

# The Mezuri Data Provenance Management Platform

*Dibyoy Majumdar*



Electrical Engineering and Computer Sciences  
University of California at Berkeley

Technical Report No. UCB/EECS-2017-131

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-131.html>

July 24, 2017

Copyright © 2017, by the author(s).  
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

# **The Mezuri Data Provenance Management Platform**

by

Dibyoo Majumdar

A thesis submitted in partial satisfaction of the  
requirements for the degree of  
Master of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Eric Brewer, Chair  
Professor Joseph M. Hellerstein

Spring 2017

---

# The Mezuri Data Provenance Management Platform

by Dibyo Majumdar

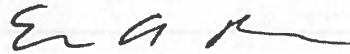
---

## Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences,  
University of California at Berkeley, in partial satisfaction of the requirements for the  
degree of **Master of Science, Plan II**.

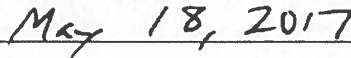
Approval for the Report and Comprehensive Examination:

### Committee:



---

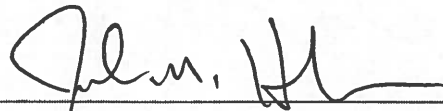
Professor Eric Brewer  
Research Advisor



---

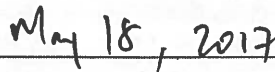
(Date)

\*\*\*\*\*



---

Professor Joseph Hellerstein  
Second Reader



---

(Date)

# The Mezuri Data Provenance Management Platform

Copyright 2017  
by  
Diby Majumdar

## Abstract

The Mezuri Data Provenance Management Platform

by

Diby Majumdar

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Eric Brewer, Chair

Data plays a crucial role in society today. With the cost of collecting, storing and processing data decreasing, more and more of it is getting collected and fed into complex analysis tools to obtain actionable results and insights. These are in turn being used to drive decisions that affect the lives of countless people in good ways and bad. It is imperative that data scientists properly record the provenance of the results they publish ie. they record the original sources of data and the exact sequence of operations performed on those sources to get to the published result. Doing so ensures that results are properly contextualized, and, more importantly, that they can be verified by other scientists. It also fosters collaboration, and leads to the standardization of common data operations and data transfer formats.

Unfortunately, this practice is not the norm in many scientific fields. We contend that this is the case because the tools available today for recording provenance information are inadequate. We presented a set of tools and systems for recording and publishing data provenance information to fill the void. These are built on top of the Git Version Control System and are geared towards data scientists of all research fields publishing the results of their research. We call this the Mezuri Data Provenance Management Platform, or Mezuri Provenance for short. Researchers can use these tools to annotate their existing data processing tools and workflows with provenance information. They can then publish this information potentially along with the actual implementation on our public registry.

To Maa and Baba

Without your steadfast love and support, I would not be here

# Contents

<b>Contents</b>	<b>ii</b>
<b>List of Figures</b>	<b>iv</b>
<b>List of Tables</b>	<b>v</b>
<b>List of Code Examples</b>	<b>vi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 The Problem . . . . .	2
1.2 Related Works . . . . .	2
1.3 Research articles management parallel: arXiv e-print archive . . . . .	4
1.4 Goals for the Platform . . . . .	5
1.5 Report Organization . . . . .	6
<b>2 Concepts and Design</b>	<b>8</b>
2.1 Inputs Outputs Parameters (IOP) . . . . .	8
2.2 Data Components . . . . .	9
2.3 Version Control . . . . .	10
2.4 Pipelines . . . . .	11
2.5 Publishing . . . . .	12
<b>3 Implementation</b>	<b>13</b>
3.1 Annotation Library . . . . .	14
3.2 Command Line Interface . . . . .	18
3.3 Registry . . . . .	20
3.4 Registry Web Frontend . . . . .	22
<b>4 Case Study: Cook-stove Temperature Time Series Analysis</b>	<b>25</b>
4.1 Data Pipelines . . . . .	26
4.2 Discussion . . . . .	29



<b>5</b>	<b>Future Work</b>	<b>32</b>
5.1	Ecosystem improvements . . . . .	32
5.2	Publicizing our platform . . . . .	33
<b>6</b>	<b>Conclusion</b>	<b>34</b>
6.1	Contributions . . . . .	34
6.2	The Big Picture . . . . .	35
<b>A</b>	<b>Git Object Hash Calculation</b>	<b>37</b>
A.1	git commit . . . . .	37
A.2	git tag . . . . .	38
<b>B</b>	<b>python Pipeline Definition DSL</b>	<b>39</b>
B.1	Methods, Parameters and Inputs validation . . . . .	40
B.2	Pipeline Step validation . . . . .	41
B.3	Version Hash Calculation . . . . .	41
	<b>Bibliography</b>	<b>44</b>

# List of Figures

2.1	Pipeline as Component . . . . .	12
3.1	The Architecture of the Mezuri Data Provenance Management Platform <i>All objects placed in the central section represent actions performed by the user. Grey arrows represent library dependencies. Dashed green arrows represent file reads and writes. Dotted blue arrows represent network communication over HTTP.</i> . . . . .	13
3.2	Screenshot of web interface for the <code>TimeSeriesInterface</code> (Listing 1) . . . . .	23

# List of Tables

3.1 Registry API Endpoints for each component type and pipeline . . . . .	21
---	----

# List of Code Examples

1	<code>TimeSeriesInterface</code> representing a time-series, <i>i.e.</i> a series of values obtained over successive times. The interface is annotated with an <code>Input</code> of a sequence of <code>times</code> and corresponding <code>values</code> . <i>Interface annotation are expected to be made on the <code>__init__</code> method, and the class definition must be assigned to the <code>__mezuri_interface__</code> variable.</i> . . . . .	15
2	<code>CookStovePowerTimeSeries</code> representing a time-series collected from a cook-stove. Here, the source is annotated with an <code>Output</code> of the <code>TimeSeriesInterface</code> (Listing 1). <i>Multiple methods can provide source outputs. This capability would be useful when there are multiple files for a data source, or in the case of a database to which multiple queries could be made. The Source can also define an <code>__init__</code> method with <code>Parameter</code> annotations. The class definition must be assigned to the <code>__mezuri_source__</code> variable.</i> . . . . .	16
3	<code>MeanOperator</code> representing a calculation of the mean for either a simple sequence of numbers or a time-series. Here, both <code>Input</code> and <code>Output</code> annotations are made. <i>As with Sources, multiple methods can be added, one for each combination of inputs and outputs that the Operator supports. The Operator can also define an <code>__init__</code> method with <code>Parameter</code> annotations. The class definition must be assigned to the <code>__mezuri_operator__</code> variable.</i> . . . . .	17
4	Sample Pipeline using sample components defined earlier . . . . .	18
5	Complete setup for the <code>MeanOperator</code> (Listing 3), from initialization to publication to local registry. . . . .	20
6	<code>mezuri_specifications.json</code> for the <code>MeanOperator</code> (Listing 3). . . . .	24
7	Cook-stove Usage Generation Pipeline definition . . . . .	26
8	Cook-stove Usage Aggregation Pipeline definition . . . . .	27
9	Cook-stove Adoption Determination Pipeline definition . . . . .	28
10	Cook-stove Adoption Determination Pipeline definition . . . . .	31
11	<code>OperatorProxy</code> definition excerpt . . . . .	39
12	<code>AbstractComponentProxy</code> definition excerpt . . . . .	40
13	<code>PipelineStepContext</code> definition excerpt . . . . .	42
14	<code>PipelineStep</code> definition excerpt . . . . .	43

## Acknowledgments

The Mezuri Data Provenance Management Platform, presented in this report, would not have been possible without the contribution of some truly remarkable individuals.

First and foremost, I would like to thank my advisor, Eric Brewer. Professor Brewer was always available to discuss the project despite his busy schedule.

In the Technology and Infrastructure for Emerging Regions (TIER) group at Berkeley, I found home. Special thanks to Jordan - our discussions on building graphical tools for creating data pipelines deeply informed the design of the platform.

I would also like to thank my second reader Joe Hellerstein. Professor Hellerstein's feedback on the first draft of this report was valuable in figuring out and refining the use cases of the platform.

Last but not least, I would like to thank my parents, Brinda and Debasish Majumdar. Their love and support kept me motivated and focused on completing the project.

# Chapter 1

## Introduction

“Knowledge is power” goes the famous adage. In today’s society, data is being used to cure diseases, tackle crime, provide better services to people and predict the weather. With the cost of collecting, storing and processing data decreasing rapidly, the amount of data available is increasing exponentially; 2.5 exabytes of data is being created every day[16] as home, among other things, deploy more physical sensors and generate more data on the internet.

This increase in data produced is accompanied by an increase in the number of decisions that are made based on insights from this data. However, most raw data is not directly actionable. In general, data passes through the following stages:

1. **Data Collection:** Data comes from a number of different sources. On one hand, it may be survey data collected through paper questionnaires and entered into a Microsoft Excel spreadsheet. On the other hand, it might be from sensors that are periodically and automatically sending temperature or readings over the network to a database.
2. **Data Analysis:** In this stage, the data might go through a number of transformations. Some of it might be automated, for instance, cleaning up raw sensor readings. Others might be at the hands of human data scientists running various algorithms on the data, continuously bringing new tools to bear, and tweaking parameters on existing tools.
3. **Data Publication:** If the data scientist obtains some results that they consider noteworthy, they may choose to publish them. They might take the time to clean up and organize their analysis code, and publish their raw data alongside the code. Or, given that their previous results have opened up new avenues of research, and that they are eager to explore those, they might simply publish a dump of their code or nothing at all.
4. **Data Usage:** We consider this the most important phase. In this phase, the published data is looked at and discussed by the researcher’s peers and used to inform their

research. Some of them might try to validate the results. The data might be used by a Congressman to inform their Bill on Federal Funding for the next ten years, and the data may be used to support that Bill in Congress. Or it might be used by the local City Municipality to decide where they should be adding new bus stops, or where they should be conducting maintenance on railway tracks.

At the same time, the author of the original results might be performing another iteration of surveys, or correcting their methodology. They might publish a new version of their results and take down the data from their old results.

## 1.1 The Problem

A large number of these data-driven decisions are highly consequential, affecting the lives of countless people, in good ways and bad. It is the responsibility of the data scientists producing certain results to ensure that their results are properly contextualized, that each step in the process that led to a given result is properly documented and that there are ways for other people to verify their results. Only then can those results be considered fit for use in making high-stakes decisions.

But quite often, scientists fail to do these things well. We believe this happens in part because the tools available currently for recording and publishing data provenance are inadequate. To demonstrate this, we discuss currently available tools and their shortcomings in this next section.

