Building Tools to Support FAIR Metadata Authorship, Automatic Capture, and Analysis for Scientific Discovery and Reproducibility

A dissertation

submitted by

Polina Shpilker

In partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Computer Science

TUFTS UNIVERSITY

August 2025

ADVISOR: Prof. Lenore J Cowen

Building Tools to Support FAIR Metadata Authorship, Automatic Capture, and Analysis for Scientific Discovery and Reproducibility

Polina Shpilker

ADVIS0R: Prof. Lenore J Cowen

Metadata is simultaneously critical for research reproducibility and discovery and difficult to annotate and store. Researchers often leave metadata annotation for the very end of their research projects, once submission and publication demands it of them for acceptance. Minimal tooling exists to try and alleviate this difficulty, but these tools often fall in the realm of being too free form for automated processing or being too complex for human annotation. I describe the process of tackling this difficulty from two perspectives: first, the development of a new metadata language that is simultaneously human-writable and machine-parseable, along with tooling for this language and secondly, the development of tools that aggregate massive amounts of metadata from high performance compute clusters into formats that may be more easily analyzed by human researchers.

To my mom and sister, who supported me all the way to this point in my life.

To all my friends and advisors who kept me on this path 'til completion.

Finally, to my cat Kiki, who kept me sane whether she liked it or not.

Acknowledgments

Much of the content of chapter 2 of this thesis is based on the paper "MEDFORD: A human- and machine-readable metadata markup language" by Polina Shpilker, John Freeman, Hailey McKelvie, Jill Ashey, Jay-Miguel Fonticella, Hollie Putnam, Jane Greenberg, Lenore Cowen, Alva Couch, and Noah Daniels, which was published in Database in 2022. Much of chapter 4 is based on the paper "Cross-referencing Metadata through an extension of the MEDFORD language" by Polina Shpilker, Benjamin Stubbs, Michael Sayers, Lenore Cowen, Shaun Wallace, Alva Couch, and Noah Daniels, presented at the Metadata and Semantics Research Conference (MTSR 2024). I thank my co-authors and collaborators for their permission to include our joint published work in this thesis. I also would like to especially thank my advisors and mentors, Dr. Lenore Cowen at Tufts University and Dr. Line Pouchard in the US Department of Energy at Sandia National Laboratories. I thank the National Science Foundation for partially funding the research summarized in these papers through grants OAC-1939263, OAC-193979, OAC-1939795, and HDR-BIO NSF-OAC 1940233, and a seed grant from the Tufts Data Intensive Study Center.

Polina Shpilker

TUFTS UNIVERSITY

August 2025

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Metadata is defined as data that exists to provide information about other data. For example, consider a file on a personal machine. It inherently has metadata – the file has a name, a file type, a date of creation, a date of modification, and so on. Often, these annotations are not consequences of the data itself but rather describe attributes about its existence as an object. The file could easily exist with the name "a" or the name "thesis-draft-01-10.tex", no matter its contents, and its date of modification says nothing about the data stored inside.

Each attribute of metadata has an individual level of importance and specificity. For example, while a name is usually used only to help a user remember which file they are working with, the file type ensures that the system recognizes what software to use when opening the file. Interestingly – and perhaps uniquely – metadata's usefulness is entirely determined by the situation and the effort put into authoring it. Imagine if the "date modified" field contained only the word "recently". How recently? Recently as of when? (and good luck searching your operating system for a file with the date modified of "recently".) In consequence, it becomes vitally important to create descriptive metadata, otherwise the metadata may not be useful at all.

Writing descriptive metadata is not as easy as it sounds. Just like how

the content of a metadata field is situationally informative, the metadata fields themselves are situationally informative. For example, a photo would still require the file fields: name, file type, and date created/modified, but these are now insufficient to completely describe the file. One has to consider what sort of metadata may be relevant for research using photographs. Does the photograph have GPS information such as latitude and longitude? Does it have resolution information? Does it have more unique information such as lens focal length? Perhaps most important – what subset of this unique metadata is useful for researchers using this file? Perhaps the researchers do not need to know the focal length or camera model, but GPS information is critically important for the photograph to be used in their research.

It is not sufficient to understand metadata as a concept to be able to accurately and completely describe the metadata of data. The author must understand what metadata is available, what metadata is important, and to what level of detail this metadata must be described; these metadata features are often application-specific. Therefore, in my PhD research on metadata, I initially chose a particular application: the study of the coral holobiont by the marine biology community.

