions before answering the main research question. In the final section, pointers are presented to possible future work extending this research.

6.1 Sub-questions

What are the various elements of change provenance related to data?
In Section 2.4 the various elements of provenance of change of data are discussed in the context of data science pipelines. Tracking the evolution of a data set is an important aspect for the reproducibility of experiments, as the data on which the experiment is performed upon directly affects the results of the experiment. Data sets can contain different types of data, each of which require a different approach to properly version them. Properly storing and tracking changes in these data sets introduces additional challenges not present in the versioning of source code as they are inherently different.

What tools exist that support tracking provenance of change in data sets?
Section 2.4.1 describes the strengths and weaknesses of existing tools which enable tracking the changes and the origin of these changes in a data set. The different tools approach the challenges of versioning data sets in different ways. Even though these tools have their merits, in general they lack the ability to handle large data sets.

How can an (adaptive) data science pipeline framework be designed, which is able to respond correctly to the different changes that can occur?
Chapters 3 through 5 describe this process, by analyzing the requirements, designing, and subsequently implementing a framework. It covers both tracking of data as well as pipeline deployments. It allows experiments to be reproduced by defining a specification format, as well as deploying pipeline artifacts based on these specifications. The implementation provides handles for extensions towards more complex topics such as dynamic updates and data visualization both by design and by implementation.

Which provenance elements can be tracked prior to the execution of a pipeline?
Before a pipeline is executed, the specific configuration of the pipeline can be tracked: the versions of the transformation steps, the various parameters and the version of the

input data set. Running this pipeline configuration with the same versions should produce a pipeline artifact which produces similar results after each execution.

Which elements of provenance, that can be tracked, are specific to a pipeline run execution?
The evolution of the data throughout the pipeline can be tracked. Each step in the pipeline performs some computation on the data, transforming input data into output data. The data between each of these steps can be tracked and stored. Additionally, the lineage of this data can be tracked: the output of one transformation step can be connected to the input data which produced it. In this way, the results of a experiment can be retraced throughout the pipeline back to the input data set. This lineage information is stored together with the (intermediate-) results of the pipeline run execution.

6.2 Main Research Question

How to track the evolution of data in (data science) pipelines in order to enable reproducibility and provenance of the results?

Keeping track of which version of a data set is used for an experiment is one of the key aspects of enabling reproducibility and provenance of results. Experiments are reproducible as long as the versions of input data and configuration of the experiment are tracked properly. Another aspect of enabling provenance is keeping track of the lineage of data: what data is used for a specific computation in order to produce a result. Results can be traced to its origin by tracking this lineage information. Existing tools and products were evaluated revealing the need for a tool supporting the aforementioned aspects in data science pipelines. In this research, a design and a proof-of-concept implementation are introduced, which attempts to fulfill this need. The framework allows users to version their data sets and define their experiments as data processing pipelines. The pipelines can then be deployed using a version of the data set as an input, tracking the lineage of data throughout the experiment. This research focuses on the data provenance aspects of the designed framework. To get a more complete view of the pipeline provenance aspects of the designed framework, the other thesis needs to be considered[2]. Two use-cases from the domain were implemented and evaluated to validate the framework. These use cases show that the framework, given the user properly uses the provided abstractions, assists the user in tracking the provenance of change of their data science pipelines. This information can then be used enable reproducible experiments, and also provide insight into how changes in the pipeline affect the rest of the experiment.

6.3 Future Work

The future work topics are split into two types; extensions and features that could be explored for Iterum, and more general topics for reproducibility and tracking of provenance of change.

6.3.1 Research Extensions

Exploring the applications of Iterum and the architecture outside of academic research
This research focuses on reproducing results and provenance tracking specifically for academic research. The features that Iterum and the overarching architecture describe are not only useful in these settings, but for software development in general. Companies

dealing with data processing and pipelines will also benefit from being able to explain how their results came to be. Researching which of the features and how and where these can be applied outside the context of academic research will expose more, possibly unforeseen, challenges.