## 1.2 Related Works

### Code Provenance Management

A number of online services such as GitHub[11] and BitBucket[5] allow users to share their code in a version-controlled manner. However, these services are tailored to serve the needs of Software Developers, not Data Scientists. *They work perfectly well for the purposes of managing data processing tools, but many of their core features, such as line-based diffs, Pull Requests and Code Reviews do not make sense for the purposes of managing datasets.*

### Dataset Provenance Management

Significant work has been done around managing dataset provenance. Cloudera Navigator[7] and Apache Atlas[1] are both end-to-end data governance solutions for Hadoop that help with data discovery and lineage exploration, capabilities that we are interested in. LinkedIn's WhereHows[15] and Google Goods[13] are more general platforms that provide provenance management for all kinds of datasets. All of these are however passive solutions that continue recording provenance information in the background without user intervention. We advocate

the exact opposite approach, wherein users have explicit control of what constitutes a new version of their dataset, or a derived version.

The MIT DataHub project[8] does just this. It is the GitHub equivalent for dataset repositories. It allows for storage and smart diff-based versioning of datasets[18]. This means however that it has exactly the reverse of the problem that Github has for our purposes - *DataHub's features do not make sense for the purposes of managing code repositories for the processing tools and workflows that are run on these datasets.*

Essentially, we are interested in a broader definition of provenance, wherein provenance information is collected not only for datasets, but also the processing tools that are run on these datasets.

## Workflow Provenance Management

*In addition to managing datasets and data processing tools, we require a solution that can manage data processing workflows made up of these components.*

A number of different approaches have been taken for workflow provenance management. PASS (Provenance-Aware Storage Systems) makes changes at the operating system kernel level to collect and store provenance information alongside files on the filesystem[24]. This information is query-able. Ground is described as an open-source data context system that aims to capture the full context on all data and exposes them through a queryable interface[14]. ProvDB is a similar system “intended for the unified management of all kinds of metadata about collaborative data science workflows that get generated during a project lifecycle”[21]. ProvDB has developed a number of metadata ingestors that run when commands are run in a shell, including version control information from `git` and makes all of its metadata queryable. Ground on the other hand only implements the storage of metadata, and relies on plugins for ingesting metadata on one side and presenting metadata on the other. This features makes Ground more powerful than ProvDB as there is the potential to add powerful extensions including a prediction plugin for suggesting possible datasets and workflows to collaborators.

These systems are highly effective in achieving their goals, but have different use cases than ours. Firstly, we are looking for tools that allow for the sharing of not just workflows, but also individual datasets and data processing tools that constitute those workflows. These separate concepts deserve tools that are targetted specifically to them. Secondly, we are looking for tools that would allow researchers to publish their work widely, not just with collaborators working in the same lab or the same company. For this purpose, we argue that all published information must be explicitly provided by users, and not collected through automated ingestors.



## Workflow Management, Execution and Publication

Another area we survey is workflow management and execution systems. From personal experience, one of the reasons we do not share our data workflows is that they are messy and badly documented. This is often because they were the results of a large number of experiments that we performed, and we were more interested in running experiments than designing a good code architecture. As a result, our experiments are hard to understand for someone who is just looking at the code. Recognizing this, we are less likely to release our code in the first place. *We would like tools that would assist us in keeping our experimentation code organized (preferably at very low overhead).*

Chimera presents a DSL for defining and executing workflows on top of SQL databases[10]. It caches intermediate results for faster re-executions. Vistrails[30], Taverna[2] and Kepler[29] are platforms where users can create and execute their own data workflow on a graphical interface using data processing modules created by developers. The platforms provide provenance recording for user workflow creation on their GUI. Vistrails allows publication of workflows online. However, neither of these platforms do not have the concept of version controlling for individual modules.

Outside the data management field, we have package managers such as the Node Package Manager (NPM)[22], Ruby Gems[28] and Python pip[26]. All of these have the concept of versions for the packages they install. These packages themselves are composable ie. a package might be dependent on other packages which are themselves dependent on others and so on. NPM and Ruby Gems also have online webpages for every package with more information about the package and its different versions that aid with discoverability. These are all features that we want in our tools. However, the similarities end there. We see the datasets and data processing tools that are the equivalent of packages in our use case to be more rigid libraries with fully-specified input and output points that have to be connected directly with other compatible datasets and data processing tools to create an executable program, a workflow. So, while package managers are interested in installing software for execution, our tools are more concerned about ensuring that the different datasets and data processing tools used in a workflow have compatible connections with each other. Enabling actual execution of these workflows is a secondary feature for us.

### 1.3 Research articles management parallel: arXiv e-print archive

In the last section, we looked at the shortcomings in currently available tools. Before explicitly discussing our goals, we take a look at the arXiv e-print archive, which we believe tries to solve a similar problem for research articles, namely the archival of research articles. We note that researchers in many different fields know about the archive and its intended use, and publish on the archive. Some notable features of the service that we believe contributed

to its success are as follows[3]:

1. The service is self-service. Researchers upload their own work.
2. The service is open to everyone and free, both for reads and publications.
3. The service is focused on maintaining the perpetual availability of submissions. It restricts changes that can be made after submissions are announced. Researchers can publish new versions of submissions. But all versions are displayed.
4. The service has a multi-stage system for managing submissions. Drafts can be uploaded at any time, but authors choose when they want to publish their submission.
5. The service provides research articles in different formats (not just PDF). To enable this and other features, it requires that submissions are made in LaTeX format.
6. The service requires certain metadata on all submissions and use these to provide additional features such as exploration of citations, explorations of other works by the same authors and categorization of submissions by research field.

## 1.4 Goals for the Platform

In a nutshell, we are trying to create a great set of tools for managing the provenance of datasets and data processing tools and workflows composed of them, that play a similar role in research data publication space that arXiv plays in the research article publication space. For doing this, we can not consider just the Data Usage stage, but must look at how provenance management can be incorporated through every stage of the data lifecycle.

### 1. Record and manage data provenance information

This is our main goal. We are trying to build a system that keeps track of the series of steps that were taken to reach a certain result, *e.g.*, the data workflow employed. With this information, other researchers can come in and reproduce results, while the original researcher can continue building on it, and publish new versions.

Additionally, we would like to track changes within individual steps. This would be useful when the steps themselves can evolve, such as a computer vision algorithm (edge detection), which might get algorithm tweaks, or start expecting different inputs.

### 2. Create tools that are easy to learn and easy to use

Our system must be easily usable by researchers of all backgrounds, otherwise it has failed its purpose. For users who would like to explore the provenance of some published result,

we only assume that they know how to follow a URL to a website and navigate within the website.

For users who want to record the provenance of their data analyses, we only assume the level of technical know-how needed to perform those analyses.

### **3. Facilitate Collaboration**

Collaboration is closely connected with our other goals, but we state this separately to emphasize its importance in our platform. The ultimate aim of maintaining data provenance is to make it easy for researchers to ensure the veracity of previous work, and then come together and build on it!

### **4. Contribute to the standardization of data processing primitives**

This is related to our previous goal of collaboration. Having data in different formats, and having different implementations for the same data processing step makes it harder for people to work together as it introduces a tax of conversion among different data formats and reconciliation of results from different implementations.

### **5. Change the culture around managing data provenance**

This is our long-term goal. Ultimately, we are creating these tools to facilitate researchers in thinking about data provenance management. We will be happy if competing tools are created because we will then know that we have accomplished our goal. Our tools will of course be completely open-source and we welcome community contributions.

## **1.5 Report Organization**

The rest of this report is organized as follows.

- In Chapter 2, we look at the different concepts that form the base for our platform and discuss their design and use cases at a conceptual level, keeping our project goals in mind.
- In Chapter 3, we present the implementation of our tools. We show how a number of different toy scenarios would be handled on our platform, including the publication of various data components built using our tools. We discuss how a web interface for published components would look like.
- Then in Chapter 4, we walk through a Case Study showcasing the capabilities of our platform. We recreate the data analysis presented on a project that aimed to study

the patterns of stove usage after introduction of an advanced cook-stove in over 200 households spread over 7 villages in Haryana, India[25] as a Data Pipeline on our platform, and discuss the tradeoffs in our approach.

- Then in Chapter 5, we discuss next steps in developing our platform, including our plans for publicizing it.
- Finally in Chapter 6, we conclude our discussion by reiterating our contributions and considering their potential impact.
- We take a technical look at some of the underlying concepts that make our tools work in Appendices A and B.

# Chapter 2

## Concepts and Design

The Mezuri Data Provenance framework was designed keeping in mind our end-users, namely researchers in all fields, and their use cases. As mentioned in the introduction, our goal is to allow our users to easily keep track of their data and foster collaboration throughout the data lifecycle. To this end, we have designed and built a number of components for the different stages of the data lifecycle.

In this chapter, we give a conceptual description of the different pieces and their design. In the next chapter, we discuss how they were implemented on the platform.

### 2.1 Inputs Outputs Parameters (IOP)

In any data analysis workflow, different types of data are produced and consumed by different computations. Inputs and Parameters are data fed in to a component while Outputs represent the results of any such computation.

Even though Inputs and Parameters perform similar roles in this respect, we provide both these concepts so that a distinction can be made between the data being processed, and the inputs that control how the data is processed. There is one concrete distinction made between the two: Parameters can have default values (and therefore do not have to be explicitly stated), but Inputs being the subject of a computation must always be explicitly specified.

On the platform, the name, type, and optionally the default value of all Inputs, Outputs and Parameters to a computation are stored alongside the definition of the computation. They are collectively referred to as the Input Output Parameter (IOP) specifications of the computation, and they help in verifying that the computation is being used properly in the larger context of the rest of the workflow, without having to parse the computation definition code. We note that the type of each Input, Output or Parameter can be something simple like a float or an array of numbers, but also arbitrarily complex, such as a record (with filename labels) from a database table.

We note that data types are an important part of this verification, since certain computations only make sense for certain data types. For example, a computation might output Doubles that need to be properly interpreted by the next computation. Or a computation might expect that its inputs always have an `id` attribute which it uses as a primary key in its operation. Or a computation may expect to get as Input an array of values sorted in ascending order.

## 2.2 Data Components

Data Components refer to the primitives of computation on the platform. We have three of them: Sources, Operators and Interfaces.

### Sources

Sources represent entrypoints for data into the data analysis workflow. Conceptually, we think of them as immutable, a read-only file. The data can come from a file, but if the file is modified, it is considered a new version of the data source. Similarly, a query performed on a database can also be considered a data source. However since a database is mutable, the platform first stores the result of running the query on the database in a file and then reads from the file for further processing. The query itself, the name of the database and the timestamp when the query was run are stored alongside the file as metadata. If the data source is used in future, the user is given the option to read from the CSV file, or create a new version of the data source (re-performing the query on the database). Therefore, a data source on our platform is both, conceptually and literally a read-only file or set of files.