1.1.1 A Case of Corals

Corals comprise thousands of different organisms, including the animal host and single celled dinoflagellate algae, bacteria, viruses, and fungi that coexist as a holobiont, or metaorganism [BMN11]. Thus, corals are like cities, rather than the individual animals that inhabit or visit them, as corals provide factories, housing, restaurants, nurseries, and more for an entire ecosystem. Research on coral reefs is pressing, given their local and global contributions to marine biodiversity, coastal protection, and economics and their sensitivity to climate chang [HBB$^+$17, WHN$^+$19]. Research in this area requires integration of interdisciplinary data across multiple environments and a range of data types: 'omic data such as gene expression data generated using RNA-Seq (RNA transcript sequencing), image and time-lapse video, and physical and environmental measurements including light and water temperature, to name but a few. The coral research community has long been committed to sharing and

open data formats, and both individual researchers and large funding agencies have invested heavily in making data available [MHC+16, LAV16, YLL+20, DRH17] and FAIR (findable, accessible, interoperable, and reusable) [WDA+16]. The next section will describe the FAIR principles and give a background on how they came to be invented and adopted.

Effective data sharing for coral research, as in all data-intensive domains, requires metadata, which is essential for data organization, discovery, access, use, reuse, interoperability, and overall management [LNH+20]. The growing amount of digital data over the last several decades has resulted in a proliferation of metadata standards supporting these functions [BGJK16, QBG12]. However, the proposed mechanisms to create metadata have been focused primarily on the ease of machine parsing and have recommended schemata that are cumbersome and difficult for humans attempting to create, edit, or read the metadata. If creating metadata in the appropriate format is difficult, or requires expert curators, then fewer scientists will be able to comply with metadata recommended standards, leading to scientific data that is not discoverable, and thus not reusable. Meanwhile, an increasing amount of scientific data in multiple countries (including in the US and the EU) now falls under mandated data sharing policies that require specification of adequate metadata for discovery. Thus, there is a need for a format that streamlines the process of providing what is mandated by law and policy.

1.2 FAIR Metadata

The FAIR principles are principles that were defined to help ensure that data and results are Findable, Accessible, Interoperable, and Reusable [JdMAJ+20]. These principles describe specific treatment and attributes that should be considered when authoring or storing data; for example, to ensure data is findable, it should be identified with a unique and persistent identifier (F1) and described using rich metadata that directly links to this identifier (F2). The purpose of FAIR data and metadata is to thoroughly describe research data such that it can be machine-actionable and

thus be discoverable and reusable by human researchers that must rely on databases and discovery tools to search the wide landscape of available data.

However, these are only principles and guidelines, not a guidebook for implementation. This is intentional - the architechts of the FAIR principles do not want to promote a particular set of software nor limit their applicability by imposing a particular structure. Therefore, it is left up to researchers and data curators to figure out how to apply these principles into a set of standards and processes to support FAIR research in their field. This challenge has seen many recent developments on many fronts: FAIR itself, metadata specifications [QBG12], and particular applications such as cloud workflows and research software [MNV+17] [BCHK+22].

While FAIR is powerful and a goal that data providers must strive towards, we still need to design a strategy to collect and store FAIR metadata.

1.3 Existing Metadata Collection Solutions

There has already been a wide variety of methods developed for storing metadata. These methods are nearly as varied as the data they define, each having its own emphasis on aspects such as human or machine readability, ease of use, and longevity. I will discuss three major types of existing solutions: traditional and analog metadata storage, metadata-specific digital formats, and code-adjacent metadata.

1.3.1 Traditional and Analog Metadata

The most straightforward approach towards storing metadata is simply to write it without any special treatment. Be it literally with a pen and pencil or by typing alongside other notes, these methods lack emphasis of the structured trait of metadata. Researchers may add metadata as part of their overall workflow description or directly on the back of physical data such as photographs. These methods are extremely easy to use for researchers as it takes little to no additional time nor organization.

A classical example is a vintage photograph: the date and subject were often

written directly on the back of the photograph. This method is extremely easy for the photographer; they simply take the photo, and when they receive the printed version back from the photo studio, they take a pen and note down when they took it. It takes minimal time out of their day and theoretically stores this information for posterity.

An issue arises when you expand this situation to someone who manages thousands of photos – perhaps tracking the growth of corals over time. Without the addition of some external structure, this metadata cannot be used. Assuming they simply wrote the dates and placed all the photographs in a pile; at worst they would have to search through every single photo to find the one they were looking for.

A solution to traditional metadata's pitfalls is to add organization: perhaps the photographer sorted them by day. However, organization is of limited utility as the data set gets larger. What if a researcher is looking for a particular coral species within a range of dates? Again in that case, because the metadata is stored individually on each photograph rather than within an organized collection, they may have to search every single photograph within that time range.