Re-implementing the architecture using a more mature pipeline deployment tool
This research shows that this architecture is able to track provenance data and that this level of provenance tracking is enough to successfully explain and reproduce results. Building a proper, complete, fault-tolerant pipeline deployment tool is a challenging task in its own right. Existing projects such as Kubeflow (Pipelines) or Flyte already attempt to solve this challenge. Another implementation of the architecture could use these tools and focus mostly on the provenance tracking aspects, rather than pipeline deployment, provided the requirements can be met (e.g. possibility to reasonably perform local runs).

6.3.2 Iterum Specific Extensions

This section is not meant to list a set of future features for Iterum, but to define some general topics that were either left unexplored or omitted. The description of such a future feature set is instead left to the documentation of the code.

Making provenance information produced by Iterum insightful to a user
Gathering information and properly conveying that information are different tasks. Iterum currently focuses on gathering and tracking all of the provenance information, but the ability to properly convey all of the tracked information to the user is yet to be implemented. For example, visually correlating changes in experiments to results would be a more intuitive way for users to understand their experiments rather than simply presenting the information using a command line interface. Researching and developing a proper way to display the provenance information is an important part of furthering the accessibility aspect of Iterum. Appendix X shows some mock-ups of how a graphical user interface for Iterum could look like.

Investigating impact of- and recovery from process failure
Iterum is evaluated in sunny day scenarios. Since the experiments take at most a few hours, few random failures could occur (e.g. network errors, disk failure). If failure was detected, either randomly or due to bugs, the experiments were run again. As experiments grow bigger, executing multiple times becomes unfeasible and more fault tolerance is needed. The design has already taken this into account by not acknowledging messages before completion for example. However, many more failure modes exist, such as failed message delivery, request overload, etc. It is important that their repercussions are thoroughly investigated and (more) measures are taken against them.

Dynamic updates and long-lived pipelines
Iterum focuses on pipelines that can be run till completion. Dynamic updates of running pipelines as well as long-lived pipelines that may process data as it becomes available are possible extensions. A few options for adding dynamic updates are discussed in 5.3.2, and long-lived pipelines can be (partially) realized by utilizing a different type of fragmenter. Undoubtedly, these topics are more complex than they seem at first sight and so, research into these topics is important.

Academic References

[1] A. Alsahaf et al. “A framework for feature selection through boosting”. under evaluation.

[2] E. Apperloo. “Provide provenance and reproducibility of pipeline executions by tracking provenance of pipeline definitions and their code”. *The other thesis that resulted from this research.*

[3] Marcos D Assunção et al. “Big Data computing and clouds: Trends and future directions”. In: *Journal of Parallel and Distributed Computing* 79 (2015), pp. 3–15.

[4] Ekaba Bisong. “Kubeflow and Kubeflow Pipelines”. In: *Building Machine Learning and Deep Learning Models on Google Cloud Platform*. Springer, 2019, pp. 671–685.

[5] David M Blei, Andrew Y Ng, and Michael I Jordan. “Latent dirichlet allocation”. In: *Journal of machine Learning research* 3.Jan (2003), pp. 993–1022.

[6] Brendan Burns et al. “Borg, omega, and kubernetes”. In: *Queue* 14.1 (2016), pp. 70–93.

[7] Alejandro Cartas et al. “Batch-based activity recognition from egocentric photo-streams revisited”. In: *Pattern Analysis and Applications* 21.4 (2018), pp. 953–965.

[8] François Chollet. “Xception: Deep learning with depthwise separable convolutions”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1251–1258.

[9] Steven N Goodman, Daniele Fanelli, and John PA Ioannidis. “What does research reproducibility mean?” In: *Science translational medicine* 8.341 (2016), 341ps12–341ps12.

[10] Hugo Hiden et al. “e-Science Central”. In: ().

[11] Eamonn J Keogh and Michael J Pazzani. “Derivative dynamic time warping”. In: *Proceedings of the 2001 SIAM international conference on data mining*. SIAM. 2001, pp. 1–11.

[12] Mariam Kiran et al. “Lambda architecture for cost-effective batch and speed big data processing”. In: *2015 IEEE International Conference on Big Data (Big Data)*. IEEE. 2015, pp. 2785–2792.

[13] Jay Kreps, Neha Narkhede, Jun Rao, et al. “Kafka: A distributed messaging system for log processing”. In: *Proceedings of the NetDB*. Vol. 11. 2011, pp. 1–7.