By default, we keep these files in CSV-format. We selected the CSV format since its highly human-readable while being easily convertible to and from other formats at the same time.

IOP Specifications: Sources can have Parameters and Outputs. Parameters can be used to tweak the Outputs. For instance, in a data source with multiple files, we may have a `filename` Parameter to choose the output file. For a database, parts of the query might be tweaked using Parameters. There might be multiple combinations of Outputs as well, implemented as different methods on the Source class. We note that it does not make sense to have an Input to a Source.

### Operators

Operators represent transformations performed on one or more Inputs to generate one or more outputs.

IOP Specifications: Operators can have Parameters tweaking their functioning, and many different combinations of Inputs and Outputs (thus allowing them to adapt their functioning

based on the incoming data). In the implementation, these are represented by different methods on the Operator class. Each combination must necessarily have at least one Input and one Output.

## Interfaces

Interfaces represent pipes between operations. They represent a cluster of data types commonly used together. They can be specified as Inputs or Outputs to operators, or as the Output of sources. (They could also technically be used to specify Parameters.)

The use of an interface in IOP specs for operators and sources can be completely replaced by the corresponding struct of data types. However, we hope that interfaces will encourage our users to reuse data structs (often created by other users) while building their operators and adding their data sources. This will reduce the number of different formats for the same type of data, thus making collaboration easier.

IOP Specifications: The data type definition of the interface is recorded as an Input to the Interface.

## Interface Conversion Operators

These are just regular operators that convert between different interfaces. We mention them separately since we anticipate efficient implementations of interface converters being crucial alongside the interfaces themselves. While one of our explicit goals is to contribute to the standardization of data formats for different use cases, it is highly likely that many types of data might have a few competing data formats. It is imperative that we support all the different formats and conversion with them in order to best support all our users.

## 2.3 Version Control

Data components are all version controlled using a semantic versioning scheme[27]. We have designed our version control mechanisms to play nicely with existing version control systems that users might have on their code. Each component is kept in its own git repository. Component versions are managed as tags to the corresponding commits, and therefore do not interfere with user commit information.

Technically, the version is represented by the hash of the `git tag` created. This in turn represents the sequence of commits up to the tagged commit and the authorship information for the tag (the author of the tag and the time when it was created). We discuss why this is the case in Appendix A. Therefore, we are taking into account all the pieces of information that are important to uniquely specify the version.

A new version means different things for different components. For a Source, it might mean an updated execution of a query on a database, or a new set of survey results. For

Operators, it might be a change to the transformation code and/or a change to the IOP specification. For an interface, it would signify a change in the data format.

Semantic versioning affords us a number of benefits. Firstly, its a popular format that many people are already familiar with. Secondly, since version numbers only increase, versions provide a chronological account of a component's progression.

## 2.4 Pipelines

Pipelines represent a directed acyclic graph of operations performed on a set of sources to produce a given result, *i.e.* a data processing pipeline. The nodes of the graph are made up of Sources and Operators, and the edges are data flows between them (potentially some of them as Interface).

Pipelines are also version controlled. A new version would represent a change in the graph structure or a change in the version of any of the components used. The version hash of a pipeline is an adapted version of a Merkle Tree[19], defined as follows:

1. The hash of a component method called as part of a pipeline, is the hash of the name of the method and its arguments.
2. The version hash of a Pipeline Source Step is the XOR of the version hash of the contained Source and the hash of the source method called to obtain the output.
3. The version hash of a Pipeline Step representing a data operation is the XOR of the version hash of the contained Operator, the hash of the operator method called to convert inputs to outputs, the hash of the provided parameters and the version hashes of all Pipeline Steps and Pipeline Sources on which this step is dependent, *i.e.* those nodes that provide inputs to the contained Operator.
4. The version hash of the Pipeline is the version hash of the final Pipeline Step (which gives the result).

As in a Merkle tree, this scheme allows us to find exactly how a pipeline has changed across versions.

### Pipelines as components

We can abstract away a pipeline as a hybrid component (Figure 2.1). Such a component has characteristics of a Source component, since it may have one or more data entrypoints, and characteristics of an Operator component as well, since the data may go through many rounds of transformation.



User can use a pipeline like a normal component, and can therefore build on each other's work.

IOP Specifications: As shown in the Figure, the Output of the derived component is the same as that of the original pipeline ie. the Output of the last step of the pipeline. The pipeline did not have any Inputs (since all leaf nodes were Source nodes), and the derived component does not have any either. Finally, the derived component's Parameters are the aggregate of the Parameters for each component in the original pipeline, and are namespaced by the pipeline step they were in previously.

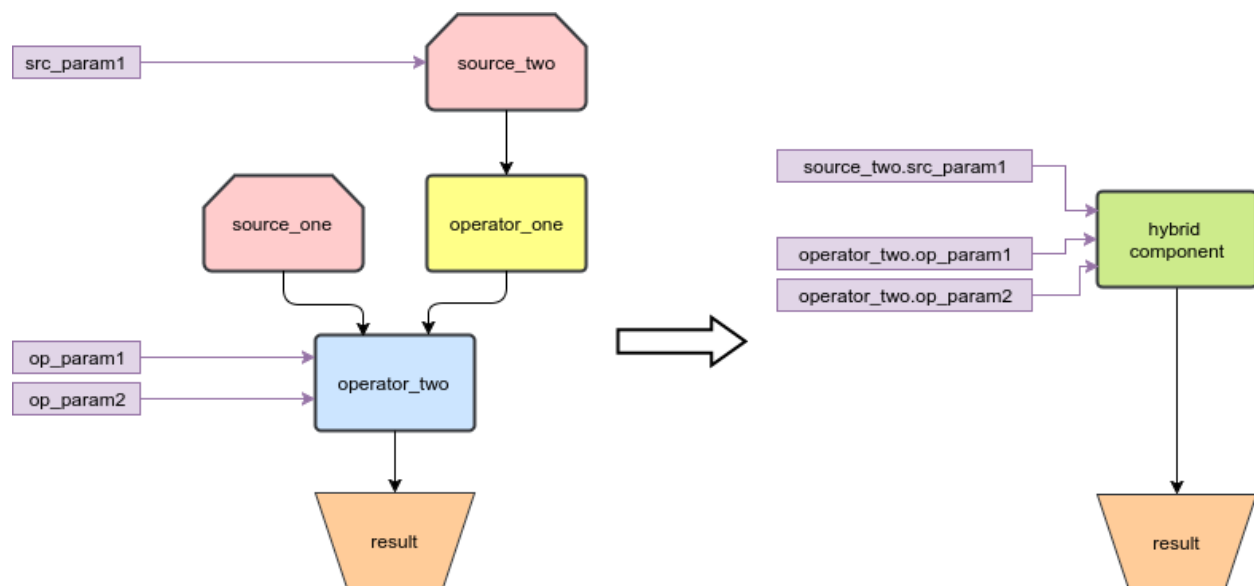


Figure 2.1: Pipeline as Component

## 2.5 Publishing

The final and most important piece of the platform is the publishing of data components and pipelines. Component authors can choose to publish different versions of their component to a web registry.

Other users can then choose to use these components in their own research. Since, components are annotated with their input and output specs, users can choose other components to go with them, or build their own.

Databases can be shared in a version controlled manner. Interfaces can help with standardizing data formats. And good implementations of different operations can mean that researchers can spend time devising new algorithms instead of reimplementing the old.

# Chapter 3

## Implementation

Data Source, Operators, Pipelines and even Data Interfaces are well-known concepts. Implementations of these concepts are not *per se* novel. We focussed our effort on making it easy to integrate these concepts into existing programs.

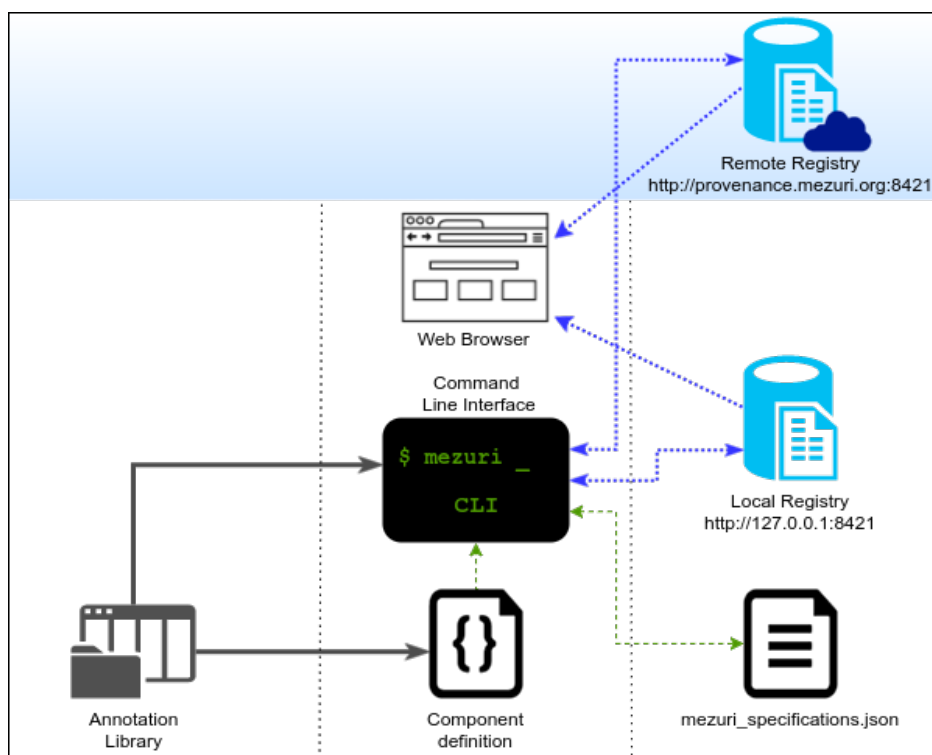


Figure 3.1: The Architecture of the Mezuri Data Provenance Management Platform *All objects placed in the central section represent actions performed by the user. Grey arrows represent library dependencies. Dashed green arrows represent file reads and writes. Dotted blue arrows represent network communication over HTTP.*

There are three pieces to this integration: an annotation library that researchers use to annotate their data components, a registry for hosting these components and a Command Line Interface that researchers can use to prepare and upload their annotated components (Figure 3.1).

Our current implementation is hosted in two repositories on the Mezuri Provenance Github Organization[20]. The `mezuri-provenance` repository contains our core contributions, including the three pieces mentioned above. The `registry-frontend` repository contains an implementation of a sample frontend for the component registry, demonstrating how our tools foster collaboration through the sharing of components.

## 3.1 Annotation Library

The Annotation Library is the only piece that might directly interact with user's code. It is a set of classes that researchers can use to specify the input, output and parameter specifications of their components. Essentially, they are language bindings to our framework.