All of the solutions provided thus far have been conceptual and require some form of external documentation to persist or be usable by other individuals. One solution to enforcing this in a structured, consistent manner is by using formats dedicated to storing metadata.

1.3.2 Metadata-Specific Formats

There are several key developments that have already been made to try and normalize metadata description. These developments vary between packaging the metadata beside the data, being a language dedicated for describing metadata, and something in between.

One example is BagIt [KLM+18], a format that will be described in further detail in Chapter 2. BagIt is a packaging-focused format; rather than providing a guideline on how to describe metadata, it focuses on providing guidelines on how to

package files with their metadata in a way that can be shared to other researchers or interested parties. A BagIt bag is a directory containing three major components: a data directory containing all data files called the "payload" directory, a `manifest` file containing checksums for all data files, and a `bagit.txt` file describing the metadata about the bag itself. The bag may be optionally compressed for easy transfer. In addition, BagIt provides guidelines on how to include metadata about the files contained within the bag using a `bag-info.txt` file, such as the source organization, bag creation date, and contact information.

Of a similar goal but different complexity to BagIt is the format RO-Crate [SRSC+22]. Much like BagIt, RO-Crate defines a collection of data files (called a "crate"), augmented with a file for describing the crate's metadata called `ro-crate-metadata.json`. While BagIt only suggests metadata descriptions, RO-Crate focuses on clearly and specifically defining metadata expectations. RO-Crates may be associated to RO-Crate Profiles that clearly define expectations for the metadata contained in the `ro-crate-metadata.json`.

With the vocabulary defined in RO-Crate of the concept of `data entities` and attributes being defined as `MUST` (required), `SHOULD` (strongly recommended), and `MAY` (could be present, but not expected), data curators define profiles that expand upon this vocabulary to create a set of expectations for a particular usage case. These profiles allow for various databases and tools to automatically process the content of an RO-Crate file – for example, the Electronic Lab Notebook format that is built using RO-Crate can be automatically ingested by electronic lab notebook tools such as eLabFTW and Rspace [CMP17, Res25].

In high performance computing circles, the Hierarchical Data Format (HDF5) is a popular way to store data entities [HDF]. Much like BagIt, HDF5 serves as a container with some organizational structure to contain research data. Unlike the previous two formats, HDF5 is a file format in itself and is not simply a container around files. The purpose of HDF5 is to collect research data in a highly compressed format that can then be quickly loaded and acted upon using common scientific research libraries such as `numpy`. This makes the goal of providing research data easy

– not only can you provide your data to an interested party quickly, this interested party can then quickly load your data and work on it using common libraries. HDF5 also creates space for metadata: metadata may be described using any arbitrary keys attached to a dataset or a group of datasets. That being said, while this makes room for the metadata to sit alongside the data, it is completely unstructured and left for the user to define.

1.3.3 Data-Adjacent Metadata

The third prong of metadata development is the concept of data-adjacent metadata – that is, metadata that is kept physically next to the objects it describes. The quintessential example are comments: in the case of strange behavior or unclear requirements that a future user or developer may need to know, the developer writing the software could simply add a comment annotating that information. For example, if an input `.csv` requires a column named "ID", a developer can annotate this in the code, and in doing so, even if they hand off this code to a collaborator with zero explanation or further documentation, the other developer has some metadata on the input of this code.

Much like traditional and unstructured metadata, a critical flaw with comments is that they are freeform. The amount and the usefulness of these comment annotations are entirely up to the developer who wrote them, and even a well-intentioned developer who thoroughly documents their comments may still accidentally omit metadata that is critical for future reusability, such as the fact that they used exactly Python 3.4 or a particular version of a Python library.

One approach to normalizing comments is the use of documentation comments (sometimes referred to as 'docstrings'). These comments are written with the intent to be exported into documentation or collected by the integrated development environment (IDE) used by the developer and provided as helpful tooltips as they code. Unlike freeform comments, docstrings are specifically to describe components like functions and classes and to provide necessary context for using these components. They are generally composed of a summary, a description of every input

and its type, every output and its type, and any exceptions that may be thrown by this function during runtime. Docstrings are specifically for internal use, however. They cannot be used to describe metadata about the overall software, such as input files or package version dependencies, and many research languages (such as Python and R) do not have formal docstring support due to their focus on experimental development and quick scripting.