[14] Jimmy Lin. “The lambda and the kappa”. In: *IEEE Internet Computing* 5 (2017), pp. 60–66.

[15] Justice Opara-Martins, Reza Sahandi, and Feng Tian. “Critical review of vendor lock-in and its impact on adoption of cloud computing”. In: *International Conference on Information Society (i-Society 2014)*. IEEE. 2014, pp. 92–97.

[16] Joseph Redmon and Ali Farhadi. “Yolov3: An incremental improvement”. In: *arXiv preprint arXiv:1804.02767* (2018).

[17] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).

[18] Estefania Talavera et al. “Topic modelling for routine discovery from egocentric photo-streams”. In: *Pattern Recognition* (2020), p. 107330.

Non-Academic References

[19] Stuart Buck. *Solving reproducibility*. 2015.

[20] Sandeep Dinesh. *Kubernetes best practices: Resource requests and limits*. May 2018. URL: https://cloud.google.com/blog/products/gcp/kubernetes-best-practices-resource-requests-and-limits.

[21] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[22] Stephen Ewen, Fabian Hueske, and Xiaowei Jiang. *Batch as a Special Case of Streaming and Alibaba's contribution of Blink*. Feb. 2019. URL: https://flink.apache.org/news/2019/02/13/unified-batch-streaming-blink.html.

[23] Flexera. *Flexera 2020 State of the Cloud Report*. URL: https://info.flexera.com/SLO-CM-REPORT-State-of-the-Cloud-2020.

Appendix A

Overlap

As discussed in Section 1.2.2, there are two theses which write about the same research, focusing on different perspectives. Since both theses originate from the same research, and introduce the design and implementation of the same framework, there is some overlap between these theses.
The overlapping sections are:

- Section 1.1
- Section 1.3
- Sections 2.1-2.3
- Section 2.6
- Chapter 3
- Chapter 4
- Sections 5.1-5.2
- Section 5.3.3-5.5
- Section 6.3
- Appendices

Appendix B

Code snippets

```
ts_in = TransformationStepInput()
ts_out = TransformationStepOutput()

# Listen for fragments
for fragment in ts_in.consumer():

    # Empty fragment means stop
    if fragment == None:
        print(f"Got kill message, finishing up...")
        ts_out.produce_done()
        ts_out.close()
        break

    # Process fragment
    result = process(fragment)

    # Produce output fragment,
    # linking the old fragment to the new
    out_fragment = LocalFragmentDesc(
        files=result.files,
        predecessors=[fragment.metadata.id])

    # Publish new fragment
    ts_out.produce(out_fragment)

    # Let Iterum know that the fragment is processed
    ts_out.done_with(fragment)
```

Listing B.1: Example transformation step in Python using Pyterum

Appendix C

Mock-ups for a GUI

Iterum currently focuses on tracking the different types of information necessary for the scientists to gain insight into their experiment. Chapters 4 and 5 shows that this information is tracked; however, it can only be shown to the user in a text-based format via the command line interface. Interaction with a system using a CLI is not something which is generally considered to be user-friendly, especially for people outside the realm of computing science. Neither is the data such as lineage trees suited for textual representation.

For this reason, Iterum would benefit immensely from the addition of a graphical user interface (GUI), which is able to display the produced information in a more intuitive manner. The actual implementation of this GUI is outside the scope of this research, but in this appendix, some mock-up designs are shown which show what a GUI for Iterum could look like.

C.1 Pipeline builder

Figure C.1 shows a drag-and-drop tool where scientists can create pipeline configurations for their experiments. Using their own transformations, or by sharing them enables researchers to collaborate more easily. Versions need to be picked specifically in order to guarantee provenance over the configuration.

C.2 Pipeline deployment

Figure C.2 shows that previously created pipelines can be deployed by selecting a specific data set version to be used as input for the experiment. By selecting the transformations, specific parameters can be set for those steps in order to further tweak the behaviour of the pipeline.

C.3 Previous pipeline executions

Figure C.3 shows which pipelines have been executed in the past, showing details on which data set version was used as an input, when the execution started and finished, and the rest of the pipeline configuration associated with this pipeline execution. It shows an overview of all the available provenance information and can be linked to the lineage trees, intermediate results and original pipeline configuration and data sets in order to provide full insight into the versions that were used.