We have defined Python language bindings currently, and expect to add bindings for other languages in future.

### Data type Primitives

We have defined a set of data type primitives that can be used to create more complex data types for use as Inputs, Outputs, Parameters and Interfaces.

One class of primitives are basic data types. So far, we have defined four of these, namely `Bool`, `Integer`, `Double` and `String`. In future, we could of course augment these with variations such as `int64`, `uint32` and `single`, if necessary. The other class consists of compositional primitives. These are as follows:

- `List<type>`: This is a finite sequence of values of a specified type.
- `Stream<type>`: Like `List`, this is a sequence of values of a specified type, but in this case, the sequence can be infinite. We define this separately from a `List` to allow for distinction between a streaming algorithm that consumes values and produces intermediate output (for example, an operator that produces a cleaned version of a stream of data points flowing in from a sensor), and an aggregation algorithm that needs all the data points to produce an output. The former could accept both a `Stream` and a `List`, while the latter could accept only a `List`. This is the case because a `List` can be converted into a `Stream`, but not vice-versa since a `Stream` has unspecified size.
- `Struct`: As the name suggests, a struct is simply a collection of data types with string names associated with each of them.

```

1 from mezuri_lib.definitions import AbstractInterface
2 from mezuri_lib.declarations import Input
3 import mezuri_lib.types as mezuri_types
4
5
6 class TimeSeriesInterface(AbstractInterface):
7     @Input('time_series', mezuri_types.Stream(mezuri_types.Struct({
8         'time': mezuri_types.Datetime(),
9         'value': mezuri_types.Double()
10    })))
11     def __init__(self, time_series):
12         self.time_series = time_series
13
14 __mezuri_interface__ = TimeSeriesInterface

```

Listing 1: `TimeSeriesInterface` representing a time-series, *i.e.* a series of values obtained over successive times. The interface is annotated with an `Input` of a sequence of `times` and corresponding `values`. *Interface annotation are expected to be made on the `__init__` method, and the class definition must be assigned to the `__mezuri_interface__` variable.*

## Component Definitions

For managing components, we have defined a set of `Abstract[Component]` classes that users subclass for their own components. See Listings 1, 2 and 3 for examples definitions of an Interface, Source and Operator respectively. In these Listings, we also discuss the constraints that the library places on the annotations.

We note here that the component annotations do not interfere with the functioning of the component classes. In fact, they assist in the implementation of the components. The `SourceReader` implementations that the Source components use (for example, `CSVFileReader` and `MySQLDatabaseReader`) are simple working implementations, and the library could easily be extended with readers for other source types. Interfaces and Operators are represented as simple Python classes that can be initialized to get instances of the same. For a researcher working in python, these could be the actual implementations, or wrappers around actual implementations of the components.

The primary goal of these annotations is to provide a simple way for researchers familiar with one language to create such annotations. In fact, researchers only need to know one language since they could quite as easily annotate their R or Java operators with these Python definitions. Conversely, language bindings in other languages are not strictly required for annotation.

However, it is preferred that component implementations or wrappers around implementations are annotated, in order to eliminate drift between implementations and provenance information. Ideally, a new version of the implementation would be accompanied with a new

```

1  from datetime import datetime
2
3  from mezuri_lib.declarations import Output
4  from mezuri_lib.definitions import AbstractSource, CSVFileReader
5  from mezuri_lib.pipelines import InterfaceProxyFactory
6  import mezuri_lib.types as mezuri_types
7
8
9  TimeSeriesInterface = InterfaceProxyFactory('http://registry.mezuri.org',
10                                             name='time-series',
11                                             version='0.0.3')
12
13
14  class CookStovePowerTimeSeries(AbstractSource):
15      @classmethod
16      @Output('time_series', TimeSeriesInterface)
17      def read(cls):
18          return CSVFileReader('0.0.0.214-power-1800.csv', lambda fields: {
19              'time': datetime.strptime(fields['date'], '%Y-%m-%d %H:%M:%S'),
20              'value': fields['value']
21          }).read()
22
23  __mezuri_source__ = CookStovePowerTimeSeries

```

Listing 2: `CookStovePowerTimeSeries` representing a time-series collected from a cookstove. Here, the source is annotated with an `Output` of the `TimeSeriesInterface` (Listing 1). *Multiple methods can provide source outputs. This capability would be useful when there are multiple files for a data source, or in the case of a database to which multiple queries could be made. The Source can also define an `__init__` method with `Parameter` annotations. The class definition must be assigned to the `__mezuri_source__` variable.*

version of the component.

## Component Proxies

In addition to component definition classes, we have component proxy classes. They are used in the definition of pipelines and other components in place of a component definition when the component definition is not available locally. Instead, the proxy class can be locally created from `ProxyFactory`s by supplying the remote publication URI for the component. Listings 2, 3 and 4 show examples of this. They have the same IOP specifications as the corresponding component definition class. For operators, this means that they have the same methods as the definition class, taking identical Inputs and Parameters, and producing identical Outputs. For sources, this means that they have the same read methods as the definition class, producing identical Outputs.

```

1 from mezuri_lib.declarations import Input, Output
2 from mezuri_lib.definitions import AbstractOperator
3 from mezuri_lib.pipelines import InterfaceProxyFactory
4 import mezuri_lib.types as mezuri_types
5
6
7 TimeSeriesInterface = InterfaceProxyFactory('http://registry.mezuri.org',
8                                             name='time-series',
9                                             version='0.0.3')
10
11
12 class MeanOperator(AbstractOperator):
13     @Input('numbers', mezuri_types.List(mezuri_types.Double()))
14     @Output('mean', mezuri_types.Double())
15     def mean(self, numbers):
16         return sum(numbers) / len(numbers)
17
18     @Input('time_series', TimeSeriesInterface)
19     @Output('mean', mezuri_types.Double())
20     def time_series_mean(self, time_series):
21         return sum(
22             point['value'] for point in time_series) / len(time_series)
23
24 __mezuri_operator__ = MeanOperator

```

Listing 3: `MeanOperator` representing a calculation of the mean for either a simple sequence of numbers or a time-series. Here, both `Input` and `Output` annotations are made. *As with Sources, multiple methods can be added, one for each combination of inputs and outputs that the Operator supports. The Operator can also define an `__init__` method with `Parameter` annotations. The class definition must be assigned to the `__mezuri_operator__` variable.*

Behind the scenes, proxy classes fetch the component specifications from the remote publication URI to resolve their own behavior. We explore this behavior in depth in Appendix B.

## Pipeline Definitions

We provide classes for building data pipelines that keep track of provenance information internally. We define pipelines in terms of step definitions and in-edges from other steps using `PipelineStep()`. The final step is wrapped in `Pipeline(last_step)` (Figure 4).

Pipelines use IOP specifications in component definitions and component proxies to provide powerful validation in pipeline steps. Firstly, only component methods that have been annotated with the library are allowed inside pipeline steps. Secondly, each step keeps track of the Outputs from the step. In all steps dependent on this step, those Outputs are available

```

1  from lib.pipelines import (
2      SourceProxyFactory, OperatorProxyFactory, PipelineStep, Pipeline
3  )
4
5  def main():
6      TimeSeriesSource = SourceProxyFactory('http://registry.mezuri.org',
7                                             'cookstove-power-time-series',
8                                             '0.0.4')
9      MeanOperator = OperatorProxyFactory('http://registry.mezuri.org',
10                                         'mean',
11                                         '0.1.1')
12
13     source_step = PipelineStep()
14     with source_step.context():
15         TimeSeriesSource().read()
16
17     mean_step = PipelineStep()
18     with mean_step.context():
19         MeanOperator().time_series_mean(time_series=source_step.output['time_series'])
20
21     pipeline = Pipeline(last_step=mean_step)

```

Listing 4: Sample Pipeline using sample components defined earlier

as Inputs/Parameters. Thirdly, when outputs from previous steps are used as arguments to an operator method, the name and types of all arguments are checked to ensure that they match the Input specifications. A similar check is performed for operator initialization where arguments must match the Parameter specifications.

Validation is completely transparent to the user. It is performed at definition time, so users are immediately notified when they first try to run their pipeline definition code. Additionally, pipeline steps also keep track of the version hash for each step and ultimately the entire pipeline (following procedure defined in Section 2.4). We discuss how this behavior is achieved in Appendix B.

## 3.2 Command Line Interface

We maintain a `mezuri_specification.json` file alongside the language-specific component definition file. This file contains general information about the component such as its name and current version, and IO specifications. This generated file is essentially a summary of the component and saves the system from having to read and interpret the definition file every time it wants to know something about the component.

For pipelines, we maintain an additional `manifest.json` file. This file keeps track of

all the dependencies of the pipeline, *i.e.* the name and version of each component in the pipeline. Once again, this is for convenience.

The Command Line Interface provides a set of four commands for managing each type of component and pipelines.

### **mezuri <component-type>|pipeline init**

This command sets up the component/pipeline. If the current folder is already a part of a git repository, it uses that repository, otherwise it initializes a git repository rooted at the current folder. It prompts the user to supply a number of component properties such as name and description and populates a `mezuri_specifications.json` file with this information.

### **mezuri <component-type>|pipeline generate [-f|--file]**

This command generates the IOP specifications, definition and dependencies sections of the `mezuri_specifications.json` file, by interpreting the annotations in the definition file specified by the `-file` option. If no file is provided, `<component-type>.py/pipeline.py` is assumed.

For pipelines, the IOP specifications contains the specifications for the corresponding operator. Additionally, the command generates the `manifest.json` file.

### **mezuri <component-type>|pipeline commit <version> <commit-message>**

This command creates a new version of the component/pipeline or updates the current latest version. It takes in the version (which must be equal to or bigger than the current version) and a commit message. If there are uncommitted changes in the source tree, it commits these first. It then creates a tag to the latest commit in the format `mezuri/<component-name>/<version>/<update-num>`. The author of the tag is set to "Mezuri Provenance <provenance@mezuri.org>". The last part of the tag `<update-num>` is required to allow for changes to the specification file without any changes to the component (since the specification file stores other information as well). It starts at 0 and always increments by 1. Therefore, a component version is represented by the tag with the largest update number among all the tags with the same version.

### **mezuri <component-type>|pipeline publish [--local]**

This command publishes the component to a registry. With the `--local` option, the user can choose to make it available only on his own machine (for use in other components and pipelines).