Bioinformatics in particular has been using R and Python for research. R is often used in an IDE called RStudio that presents a multitude of runtime information to the researcher, letting them picture results immediately and inspect the data that is in memory. Python, originally much less transparent than R since it still had to be run as whole-script files, is now often developed using an environment called Jupyter notebooks. Jupyter notebooks add an interesting method of adding metadata: while still extremely freeform like in-line comments, Jupyter notebooks allow for entire text blocks to be interspersed throughout the file, directly next to the code. These text blocks can provide thorough explanation of the purpose and requirements of the code around it, to the point of giving mathematical proofs and example expected outputs and inputs. A researcher can provide a single Jupyter notebook to a collaborator and this collaborator will have all the context they need to replicate their results and understand results from the scientific perspective.

Jupyter notebooks are still not sufficient for complete metadata description. While a researcher can provide thorough context about the research, the notebooks themselves have no support for linking data files or environment information. The researcher would still have to impose some external organization, such as a README or a BagIt bag, to provide any required input files and to declare package dependencies.

To circumvent this, there exist automated tools to generated metadata from a pipeline. The python package Microbench, meant for benchmarking workflow performance, includes functionality to capture environment information such as installed packages and host information [LL22]. From another angle, there are "all-inclusive" environments that create controlled sandboxes that contain both the data and a data

analytics workflow, such as Nephele or the AnVIL project [WLD+17, SPA+22]. These tools specifically curate the environment in which these workflows run, allowing for workflows to easily be re-run and also being capable of exporting and reporting the environment metadata.

1.4 Contributions of This Thesis

There are two major issues with all of the solutions described earlier in this chapter. First, all of the described methodologies for including metadata oversight or correction are difficult for researchers to write. Highly structured formats with specifications like RO-Crate profiles or RDF have complex vocabularies and metadata relationships that take time for researchers to become familiarized with. Even after learning the new vocabulary, it is likely that the vocabulary specified by these formats does not perfectly align with the vocabulary used in the author's research, forcing them to find a way to translate what they want to write into what the format expects. While these formats often have tools to aid with the process of writing the metadata itself, the tools themselves can be complex and at times require purchase.

Therefore, the first set of contributions of this thesis were to develop a more user-friendly alternative to these highly structured formats. This format is dedicated to being as easy as possible for researchers to craft and store alongside their data while still being machine parseable so that it can interface with the complex standards mentioned earlier or parsed by databases directly. This novel metadata description language, MEDFORD, is described in chapter 2. To better support its adoption, we develop extensions to allow for the description of metadata expectations and metadata relationships like those used in RO-Crate Profiles in Chapters 3 and 4.

Secondly, while the described RDF and RO-Crate languages are built to describe complex and structured metadata, such as those from High Performance Compute (HPC) clusters, these metadata are highly dense and often scattered across storage. In the case of workflows run on HPC, there are many individual components

that each generate their own metadata, and there is very limited existing tooling to collect these metadata and provide them to users for analysis. Rather than the user-facing metadata collection suite described earlier, there needs to be a suite of tools that collects scattered, complex metadata into a single location and a single approachable format for researchers to analyze.

To resolve this, in Chapter 5, we describe the second major contribution of this thesis. We develop a suite of tools that sit as part of HPC pipelines, collecting metadata from the various software that make up these pipelines. I also describe the process of taking these highly heterogeneous metadata and developing a novel unified description format that better suits the needs of a high performance compute cluster.

These contributions support thorough and uniform metadata annotations either by making it easier for researchers annotate the metadata themselves or by annotating the massive amount of metadata available automatically. By ensuring these annotations exist in a consistent format, these contributions help researchers make their metadata FAIR and supports research reproducibility and discoverability. In Chapter 6, we describe further work that can be done to improve the work done in other chapters and further support FAIR metadata.

Chapter 2

MEDFORD: A human and machine readable metadata markup language

2.1 Introduction

For the purposes of maintaining and transferring data itself, there already exists a format known as BagIt [KLM+18] that can handle stable transfer of arbitrary files and their directory structure. BagIt specifies a file directory called a "bag" that contains an arbitrary directory structure, a payload manifest, and a bag metadata file. The arbitrary directory structure , called the "data" directory or the "payload", contains all the data files related to the research project. The payload manifest ensures that all data is transferred without error by containing a checksum hash of the data contained within the payload. The bag metadata file, named `bagit.txt`, contains metadata about the BagIt format used to create this bag. Additionally, users may also define a remote manifest that points to remotely available data files that may be related to the data in the bag and their checksums. Users may also write a `bag-info.txt` file, which may contain loosely structured metadata about the research itself. This `bag-info.txt` file allows researchers to write some basic

metadata about the overall project, but does not prescribe a specific metadata format nor provide room to describe metadata in detail.