C.4 Experiment analysis

Figure C.4 shows a view where the user can correlate the different input parameters with different output metrics in order to understand how changes in parameters affect the results. This would be more of an extended feature, allowing data scientists to perform analysis on their results as well. These features have not been covered in this research, by would be a natural extension of such a platform. It allows a user to gain insight into correlations between parameters and across multiple deployments of the same pipeline, only with different parameters. This yields a form of sensitivity analysis and parameter tuning.

C.5 Fragment lineage analysis

Figure C.5 shows a view where a lineage tree is constructed from the lineage data captured by the provenance tracker of Iterum. In this view, the users can scroll through the input data, intermediate results and output data, seeing how the data changes between each transformation of the pipeline. By selecting any one file or fragment the user can instantly gain insight into how this has evolved throughout a pipeline. This process could also be applied to analyze lineage and history between data set versions, rather than how data is transformed by a pipeline.

Figure C.1: A GUI concept of what a pipeline builder view might look like

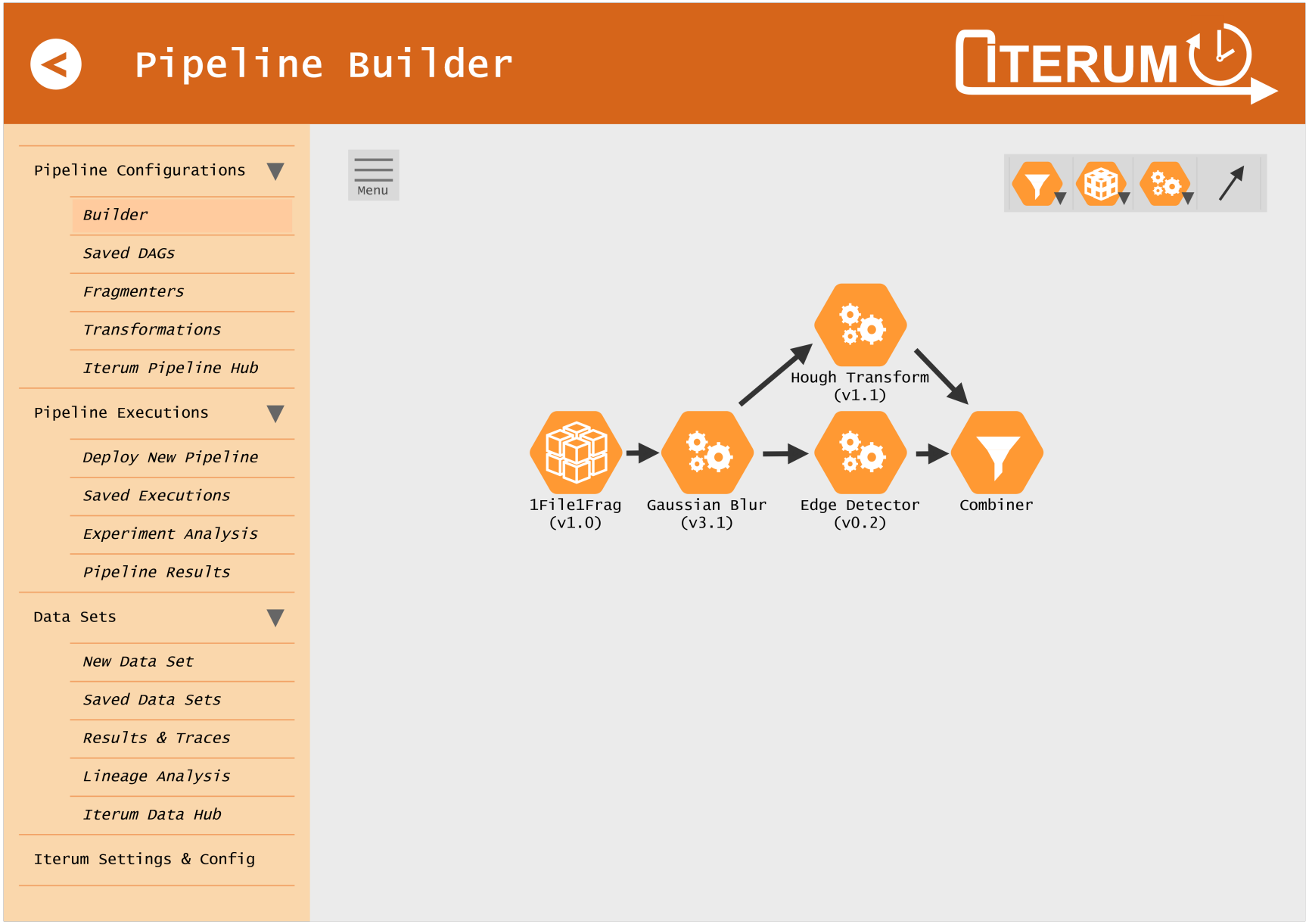


Figure C.2: A GUI concept of what a deploy new pipelines view might look like

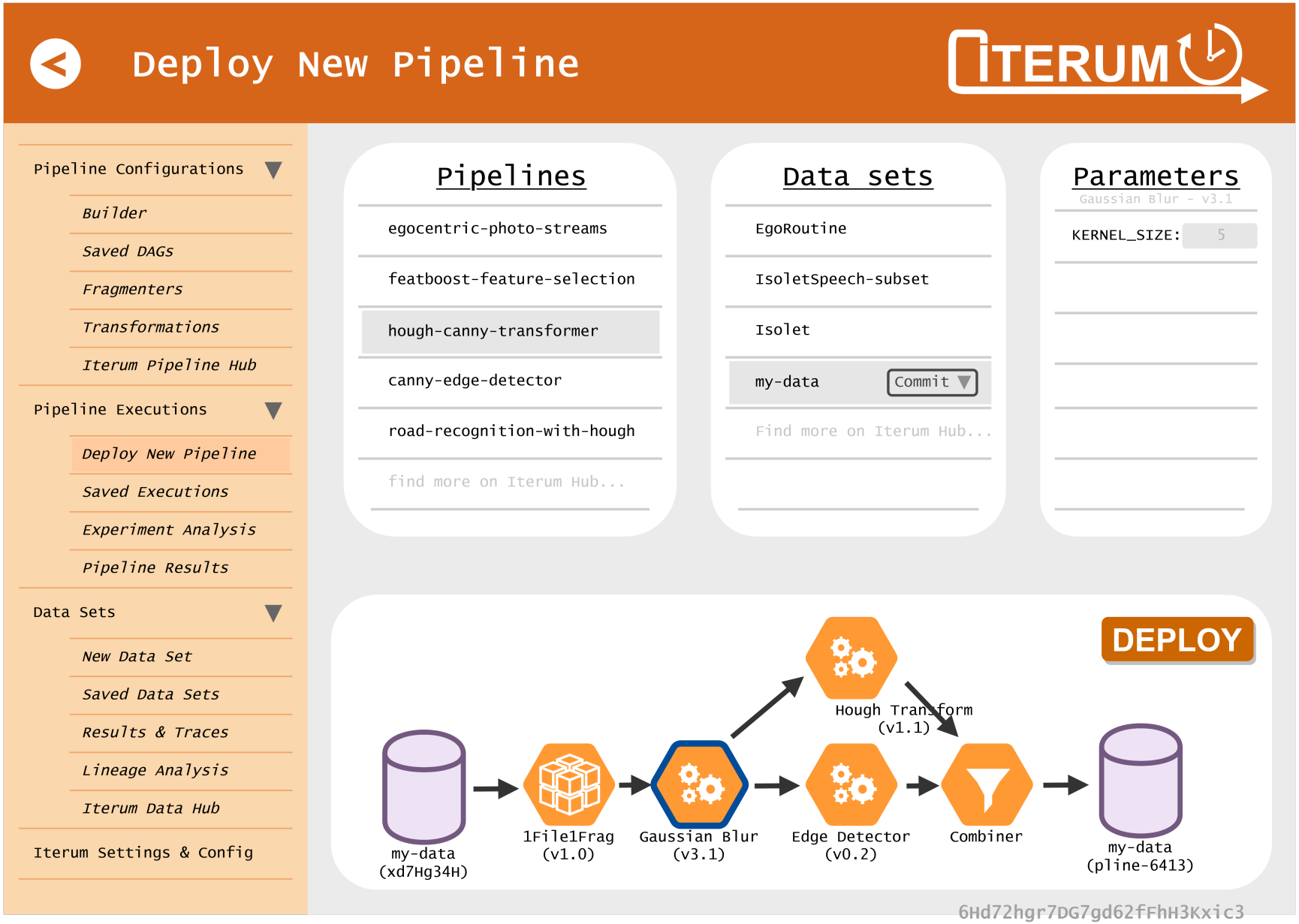


Figure C.3: A GUI concept of what a pipeline execution inspection view might look like