Otherwise, the user must set up a git remote for the repository and the address of the registry they would like to publish to. If the user has not done so earlier, they are prompted to do so. Note that this remote has to be public, or at least accessible to the Mezuri Registry git user. If these were not provided earlier, the command first commits the updated specification file. Then it creates a new tag with the same version number but incremented update number pointing to the new commit. (Both the tag and the commit are authored by Mezuri Provenance.) Then, it sends the component version information, including the url of the remote repository, to the registry. We discuss why the URL is sent in the next section.

```
$ mezuri operator init
Name (only a-z,-): mean
Description: Calculates the mean of a list of numbers or a time series.
Version [0.0.0]: 0.0.1
$ mezuri operator generate
$ mezuri operator commit 0.0.1 "Create first version of mean"
$ mezuri operator publish --local
Pushing branch master to remote
Password for 'http://diby@localhost:8422':
Pushing tag /mezuri/operators/mean/0.0.1/1 to remote
Password for 'http://diby@localhost:8422':
```

Listing 5: Complete setup for the `MeanOperator` (Listing 3), from initialization to publication to local registry.

Listing 6 shows the `mezuri_specifications.json` for the mean operator we defined earlier. Besides the information we provide when setting up the component (`mezuri operator init`), it contains the IOP specifications and definition information (`mezuri operator generate`), the version information (`mezuri operator commit 0.0.1 "Add first version"`) and publication information (`mezuri operator publish`).

### 3.3 Registry

The components and pipelines registry is implemented as a REST API with five endpoints per component type as listed in Table 3.1. The Registry is backed by a MongoDB data store. We selected MongoDB simply because of the ease of getting set up. However, we anticipate moving to a relational database management system in future to better model our data.

When a component is published, the CLI makes a number of calls to the registry API. It first checks if the component already exists in the registry (by calling the `component` endpoint). Components are uniquely identified by their remote git repository URL. This is

Endpoint	URL	Methods
components	/<component-type>	GET, POST
component	/<component-type>/<component-name>	GET
versions	/<component-type>/<component-name>/versions	GET, POST
version	/<component-type>/<component-name>/versions/<version>	GET
dependents	/<component-type>/<component-name>/versions/<version>/dependents	GET

Table 3.1: Registry API Endpoints for each component type and pipeline

one reason the URL is sent to the registry. If the component already exists, the CLI POSTs the new version to the `versions` endpoint.

The `versions` endpoint then goes and clones the git remote repository. By doing so, it verifies that the repository is publicly accessible, and verifies that the version hashes sent by the client matches the version hash of the tag on the remote. Once this verification is complete, the registry extracts the `mezuri_specification.json` file and stores it in its database. The registry does not save a copy of the repository, but stores the git remote URL in its database. Since, the main goal of the registry is to keep track of provenance information, we do not need to save the actual implementation. Our users can always verify that the implementation and the provenance information are in sync by comparing version hashes.

We note that the only methods allowed on the registry endpoints are GET and POST, implying that once a component name is claimed, it can not be used by another component. It also means that once a component version has been published, it can not be edited. (Changes can be made to a version before publishing).

We do not currently have the DELETE method on components and component versions. We plan to do so once we have implemented user authentication (We want only the authenticated author of a component to be able to delete the component or component version.)

## Local Registry

As we noted in the previous section, it is possible to have a local registry so as to be able to use a component without having to figure out how to import it. We have Dockerized the Registry API server[9], and provided a Docker Compose configuration[23] that will start up a local Gitlab instance[12] for storing repositories (accessible on local port 8422) in addition

to starting up an instance of the Registry API server (accessible on local port 8421).

Users will have to manually create a repository for their component on the local Gitlab instance and enter it into the mezuri publish flow, but all other parts of the publish flow are automated.

We note that it may be better to simply consider the local repository the remote repository for the local registry. We would then not have to run a Gitlab instance locally. We have not done so already because of the convenience afforded by the Gitlab GUI for debugging repository related issues on our platform. We will do so before a public release.

## Private Registries

Although we plan to host a public registry, users are free to create and host their own registries. Notably, these might be private registries that corporations might create for use by their employees. Of course, the hope is that over time, these private registries also decide to publish their components to the general public.

## Private Implementations

A corporation or other entity may not want to share their operator, source or interface, but may be fine with publishing the IOP specifications. Consider a database that contains sensitive private information, but which might support queries obtaining aggregate information. Or consider a proprietary machine learning algorithm that an entity exposes as a paid service with corresponding interfaces. In such cases, the entity might give only the Mezuri Registry git user access to the implementation. Only information in the `specifications.json` file would be publicly exposed. Alternatively, the entity might choose to create a set of annotations separate from the implementation. But as we discussed earlier, this is not preferred.

The entity might choose to expose APIs for using these components. Researchers will be able to use it in their own pipelines and record the publicly available provenance information.

## 3.4 Registry Web Frontend

In addition to the Registry API, we have also created a web front-end and rudimentary backend to the registry. This will allow users to browse the components that have been published, and interactively create new ones.

### Library of Components

This is a searchable index of all components. Each component and pipeline gets its own page which has a visualization of the component or pipeline and links to its dependencies.

Interface pages have links to all the Sources and Operators that it can be used with. While Sources and Operators have links to the interfaces they can take.

Components are searchable by tags that users can add to their components, and component authors.

MEZURI REGISTRY										
INTERFACE	time-series (source)									
0.0.1	A sequence of values at successive times									
0.0.3	<table border="1"> <tr> <td>Data Type</td> <td>time_series</td> <td> <div style="display: flex; gap: 5px;"> <span style="background-color: red; color: white; padding: 2px;">STREAM</span> <span style="background-color: blue; color: white; padding: 2px;">STRUCT</span> <span style="background-color: lightblue; padding: 2px;">time</span> <span style="background-color: lightgrey; padding: 2px;">DATETIME</span> </div> <div style="display: flex; gap: 5px;"> <span style="background-color: lightblue; padding: 2px;">value</span> <span style="background-color: lightgrey; padding: 2px;">DOUBLE</span> </div> </td> </tr> <tr> <td>Dependencies</td> <td colspan="2">None</td> </tr> <tr> <td>Dependents</td> <td colspan="2"> <span style="border: 1px solid grey; border-radius: 10px; padding: 2px 5px;">cookstove-timeseries v0.0.4</span> </td> </tr> </table>	Data Type	time_series	<div style="display: flex; gap: 5px;"> <span style="background-color: red; color: white; padding: 2px;">STREAM</span> <span style="background-color: blue; color: white; padding: 2px;">STRUCT</span> <span style="background-color: lightblue; padding: 2px;">time</span> <span style="background-color: lightgrey; padding: 2px;">DATETIME</span> </div> <div style="display: flex; gap: 5px;"> <span style="background-color: lightblue; padding: 2px;">value</span> <span style="background-color: lightgrey; padding: 2px;">DOUBLE</span> </div>	Dependencies	None		Dependents	<span style="border: 1px solid grey; border-radius: 10px; padding: 2px 5px;">cookstove-timeseries v0.0.4</span>	
Data Type	time_series	<div style="display: flex; gap: 5px;"> <span style="background-color: red; color: white; padding: 2px;">STREAM</span> <span style="background-color: blue; color: white; padding: 2px;">STRUCT</span> <span style="background-color: lightblue; padding: 2px;">time</span> <span style="background-color: lightgrey; padding: 2px;">DATETIME</span> </div> <div style="display: flex; gap: 5px;"> <span style="background-color: lightblue; padding: 2px;">value</span> <span style="background-color: lightgrey; padding: 2px;">DOUBLE</span> </div>								
Dependencies	None									
Dependents	<span style="border: 1px solid grey; border-radius: 10px; padding: 2px 5px;">cookstove-timeseries v0.0.4</span>									

Figure 3.2: Screenshot of web interface for the `TimeSeriesInterface` (Listing 1)

The main purpose of this Library of Components is to facilitate collaboration. We hope that researchers will upload their data pipelines to this library and add a link to it in their papers so that it is discoverable.

```

1  {
2    "name": "mean",
3    "componentType": "operators",
4    "description": "Calculate the mean of an array of numbers or a time series",
5    "version": "0.0.1",
6    "iopDeclaration": {
7      "parameters": {},
8      "methods": {
9        "mean": {
10         "input": {
11           "numbers": ["LIST", ["DOUBLE", null]]
12         },
13         "output": {
14           "mean": ["DOUBLE", null]
15         }
16       },
17       "time_series_mean": {
18         "input": {
19           "time_series": [
20             "INTERFACE", ["http://registry.mezuri.org", "time-series",
21               ↪ "0.0.3"]
22           ]
23         },
24         "output": {
25           "mean": ["DOUBLE", null]
26         }
27       }
28     },
29     "dependencies": [
30       {
31         "componentType": "interfaces",
32         "registryUrl": "http://127.0.0.1:8421",
33         "componentName": "time-series",
34         "componentVersion": "0.0.3"
35       }
36     ],
37     "definition": {
38       "file": "operator.py",
39       "class": "Operator"
40     },
41     "publish": {
42       "registry": "http://127.0.0.1:8421",
43       "remote": {
44         "url": "http://diby@localhost:8422/mezuri/operator-mean.git",
45         "name": "gitlab"
46       }
47     }
48 }

```

Listing 6: mezuri\_specifications.json for the MeanOperator (Listing 3).

## Chapter 4

# Case Study: Cook-stove Temperature Time Series Analysis

We put together all the tools we have discussed and demonstrate qualitatively their suitability and usability by annotating an existing data pipeline with provenance information. The pipeline represents analysis performed by our colleagues here at UC Berkeley (and collaborators) on a project that aimed to study the patterns of stove usage after introduction of an advanced cook-stove in over 200 households spread over 7 villages in Haryana, India[25]. We selected this project because the authors describe the analysis they performed in their paper in enough detail to allow us to replicate it.

The advanced cook-stoves were introduced as part of an intervention to replace traditional cook-stoves that used solid biomass as fuel and released a number of poisonous pollutants in the household environment. In order to measure the effectiveness of the intervention, both the traditional and newly-introduced advanced cook-stoves were fitted with Stove Use Monitors (SUMs) that measured and stored the temperature of the stove at 10 minute intervals. These readings were then retrieved and analyzed to figure out when either of them were being used for cooking meals. Since, the temperature of the stove could vary based on the ambient temperature (which varied quite widely day to day), a third reading of ambient temperature was required to give a baseline.

Since the advanced cook-stoves were more efficient, they cooked food faster, and the usage duration for the two types of cook-stoves could not be directly compared. They had to be first scaled by their cooking power. The ratio of usage for the two types of cook-stoves could then be found. By thresholding this ratio, we could figure out if that household had successfully been transitioned to the advanced cook-stove.

## 4.1 Data Pipelines

We build our data pipeline in terms of four pipelines. The first pipeline takes the temperature time-series for a single cook-stove and the ambient temperature time-series and creates a new time-series of cook-stove usage events. The second pipeline aggregates values in the time-series to get total usage for the stove. The third pipeline runs the second for each type of cook-stove, and returns the ratio of usage for the two types. The third pipeline runs the third pipeline on sets of time-series corresponding to each household to calculate effectiveness of the intervention.