We built on top of BagIt by developing and implementing the MEtaData Format for Open Reef Data (MEDFORD). The MEDFORD markup language file format is simultaneously human and machine writable and readable. In this regard, we were inspired by the specification language for the Protein Data Bank (PDB) [BWF$^+$00, YWF$^+$18]. PDB files are easily machine-parsable, but unlike JSON files or other commonly-used database submission file types, are also easily human-readable. This human-readability allows for human verification of their contents, although PDB files are still too complex to be written by hand. Unlike the PDB format, MEDFORD is intended to be extensible. MEDFORD is designed to work in conjunction with BagIt's filesystem convention, allowing easily accessible and interoperable bundles of data and metadata to be created and stored. The MEDFORD language is currently implemented as the `medford` parser, which is itself written in Python.

This chapter reports on our first use case, which focuses on the coral holobiont. Specifically, we focus on coral holobiont transcriptomics data (e.g., RNA-Seq, one of the most powerful and common types of omics experiments to explore the genetic basis of factors that lead to coral resilience or vulnerability to environmental stressors), where we build the needed complexity to manage spatial-temporal holobiont expression metadata into MEDFORD from the start. We chose this use case due to difficulty we experienced collecting and organizing metadata about existing coral transcriptomics datasets. A coral researcher, untrained in programming and not a database expert, will be able to directly produce and interpret MEDFORD files more easily than working with RDF authoring tools. We have developed the `medford` parser to automatically translate MEDFORD files into existing file format standards for depositing in databases and repositories. MEDFORD will enable transcriptomic data to be findable, accessible, and interoperable (FAIR) [WDA$^+$16]. While MEDFORD is capable of becoming a general-purpose metadata format, we are implementing the specific use case of coral data to both provide a proof of

concept and to aid the coral research community via a set of detailed metadata constructions specific to coral research. An extended abstract describing MEDFORD appeared as [SFM+22].

2.2 MEDFORD Design Principles

As was described in Chapter 1, languages proposed for metadata specification normally consider ease of *either* human generation and parsing, or machine generation and parsing. Human-legible formats, such as unstructured text files, are easy to write but difficult to store in databases or even provide publicly. Meanwhile, highly structured formats such as RDF and JSON are exceptional for import into databases, but are nearly impossible for a researcher to write on the fly. MEDFORD fills a previously unmet need by intentionally balancing ease of human and machine generation and parsing simultaneously.

In addition to being designed as both a machine and human readable and writable format, we also decided that MEDFORD describes the entirety of a project's metadata within a single file. This allows MEDFORD to be extremely lightweight, and it can be simply incorporated into a BagIt bag without any modification. This ensures that if a MEDFORD file is created, it is straightforward to transfer it alongside its related data.

MEDFORD's design principles are informed by the those underlying highly successful metadata standards, such as the Dublin Core [WK00], Ecological Metadata Language (EML) [FAJS05], and the Data Document Initiative (DDI) [Var14], while addressing additional requirements enabling ease of metadata creation and other aspects. The design requirements for creating MEDFORD were as follows:

1. A mechanism for use by scientists at the point of data collection.

2. A human-readable and human-understandable format for specifying metadata.

3. A simple and easily understandable syntax for specifying metadata elements.

4. The ability to create and reuse templates for specifying metadata for common data types.

5. Applicability beyond the coral use case, to other research domains.

6. The ability to author metadata in a user's preferred text editor without a dependency on special-purpose software.

7. The ability to detect and explain errors in metadata specification via easily understandable error messages targeted toward scientists.

8. Automatic translation into a number of useful machine-readable formats after initial specification, including the Resource Description Format (RDF) [Gro09], Extensible Markup Language (XML) and JavaScript Object notation (JSON), as well as database formats.

These requirements are justified by past experience of scientists crafting metadata [QBG12]. Web-based metadata interfaces can be cumbersome when one is entering metadata for a set of similar data files or publications. The machine-readable formats XML, RDF, and JSON are difficult to understand and edit for scientists who are not programmers. Furthermore, error messages for mistakes in XML, RDF, and JSON specifications are cryptic for those same people. Plain-text specification of metadata, such as the NSF's BCO-DMO resource [CGK+16] is intensive in human labor for those who must then translate it into a machine-readable form in order for BCO-DMO to ingest such data (currently, BCO-DMO requires metadata to be submitted in Rich Text Format, and it is then transcribed by a human operator). Thus, there is a need for an intermediate format that is both machine-readable and human-readable and understandable by the scientists most qualified to specify the metadata correctly.