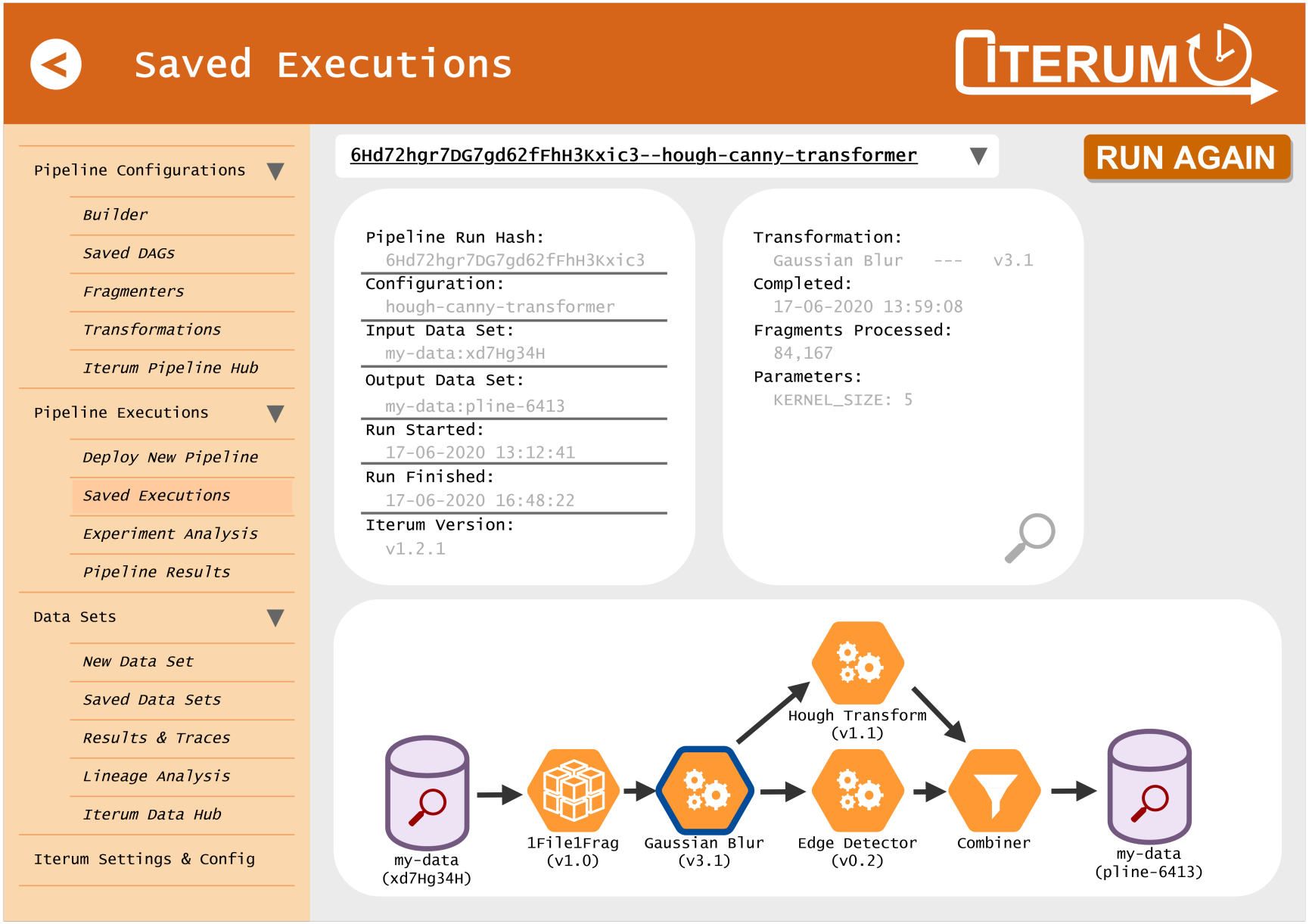


Figure C.4: A GUI concept of what an experiment analysis view might look like