### Pipeline 1: Cook-stove Usage Generation

```
1 CookStoveTemperatureTimeSeries = SourceProxyFactory(...)
2 LinearInterpolatingSubtractor = OperatorProxyFactory(...)
3 CookStoveUsageGenerator = OperatorProxyFactory(...)
4 TimeSeriesMultiplier = OperatorProxyFactory(...)
5
6 cook_stove_temp = PipelineStep()
7 with cook_stove_temp.context():
8     CookStoveTemperatureTimeSeries('trad-cook-stove-001.csv').read()
9
10 amb_temp = PipelineStep()
11 with amb_temp.context():
12     CookStoveTemperatureTimeSeries('ambient-001.csv').read()
13
14 stove_amb_diff = PipelineStep()
15 with stove_amb_diff.context():
16     LinearInterpolatingSubtractor().difference(
17         a=cook_stove_temp.output['time_series'],
18         b=amb_temp.output['time_series'])
19
20 stove_usage_time = PipelineStep()
21 with stove_usage_time.context():
22     CookStoveUsageGenerator(threshold_temp=20).generate(
23         temp_diff=stove_amb_diff.output['time_series'])
24
25 stove_usage_energy = PipelineStep()
26 with stove_usage_energy.context():
27     TimeSeriesMultiplier(multiplier=2017).multiply(
28         time_series=stove_usage_time.output['time_series'])
29
30 cook_stove_usage_generation = Pipeline(last_step=stove_usage_energy)
```

Listing 7: Cook-stove Usage Generation Pipeline definition

We use the `TimeSeries` interface we defined in the previous section (Listing 1) along with a `CSVFileReader` source (Listing 2) to read in from the CSV files for each time-series.

Next we define a `LinearInterpolatingSubtractor` operator. This takes two time-series streams and returns the difference in values at each time in the second stream. Since, the two time-series are not synced, the value for the first stream at a time point for the second stream is linearly interpolated from the nearest past and future values for the first stream. This operator is used to get the difference in temperature between the two cook-stoves and the environment.

Next we define a `CookStoveUseGenerator` operator. This takes the temperature differential time-series for a stove, and generates a cook-stove usage time-series, wherein each time is a time at which the cook-stove has started to be used, and the corresponding value is the time duration for which it was determined to be used. The determination of when the cook-stove is in use is done by an algorithm which takes in a number of parameters (for instance, the minimum temperature hike expected when the cook-stove is being used) to tweak its results.

Then we use a `TimeSeriesMultiplier` operator to scale the cook-stove usage time-series by its cooking power, fed in as a parameter.

## Pipeline 2: Cook-stove Usage Aggregation

```
31 TimeSeries2List = OperatorProxyFactory(...)
32 TimeSeriesListSum = OperatorProxyFactory(...)
33
34 time_series_list = PipelineStep()
35 with time_series_list.context():
36     TimeSeries2List().convert(time_series=stove_usage_energy.output['time_series'])
37
38 usage_sum = PipelineStep()
39 with usage_sum.context():
40     TimeSeriesListSum().sum(time_series_list=time_series_list.output['list'])
41
42 cook_stove_usage_aggregation = Pipeline(last_step=usage_sum)
```

Listing 8: Cook-stove Usage Aggregation Pipeline definition

We define a `TimeSeriesListSum` operator to add up the values from each cook-stove usage time-series from the previous. While the operators in the Pipeline 1 worked on streams, the `TimeSeriesSum` operator requires a list. This is not a problem since our time-series are finite. We need an interface conversion operator to convert from `TimeSeries` to `TimeSeriesList`.



**Pipeline 3: Cook-stove Adoption Determination**

```

43 CookStoveUsageAggregator = cook_stove_usage_aggregation.as_component('aggregate')
44 Ratio = OperatorProxyFactory(...)
45 MinThreshold = OperatorProxyFactory(...)
46
47 trad_stove_aggregate = PipelineStep()
48 with trad_stove_aggregate.context():
49     CookStoveUsageAggregator(**{
50         'cook_stove_temp.file': 'trad-cook-stove-001.csv',
51         'amb_temp.file': 'ambient-001.csv'
52     }).aggregate()
53
54 adv_stove_aggregate = PipelineStep()
55 with adv_stove_aggregate.context():
56     CookStoveUsageAggregator(**{
57         'cook_stove_temp.file': 'adv-cook-stove-001.csv',
58         'amb_temp.file': 'ambient-001.csv'
59     }).aggregate()
60
61 ratio = PipelineStep()
62 with ratio.context():
63     Ratio().ratio(x=adv_stove_aggregate.output['sum'],
64                 y=trad_stove_aggregate.output['sum'])
65
66 threshold = PipelineStep()
67 with threshold.context():
68     MinThreshold(threshold=0.8).indicate(ratio.output['ratio'])
69
70 cook_stove_adoption_determination = Pipeline(last_step=threshold)

```

Listing 9: Cook-stove Adoption Determination Pipeline definition

We use a simple `Ratio` operator to get the ratio of usage times (from Pipeline 2) for advanced vs. traditional cook-stove followed by a `Threshold` indicator operator to see if the ratio was high enough to be considered a success. We note that we use Pipeline 2 as a component in Pipeline 3.

**Pipeline 4: Intervention Effectiveness Calculation**

For each household, we determine if it has adopted the new cook-stove (using Pipeline 3 as a component) and then perform a `Sum` over all households. This aggregate divided by the total number of households gives a measure of the effectiveness of the intervention.

Our task is slightly complicated by the fact that we have been hardcoding CSV filenames in the pipeline. So we write a small operator locally called `CookStoveFilesEnumerator` for

enumerating through CSV filenames for all households as a stream. We de-stream it inside the adoption determination step since Pipeline 3 does not take a stream.

## 4.2 Discussion

### Pro: Modularity

Pipelines created on the platform are highly modular. Any given set of steps could be made into their own pipeline quite easily. The built-in version tracking system ensures that data provenance information can be recorded as part of a new version at any point.

In our case study, we could have easily combined Pipeline 1 and Pipeline 2, but choose to keep them separate to demonstrate the usefulness of this modularity: a researcher can choose to publish just Pipeline 1 as a general-purpose `CookStoveUseGenerator`. Other researchers may choose to use this instead of creating their own algorithm for recovering usage events from temperature data. They may tweak the parameter values (`threshold_temp` and `cooking power multiplier`) to fit their own model of cook-stoves/deployment environment. One of these other researchers may realize that considering some additional parameters may improve the algorithm significantly. They could then publish their own `EnhancedCookStoveUseGenerator` pipeline/component, and others including the author of the original `CookStoveUseGenerator` can benefit from this.

### Con: Verbosity

Pipelines created on the platform can be quite verbose, and tedious to write up. It is tedious to have to define and publish a large number of components even before being able to create the first pipeline. This will be especially true during the early stages of adoption of the Public Registry when researchers will often have to create the first versions/first public versions of common operators in different fields.

We also think it is unrealistic and non-productive to break up the pipeline into components for individual operations. We did that in this case study to better demonstrate how generic components could be created and fitted into different scenarios. For real world use, it would make more sense to have a few bigger components.

### Trade-off

The fact remembers that any overhead to the actual data analysis workflow will disincentivize researchers from using our tools. We make two observations regarding this:

- We hope to create value for our researcher users in the form of existing published components on the public registry. We hope that being able to use these components will convince researchers to use our platform.

- We have designed our tools to be just an additional layer on top of implementations. Therefore, researchers could use components from our public registry without using any of our provenance recording tools. This use case is one we are explicitly optimizing our platform for.

This also means that researchers do not need to continuously generate provenance information throughout their data analysis workflow in order to have a functional pipeline. (They could use actual implementations of components in place of the proxy components as we did in Pipeline 4. ) We hope that researchers will go ahead and annotate their implementations with provenance information (and publish those implementations) as they are in the process of publishing their research. We are optimizing our platform for that use case.

```

71 CookStoveAdoptionDeterminator =
   ↪ cook_stove_adoption_determination.as_component('determine')
72 Sum = OperatorProxyFactory(...)
73
74 from lib.definitions import AbstractOperator
75 from lib.declarations import Input, Output, Parameter
76 import lib.types as mezuri_types
77
78 filenames_type_definition = {
79     'trad-stove-temp': mezuri_types.String(),
80     'adv-stove-temp': mezuri_types.String(),
81     'amb-temp': mezuri_types.String()
82 }
83
84 class CookStoveFilesEnumerator(AbstractOperator):
85     @Parameter('filename_format', mezuri_types.String())
86     def __init__(self, filename_format):
87         self.filename_format = filename_format
88
89     @Input('range', mezuri_types.Struct({
90         'start': mezuri_types.Int(),
91         'end': mezuri_types.Int()
92     })))
93     @Output('file_names',
94         ↪ mezuri_types.Stream(mezuri_types.Struct(filenames_type_definition)))
95     def get_filenames(self, range):
96         return map(lambda i: {fn: self.filename_format.format(temp_src=fn, number=i)
97             for fn in filenames_type_definition},
98             range(range['start'], range['end']))
99
100 filenames = PipelineStep()
101 with filenames.context():
102     CookStoveFilesEnumerator('{temp_src}-{number:03d}.csv').get_filenames(
103         {'start': 0, 'end': 195})
104
105 adoption_determination = PipelineStep()
106 with adoption_determination.context() as ctx:
107     fn_out = ctx.as_individual(filenames.output['file_names'])
108     CookStoveAdoptionDeterminator(**{
109         'trad_stove_aggregate.cook_stove_temp.file': fn_out['trad-stove-temp'],
110         'trad_stove_aggregate.amb_temp.file': fn_out['amb-temp'],
111         'adv_stove_aggregate.cook_stove_temp.file': fn_out['adv-stove-temp'],
112         'adv_stove_aggregate.amb_temp.file': fn_out['amb-temp']
113     })
114
115 sum_op = PipelineStep()
116 with sum_op.context():
117     Sum().sum(numbers=adoption_determination.output)
118
119 intervention_effectiveness_calculation = Pipeline(last_step=sum_op)

```

Listing 10: Cook-stove Adoption Determination Pipeline definition

# Chapter 5

## Future Work

Our work on the Data Provenance Management Platform has only just began, and there is a long way to go. Besides general maintenance and improvement of the existing pieces, there are a number of new directions that we hope to pursue.