MEDFORD aims to solve problems associated with specifying interdisciplinary research metadata, as demonstrated by our initial use case applied to coral reef 'omics data. Coral researchers study a wide variety of properties of corals (bioinformatics, growth, bleaching, phylogeny) and for a variety of purposes (ecology, basic

biology, biomedicine). Connecting the work of this diverse group of researchers requires developing sustainable scientific databases so that researchers can discover each others' datasets, integrate them into more novel research, and support further scientific discovery. These databases need to support both accurate analysis of research as well as data discovery and reuse. In general, however, the principles above apply to any scientific metadata specification problem, and the specific extensions identified here may be supplemented for other scientific disciplines. Thus MEDFORD can be used as a tool for metadata creation in any scientific discipline.

The requirements above are realized by MEDFORD by adding design elements that satisfy the above principles:

1. A contextual grammar, devoid of parentheses and the need to close clauses with specific end statements.

2. A simple way to denote kinds of metadata, starting with an `@`, and containing at most three parts: the major tag, minor tag, and the metadata itself. A major tag (such as `@Contributor`) indicates the type of metadata being described, while a minor tag (such as `ORCID` in the context of `@Contributor-ORCID`) indicates the name of the metadata attribute being described.

3. A two-level hierarchy based on major and minor tags organizes the metadata into categories and subcategories which provide the relational structure without compromising the simplicity of the metadata description.

4. A simple concept of user-extensible formatting, in which metadata details not covered by the main keywords can be added via notes.

Consider the following example of a `@Contributor` clause, where `@Contributor` is the major tag and `ORCID` and `Role` are the minor tags which associate those metadata with that contributor.

```
@Contributor Hollie M. Putnam
@Contributor-ORCID 0000-0003-2322-3269
@Contributor-Role Corresponding Author
```

If we wanted to additionally include this contributor's email address, we simply add an additional line:

```
@Contributor Hollie M. Putnam
@Contributor-ORCID 0000-0003-2322-3269
@Contributor-Role Corresponding Author
@Contributor-Email hputnam@uri.edu
```

2.2.1 The MEDFORD Language Syntax

In this section we discuss the principles of the design of the syntax of a MEDFORD file format for metadata. MEDFORD is written in UTF-8 though all reserved tokens and characters fall within the ASCII range, while user-defined tags may use extended UTF-8 characters. MEDFORD tags are indicated with the `@` character. Anything after an `@` character, until the next space in the file, is read as a tag by the `medford` parser. There are two other protected symbols that have special meanings in the MEDFORD language: these are `#` which is treated as a comment character: characters after a `#` on the same line are ignored and not processed by the `medford` parser. Finally, the `$$` string (two dollar signs in a row) is used to indicate the beginning and end of a string that should be parsed by LaTeXmath mode: this enables a MEDFORD language parser to either render or pass through special characters from raw MEDFORD files, in which non-ascii characters are strongly discouraged.

The following design principles are important in MEDFORD file syntax:

- MEDFORD files use the ASCII character set whenever possible. The characters `@`, `#`, and `$$` (as an enclosing pair to denote a region that should left verbatim without processing) are reserved and protected.

- MEDFORD tags are referred to as `@`-tags and always start with the `@` character. Particular `@`-tags are given meanings, and formatting requirements and rules that are either recommended or required

- If a version of the MEDFORD language parser encounters an `@`-tag it does not recognize, the parser passes its associated text through verbatim, treating it identically to how `@COMMENT` is treated. Thus, scientists are free to make up new tags that extend what is currently defined in the language.

To make MEDFORD files more easily human-readable, considering our analogy to the protein databank, we adopted a similar approach to the Fasta file format, where each header line is distinguished by an '`>`' symbol. We chose to use the '`@`' character, as it is commonly used for tagging users or keywords in systems like GitHub and Twitter, and is not found in everyday text except for emails. Therefore, a line headed by an '`@`' symbol can be assumed to be a MEDFORD tag in all cases. Meanwhile, later '`@`' characters have no effect on the `medford` parser, as only '`@`' characters at the very beginning of a line matter.