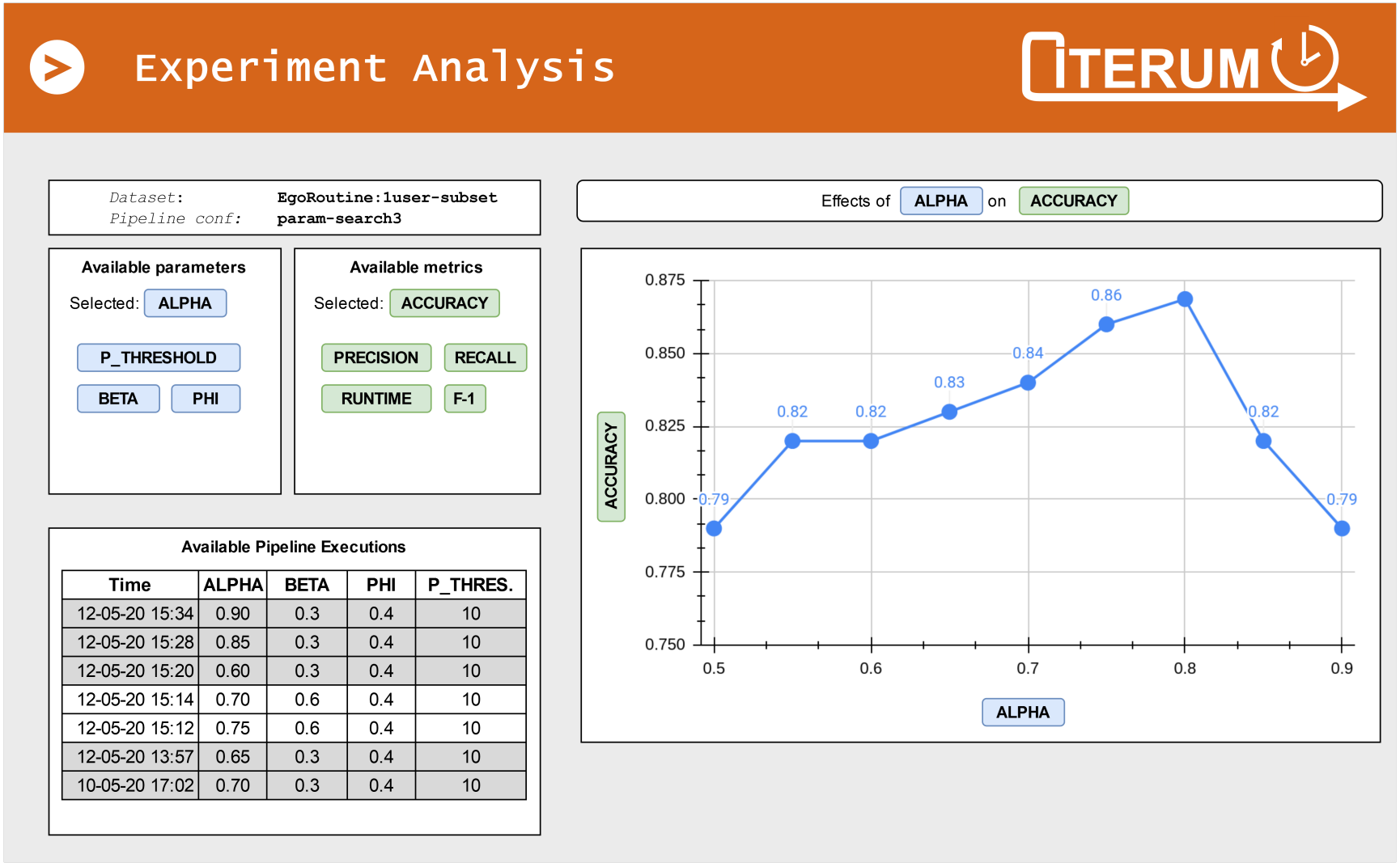


Figure C.5: A GUI concept of what a lineage analysis view might look like