### 5.1 Ecosystem improvements

#### **Building out the registry with widely used components**

To catalyze usage of the registry, we will be building out common operators, interfaces and interface conversion operators in different domains. So when researchers look to use our platform, they can go straight to adding their work instead of laying scaffolding. We anticipate that our implementations will be somewhat hit or miss given we have limited domain knowledge, and we hope domain experts will improve the interfaces. (Our implementations will all be tagged version 0.0.1. )

#### **Building annotation libraries in other languages**

As we discussed earlier, we would also like to build annotation libraries in other languages to encourage our users to record provenance information alongside their data pipeline implementations.

The languages that we are considering tackling next are R and MATLAB. This would allow us to target a large subset of datascientists.

#### **Allowing the creation of simple components on the web interface**

We will be adding the option of creating simple components on the web interface. This will be a interactive process wherein users will specify the IOP specifications of their component, and optionally add code to implement the functionality of their components.

Currently, our web interface is only a frontend that takes directly to the registry API. But for the creation and management of a Git repository for the component, we require a separate server instance (in addition to the Registry server).

## Integration with Jupyter notebooks

Jupyter notebooks have support for over 40 languages[17] and are widely used by researchers to run their data pipelines. Jupyter allows extensions to their notebooks to integrate with outside services. We would like to integrate access to the public registry with jupyter notebooks so that users can record provenance information directly from their notebooks.

## Integration with the Dataset Version Control System

We have structured our code to make it easier to integrate with different version control systems. Currently, we support only the Git Version Control System. The DataHub team are developing a new VCS for datasets which they call the Dataset Version Control System (DVCS)[4]. We plan to support DVCS in future as well.

## 5.2 Publicizing our platform

We hope that our Platform will become the *de facto* place for recording data provenance. Once we have made the ecosystem improvements listed above among others, we will look to publicize the platform. The first step leading to that is letting researchers in different fields know that such a platform exists, and educating them in how it can potentially augment their work results.

### Colleagues and Collaborators

We plan to start with our fellow collaborators on the Mezuri Platform at UC Berkeley, University of Michigan and University of Washington, and fellow researchers at UC Berkeley.

Additionally, we will be looking to reach out to corporations that have significant data science departments about potentially test driving the platform.

### Encouraging community contributions

As we mentioned in the introduction, we plan to make all our tools open-source on Github, and we welcome contributions.

# Chapter 6

## Conclusion

### 6.1 Contributions

In this technical report, we presented a set of tools and systems for recording and publishing data provenance information that are built on top of the Git Version Control System and are geared towards data scientists of all research fields publishing the results of their research. We call this the Mezuri Data Provenance Management Platform, or Mezuri Provenance for short.

The platform has two major components:

1. **Local Provenance Manager:** This is a set of software packages that users install on their development machines. It has the following subcomponents:
  - Annotation Library (Section 3.1): A set of classes that users use to annotate their Data Components (data processing tools, datasets)(Section 2.2) and Data Pipelines (Section 2.4) with provenance information. *We presented and discussed an implementation of these classes in Python, and plan to add similar libraries in other languages that are popular with data scientists.*
  - Local Component Registry (Section 3.3): A set of servers running inside Docker containers that implement a local version of a Data Component Registry. *We presented a Dockerized implementation of this API and currently use a local Gitlab Docker container to host our Data Component repositories.*
  - Command Line Interface (Section 3.2): A command line tool for creating Data Component versions and publishing them to the Component Registry, either local or remote. *We presented a python implementation of this.*
2. **Public Web Registry** (Section 3.4): This is the public-facing component, a frontend to a public Component Registry on the internet. *We presented a proof-of-concept implementation showcasing how this interface could be used to browse published Data*

*Components and Pipelines.* In future, this web interface will also allow users to create simple Data Components and Pipelines.

## 6.2 The Big Picture

In the Introduction, we argued that scientists quite often fail to provide sufficient data provenance information on their publications because currently available tools for recording and publishing data provenance are inadequate. We hope that Mezuri Provenance will at least partially fill this void.

We have a long way to go to make this a production-ready and fully-featured platform but we present here our vision (and goal) of what the future could look like.

### Vision for the future

*A data scientist is exploring newly published papers from the Annual Conference on Computational Genomics on the arXiv eprint archive. A paper on ... catches her attention and she wants to explore the presented data analysis further. She follows the reference presented in the paper to <https://provenance.mezuri.org/pipelines/rice-gene-comp> where she views a graph diagram of the entire workflow that the authors of the paper created to obtain their results. She clicks on one of the Data Sources in the graph and a sidebar pops out from the right showing v2.1.0 of the 3,000 Rice Genomes Project used in the pipeline. From there, she is able to follow the steps the authors took to obtain their results.*

*This gets the data scientist thinking. She explores some of the Operators from the related section of the 3,000 Rice Genomes Project. Back on the original pipeline page, she clicks on Open Locally and is taken to `mezuri-prov://pipelines/rice-gene-comp?registry=provenance.mezuri.org`. Behind the scenes, her local Provenance Manager downloads the Pipeline, and opens it in the browser. She clicks on Get Component, and the right sidebar opens up with a single box representing the entire pipeline. She drags this onto the New Pipeline button, and the Create New Pipeline opens up with just that component. In the left sidebar, she gets suggestions for other operators to add to the pipeline, but she drops into a terminal window and types in `mezuri operator init`. She creates a basic python definition for the Operator and back in the terminal, types in `mezuri operator commit --publish -local 0.0.1`. She is now able to see the operator on the local Provenance Manager and drags it onto the new pipeline she just created...*

*The data scientist is getting ready to publish her analysis building off the Rice Genetics Comparison paper. She creates the final version of her Pipeline `mezuri`*



*pipeline commit 1.0.0 and publishes it to <https://provenance.mezuri.org/pipelines/rice-gene-similarities>. She adds the link to her paper and uploads it to arXiv for submission.*

# Appendix A

## Git Object Hash Calculation

In this Appendix, we discuss how Git calculates hashes for git commit and tag objects with the aim of proving that we capture all pertinent data provenance information in component version hashes.

To obtain the object hash, git calculates the SHA1 hash of the following blob[6]:

```
printf "${obj_type} %s\0" $(git cat-file ${obj_type} ${obj_ref} | wc -c)
git cat-file ${obj_type} ${obj_ref}
```

Here, the first line is of the form “commit 176\0” where 176 is the size of the commit object file. This is followed by the contents of the object file.

### A.1 git commit

A typical commit hash object blob for a component looks as follows:

```
commit 251\0tree f77f9adcf7d67563ab5d11637403ba3e68311fd2
parent 9f3591fbb6db25cfa864f3e32b5271a8425dd643
author Mezuri Provenance <provenance@mezuri.org> 1494804281 -0700
committer Mezuri Provenance <provenance@mezuri.org> 1494804281 -0700
```

Update specification

The tree object reference is a reference to the state of the tree at the time of the commit. Its hash is the Merkle hash[19] with files as leaf nodes and directories as non-leaf nodes. Leaf hashes are the hashes of the contents of the file they represent.

The parent object reference is a reference to the previous commit on this branch.

The hash of this blob is `e12a8b412a91982de5237d8b00c2b77d2d7322cd`. We note that that hash is derived from the combination of the parent commit hash and the tree hash of the repository and therefore, captures both history and current state of the repository.

## A.2 `git tag`

The corresponding tag hash object blob for the component looks as follows:

```
tag 190\0object e12a8b412a91982de5237d8b00c2b77d2d7322cd
type commit
tag mezuri/operators/mean/0.0.1/1
tagger Mezuri Provenance <provenance@mezuri.org> 1494804281 -0700
```

```
Create first version of mean
```

We note that the object pointed to by the tag is the commit displayed above.

The hash of this blob is `92f6299b88ace9e11eb996b7b8ab6fdc106dfd09`. This captures history and current state of the repository and all relevant information regarding the tag. This is what we use as the version hash. Therefore we capture all pertinent data provenance information in the component version hash.

## Appendix B

# python Pipeline Definition DSL

```

1 class OperatorProxyFactory(AbstractComponentProxyFactory):
2     data_type = 'OPERATOR'
3     component_type = 'operators'
4
5     def __call__(self, **param_kwargs):
6         param_specs = self.specs[SPEC_IOP_DECLARATION_KEY]['parameters']
7         if set(param_kwargs.keys()) != set(param_specs):
8             raise PipelineError('arguments to __init__ do not match parameter '
9                                 'specifications')
10
11        for name, type_ in param_kwargs.items():
12            if type_ != param_specs[name]:
13                raise PipelineError("type of argument '{}' does not match parameter "
14                                    "specifications".format(name))
15
16        return super().__call__(**param_kwargs)
17
18    def __getattr__(self, method_name: str):
19        method_specs = self.specs['iopDeclaration']['methods'].get(method_name, None)
20        if method_specs is None:
21            raise AttributeError('{} has no output method {}'.format(
22                self.specs[SPEC_DEFINITION_KEY]['class'], method_name))
23
24        method_specs['input'] = {k: mezuri_types.get_deserialized(v)
25                                for k, v in method_specs['input'].items()}
26        method_specs['output'] = {k: mezuri_types.get_deserialized(v)
27                                  for k, v in method_specs['output'].items()}
28        return super().ComponentMethodProxy(self, method_name, method_specs)

```

Listing 11: OperatoryProxy definition excerpt

```

1 class AbstractComponentProxyFactory(mezuri_types.AbstractMezuriSerializable):
2     ...
3
4     @abstractmethod
5     def __call__(self, **param_kwargs):
6         if not PipelineStepContext().in_context:
7             raise PipelineError('{}() can only be called in a pipeline step '
8                                 'context'.format(str(self)))
9
10        PipelineStepContext().notify_method_call_in_context(MethodCall(
11            self, '__init__', param_kwargs, {}))
12
13        return self
14
15    class ComponentMethodProxy(object):
16        def __call__(self, **input_kwargs):
17            if not PipelineStepContext().in_context:
18                raise PipelineError('{} can only be called in a pipeline step '
19                                    'context'.format(str(self)))
20
21            input_specs = self._method_specs.get('input', {})
22            if set(input_kwargs) != set(input_specs):
23                raise PipelineError("arguments to method '{}' do not match method "
24                                    "input specifications".format(self._method_name))
25
26            for name, type_ in input_kwargs.items():
27                if type_ != input_specs[name]:
28                    raise PipelineError(
29                        "type of argument '{}' to method {} does not match method "
30                        "input specifications".format(name, self._method_name))
31
32            PipelineStepContext().notify_method_call_in_context(MethodCall(
33                self._proxy, self._method_name, input_kwargs,
34                ↪ self._method_specs['output']
35            ))
36            return self._method_specs['output']

```

Listing 12: AbstractComponentProxy definition excerpt

The Pipeline Definition DSL provides a simple way to define pipeline steps and compose them to form pipelines, while performing validation of component initializations and method calls.