This is best explained by example. Consider the example of specifying a pipeline used for RNA-seq analysis of coral data (note that the metadata associated with a tag can be arbitrarily long and may span multiple lines):

```
@Software R
@Software-Version 4.0.4 ("Lost Library Book")
@Software-Notes Packages used include dplyr, stringr,
    and genefilter.

@Software DESeq2
@Software-Version 1.28.1
@Software-Notes Used as a package in R.
@Software-Notes Installed through BioCManager.
```

The `Software` tag specifies a piece of software involved in the research. In this case, R is being described as a relevant piece of software, with a `Version` tag used to specify what version of R was used as this is critical information for repeating the analysis. An important feature of this example is the arbitrary difference

between the way R packages are described. The author has determined that DESeq2 is a critical package and decided to use a separate `Software` tag to describe it. Meanwhile, the researcher decided that dplyr and stringr were useful in the analysis but not critical and left them as `Notes` on the R `Software` block. This showcases one of the strengths of the MEDFORD file format; researchers are free to determine whether something is important enough to warrant having a dedicated `Software` tag, or if they can be listed as an arbitrary `Note` on a parent piece of software.

2.2.2 MEDFORD Data Provenance

One of the major goals of MEDFORD is to enable the simple association and description of related but possibly separate data resources. The BagIt filesystem convention [KLM+18] provides a convenient way to wrap multiple files into a consistent directory structure. However, BagIt's own metadata capabilities are limited to describing the files present or how to fetch them from a network. By including a MEDFORD file into a bag, we are able to therefore describe the metadata as well as reference or include the data themselves. MEDFORD does not try to supplant the W3C data provenance standard (RDF) but rather provides a tangible, simple format that meets users' need. A MEDFORD metadata description can be automatically converted into an RDF representation.

All MEDFORD files are defined in reference to a BagIt bag, although the special use-case of an empty bag is common and acceptable. The BagIt bag binds a set of files to the MEDFORD file according to the BagIt standard, where these files describe a variety of resources, including source code, scientific papers, or raw data, each represented by a major tag in the MEDFORD file. The versioning and origins of that file are marked using a secondary major tag, where the tag can represent that the bag is considered to be the primary and authoritative source for the data or resource. Other secondary major tags describe the file as either a copy of an existing source, or simply a pointer to a URI where the resource can be obtained. These tags are shown in Table 2.1

MEDFORD's place in a BagIt directory structure is that the MEDFORD

@Data_Primary	@Code_Primary	@Paper_Primary
@Data_Copy	@Code_Copy	@Paper_Copy
@Data_Ref	@Code_Ref	@Paper_Ref

Table 2.1: Table of data types and their secondary major tags, describing the role of the file contained within the bag.

Researchers wish to create an index of all publicly available RNAseq raw data that have been released on the Internet. They create a MEDFORD file to point to all these data resources, but they will store none of these themselves; the MEDFORD file will just be an index, and all `@Data` tags will be of the form `@Data_Ref`. This is an example MEDFORD file which would be associated with an empty bag.

Figure 2.1: Example use case of the MEDFORD `@Data_Ref` tag.

(`.mfd`) file is placed at the top level of the bagit directory structure. Any files carried along in the BagIt archive exist as `Copy` or `Original` directives (whether `Data`, `Code`, or `Paper`). The BagIt `manifest-sha512.txt` manifest refers to these files and their checksums. In contrast, any files only referred to using `Ref` directives are not listed in the BagIt manifest and are instead described in the BagIt's `fetch.txt` as remote resources.

`@Data_Primary` and `@Data_Copy` both refer to resources that have been packaged with the MEDFORD metadata file, and should be available from the bag in a self-contained fashion, without having to visit external sources. From the point of view of the bag itself, there is no difference between these two tags; the difference is based on user context: `@Data_Primary` means that the BagIt bag is considered to be the primary and authoritative source for the data or resource; `@Data_Copy` means that BagIt has placed a copy of the data or resource into the bag, but that it does not claim the primary role. Finally `@Data_Ref` refers to DOIs, URLs, or other pointers to data or resources that are *not* placed in the bag, but rather represent external databases or resources.

Two example use cases are provided in Figure 2.1 and Figure 2.2.

A specification document for the MEDFORD language is available at `https://github.com/TuftsBCB/MEDFORD-Spec`.

Researchers wish to store all the necessary data and programs necessarily to replicate their RNAseq analysis. They are the owners/collectors of the raw transcriptomic data and perform data clean-up using a couple of small home-grown scripts to filter out bad reads. They then complete their downstream analysis using several popular software packages, including STAR and DESEQ2. They then used a novel dimension reduction package called SQUISHSEE from other researchers to visualize their results.
They elect to include their transcriptomic data and homegrown scripts in the bag, and use `@Data_Primary` and `@Code_Primary` tags to reference them. The `@Code_Primary` tag is not appropriate for STAR, DESEQ2 and SQUISHSEE, since they do not own or maintain these resources; they need to decide whether to use `@Code_Copy` and place a copy of these resources into the bag, or not, in which case they would use instead the `@Code_Ref` tag. In this case, because STAR and DESEQ2 are well-maintained and supported standard packages, they elect `@Code_Ref`, and don't include a copy of the code in the bag.
On the other hand SQUISHSEE is only used by a handful of researchers, and they worry about its longevity. Thus they also put a copy of the version of SQUISHSEE they are using in the bag, with a `@Code_Copy` tag. Later, when DESEQ2's new update uses a library that is not completely standard, they update the bag and decide to put a copy of the old version of DESEQ2 into the bag, just in case.