## B.1 Methods, Parameters and Inputs validation

While defining pipelines, users set up pipeline steps by making method calls on the

Component Proxy classes. When method calls are made on these classes, they dynamically go in and check first, that the method is a valid method (either it is the `__init__` method for setting parameters, or it is an IO method with `Input` and `Output` annotations), and that the arguments provided match the specifications for the method.

Consider the following pipeline step definition:

```
stove_usage_time = PipelineStep()
with stove_usage_time.context():
    operator = CookStoveUsageGenerator(threshold_temp=20)
    operator.generate(temp_diff=stove_amb_diff.output['time_series'])
```

Here, `CookStoveUsageGenerator` is an instance of class `OperatorProxyFactory`. The `__call__` method of the instance is invoked on the first line of the step definition. Looking at Listing 11, we see that `__call__` validates the Parameters passed in.

The `__getattr__` method is invoked on the second line of the step definition since the `generate` method has not been explicitly defined on `OperatorProxyFactory`. `__getattr__` ensures that `generate` is an annotated method for the operator and returns a `ComponentMethodProxy` object. Looking at its `__call__` method (Listing 12), we see that it in turn validates the Inputs passed in.

## B.2 Pipeline Step validation

The `PipelineStep` itself performs some validations. It does so by making use of the `PipelineStepContext` singleton class (Listing 13). Component proxies send `MethodCall` notifications to this context class (Listing 12, lines 10, 32) and the `PipelineStep` class subscribes to these notifications (Listing 14).

`PipelineStep` ensures that exactly one component has been instantiated in the step, and that that one component is not instantiated multiple times. It also ensures that exactly one component method (other than `__init__`) has been called.

`PipelineStep` also keeps track of all the previous steps whose output has been accessed inside the step. To achieve this, `PipelineStep` itself sends `StepOutputAccess` notifications to `PipelineStepContext` and the current step subscribes to these notifications.

## B.3 Version Hash Calculation

The final role that the Pipeline DSL performs is the calculation of the version hash for the pipeline.

The process starts at the Component Proxy level which retrieve their version hash from the registry when they are retrieving their specifications. `PipelineStep` records all method

```

1 class PipelineStepContext(SingletonClass):
2     _in_ctx = False # This is not thread-safe.
3     _mc_callback = None
4     _soa_callback = None
5
6     @property
7     def in_context(self):
8         return self._in_ctx
9
10    def context(self, method_call_callback=None, step_output_access_callback=None):
11        self._mc_callback = method_call_callback
12        self._soa_callback = step_output_access_callback
13        return self
14
15    def __enter__(self):
16        self._in_ctx = True
17        return self
18
19    def __exit__(self, exc_type, exc_val, exc_tb):
20        self._in_ctx = False
21        self._mc_callback = None
22        self._soa_callback = None
23
24    def notify_method_call_in_context(self, method_call: MethodCall):
25        if self._in_ctx and self._mc_callback is not None:
26            self._mc_callback(method_call)
27
28    def notify_step_output_access_in_context(self, step_output_access:
29        ↪ StepOutputAccess):
30        if self._in_ctx and self._soa_callback is not None:
31            self._soa_callback(step_output_access)

```

Listing 13: PipelineStepContext definition excerpt

calls within its context, and so knows which component was used. It also records all accesses to outputs of previous methods, and so knows all step dependencies. Therefore, it and all its step dependencies can recursively calculate their own version hash following the procedure described in Section 2.4.

The Pipeline’s version hash is just the version hash of the last step.

```

1 class PipelineStep(object):
2     @contextmanager
3     def context(self):
4         if self._is_set:
5             raise PipelineError('pipeline step already set up')
6
7         try:
8             with PipelineStepContext().context(self._validate_and_record_method_call,
9                                                 self._record_step_output_access):
10
11                 yield self
12         except:
13             self._reset()
14             raise
15
16         if self._output is None:
17             raise PipelineError('no component methods have been called for producing
18                                     ↪ output')
19         self._is_set = True
20
21     def _validate_and_record_method_call(self, method_call: MethodCall):
22         component_class = method_call.class_
23         if self._component is not None:
24             if component_class != self._component:
25                 raise PipelineError(
26                     'component {} used when component {} is already being used '
27                     'in step'.format(component_class, self._component))
28         else:
29             self._component = component_class
30
31         if method_call.method == '__init__':
32             if self._component_initialized:
33                 raise PipelineError('component being reinitialized in step')
34             self._component_initialized = True
35         else:
36             if self._output is not None:
37                 raise PipelineError('a component method has already been called')
38             self._output = method_call.output_specs
39
40         self._method_calls.append(method_call)
41
42     def _record_step_output_access(self, step_output_access: StepOutputAccess):
43         self._prev_steps.add(step_output_access.step)
44
45     @property
46     def output(self):
47         PipelineStepContext().notify_step_output_access_in_context(StepOutputAccess(
48             ↪ self))
49         return self._output

```

Listing 14: PipelineStep definition excerpt



# Bibliography

- [1] *Apache Atlas - Data Governance and Metadata framework for Hadoop*. [Online; accessed 17-May-2017]. URL: <http://atlas.incubator.apache.org>.
- [2] *Apache Taverna - Apache Taverna (incubating)*. [Online; accessed 17-May-2017]. URL: <https://taverna.incubator.apache.org/>.
- [3] *arXiv.org help = arXiv Primer*. [Online; accessed 14-May-2017]. URL: <https://arxiv.org/help/primer>.
- [4] Anant Bhardwaj et al. “Datahub: Collaborative data science & dataset version management at scale”. In: *In CIDR*. 2015.
- [5] *Bitbucket | The Git solution for professional teams*. [Online; accessed 17-May-2017]. URL: <https://bitbucket.org>.
- [6] Scott Chacon. *Pro Git Book*. 2014. URL: [https://git-scm.com/book/en/v2/Git-Internals-Git-Objects#\\_object\\_storage](https://git-scm.com/book/en/v2/Git-Internals-Git-Objects#_object_storage).
- [7] *Cloudera Navigator: data governance solution for Hadoop*. [Online; accessed 17-May-2017]. URL: <https://www.cloudera.com/products/product-components/cloudera-navigator.html>.
- [8] *DataHub*. [Online; accessed 17-May-2017]. URL: <https://datahub.csail.mit.edu/www>.
- [9] *Docker - Build, Ship and Run Any App, Anywhere*. [Online; accessed 17-May-2017]. URL: <https://www.docker.com>.
- [10] Ian Foster et al. “Chimera: A Virtual Data System For Representing, Querying, and Automating Data Derivation”. In: *In Proceedings of the 14th Conference on Scientific and Statistical Database Management*. 2002, pp. 37–46.
- [11] *GitHub - About*. [Online; accessed 17-May-2017]. URL: <https://github.com/about>.
- [12] *Gitlab Docker Images - Gitlab Documentation*. [Online; accessed 17-May-2017]. URL: <https://docs.gitlab.com/omnibus/docker/README.html>.
- [13] Alon Halevy et al. “Goods: Organizing Google’s Datasets”. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD ’16. San Francisco, California, USA: ACM, 2016, pp. 795–806. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2903730. URL: <http://doi.acm.org/10.1145/2882903.2903730>.

- [14] Joseph M. Hellerstein et al. “Ground: A Data Context Service”. In: *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. 2017. URL: <http://cidrdb.org/cidr2017/papers/p111-hellerstein-cidr17.pdf>.
- [15] Home . *linkedin/WhereHows Wiki*. [Online; accessed 17-May-2017]. URL: <https://github.com/linkedin/WhereHows/wiki>.
- [16] *IBM - What is big data?* [Online; accessed 14-May-2017]. URL: <http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html>.
- [17] jupyter. *Project Jupyter*. [Online; accessed 15-May-2017]. URL: <http://jupyter.org>.
- [18] Michael Maddox et al. “Decibel: The Relational Dataset Branching System”. In: *Proc. VLDB Endow*. 9.9 (May 2016), pp. 624–635. ISSN: 2150-8097. DOI: 10.14778/2947618.2947619. URL: <http://dx.doi.org/10.14778/2947618.2947619>.
- [19] Ralph C. Merkle. “A Digital Signature Based on a Conventional Encryption Function”. In: *Advances in Cryptology — CRYPTO ’87: Proceedings*. Ed. by Carl Pomerance. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378. ISBN: 978-3-540-48184-3. DOI: 10.1007/3-540-48184-2\_32. URL: [http://dx.doi.org/10.1007/3-540-48184-2\\_32](http://dx.doi.org/10.1007/3-540-48184-2_32).
- [20] *Mezuri Provenance Github Organization*. [Online; accessed 17-May-2017]. URL: <https://github.com/mezuri-provenance>.
- [21] Hui Miao, Amit Chavan, and Amol Deshpande. “ProvDB: Lifecycle Management of Collaborative Analysis Workflows”. In: *Proceedings of the 2Nd Workshop on Human-In-the-Loop Data Analytics*. HILDA’17. Chicago, IL, USA: ACM, 2017, 7:1–7:6. ISBN: 978-1-4503-5029-7. DOI: 10.1145/3077257.3077267. URL: <http://doi.acm.org/10.1145/3077257.3077267>.
- [22] *npm | npm Documentation*. [Online; accessed 17-May-2017]. URL: <https://docs.npmjs.com/cli/npm>.
- [23] *Overview of Docker Compose - Docker Documentation*. [Online; accessed 17-May-2017]. URL: <https://docs.docker.com/compose/overview>.
- [24] *PASS: Provenance-Aware Storage Systems / SYRAH*. [Online; accessed 17-May-2017]. URL: <https://syrah.eecs.harvard.edu/pass>.
- [25] Ajay Pillarisetti et al. “Patterns of Stove Usage after Introduction of an Advanced Cookstove: The Long-Term Application of Household Sensors”. In: *Environmental Science & Technology* 48.24 (2014). PMID: 25390366, pp. 14525–14533. DOI: 10.1021/es504624c. eprint: <http://dx.doi.org/10.1021/es504624c>. URL: <http://dx.doi.org/10.1021/es504624c>.
- [26] *pip 9.0.1 : Python Package Index*. [Online; accessed 17-May-2017]. URL: <https://pypi.python.org/pypi/pip>.

- [27] Tom Preston-Werner. *Semantic Versioning 2.0.0 / Semantic versioning*. [Online; accessed 17-May-2017]. URL: <http://semver.org>.
- [28] *RubyGems.org / your community gem host*. [Online; accessed 17-May-2017]. URL: <https://rubygems.org>.
- [29] *The Kepler Project – Kepler*. [Online; accessed 17-May-2017]. URL: <https://kepler-project.org>.
- [30] *VisTrailsWiki*. [Online; accessed 17-May-2017]. URL: [https://www.vistrails.org/index.php/Main\\_Page](https://www.vistrails.org/index.php/Main_Page).