Figure 2.2: Example use case of the MEDFORD `_Primary`, `_Copy`, and `_Ref` tags.

2.3 Reusability

2.3.1 Tag Extensibility

MEDFORD has a set of pre-defined major and minor tags that it uses for conversion into various other formats, but if a user cannot find a tag that they believe suits the metadata that they are storing, they can simply define one of their own without any additional overhead. All the user must do is use it as if it were already defined, and the data and its structure will be read by the MEDFORD parser. Any novel tags defined this way will be treated as `*-Unstructured` tags, and not validated, though they will persist across copies of the MEDFORD file. This provides a dynamic aspect whereby any model created or adjusted to include the new user-defined tag could be output to any secondary formats without any changes in MEDFORD structure.

2.3.2 MEDFORD Templates

Due to the simple plaintext structure of a MEDFORD file, it is easy to create templates. A template is a MEDFORD file that is partially filled, saved, copied,

```
@Species Pocillopora damicornis
@Species-Loc Sabago Isthmus, Panama
@Species-ReefCollection 06/12/20
@Species-Cultured University of Miami Coral Resource Facility
@Species-CultureCollection 06/21/20
```

Figure 2.3: Example MEDFORD template describing the metadata of a coral species.

```
@Species Pocillopora damicornis
@Species-Loc Sabago Isthmus, Panama
@Species-ReefCollection [..]
@Species-Cultured University of Miami Coral Resource Facility
@Species-CultureCollection [..]
```

Figure 2.4: An example MEDFORD template describing the metadata of a coral species, utilizing the template placeholder `[..]` in place of metadata that is expected to vary.

and then re-used. For example, a lab may template a list of contributors and funding sources and when an individual needs to create a MEDFORD file they simply create a copy of this MEDFORD file (and change contributor roles as necessary) before filling in the rest of the file. MEDFORD files describing similar data may also be re-used like this.

For example, consider a researcher who commonly works on one species of coral, such as *Pocillopora damicornis*. The researcher could use a MEDFORD template with the commonly-used tags filled in, shown in Figure 2.3

For further reuse, the researcher may also include MEDFORD's "invalid value" token, which can be used to force users of a template to fill it out with complete information. The `medford` parser will require the user to fill in the specific placeholders (`[..]`) prior to validation, throwing a unique error type. This eliminates the possibility that a researcher could accidentally leave a value for an older version of the template, further error-proofing MEDFORD templates. The same template, but using these reserved template tokens, is shown in Figure 2.4.

```
‘@myinstitute 100 Institute Drive, State, Zip

@Contributor John Doe
@Contributor-Role Author
@Contributor-Association ‘@myinstitute
```

Figure 2.5: Example of a MEDFORD macro being defined and used.

2.3.3 MEDFORD Macros

To further alleviate the workload placed on researchers to document their work, MEDFORD includes the concept of a macro. Similar to a variable defined in BASH, a macro is a string name that is directly replaced with another, longer string.

In the initial version of MEDFORD, a macro is defined by specifying a backtick (‘), `@`, a one-word name, and the macro body (which can contain multiple lines, ending at the next reserved word, which could begin another macro definition or could be a regular tag). An example is shown in Figure 2.5. In future versions of MEDFORD, this syntax will be updated to separate syntax between macro definition and macro usage.

2.3.4 Backend Extensibility

Many other formats are simple to add to the MEDFORD parser. For example, one may wish to submit their data to a database such as BCO-DMO [CGK+16], which requires an RTF (rich text format) file structure with unique content requirements. For example, for a data submission, BCO-DMO requires at least some form of identification of which `@Expedition` the samples were collected on. This identification, may be either some combination of `ShipName` or `CruiseID`, etc. In defining a backend translation to BCO-DMO for MEDFORD, the MEDFORD parser can ensure this is upheld. This ability to act as an intermediary allows for a lab to write a single MEDFORD file to describe their research and export it to a multitude of different formats.

Similarly to BCO-DMO, other formats can be added to the MEDFORD parser easily. It is worth noting that metadata associated with user-defined tags will

```
@Method Illumina HiSeq2500
@Method-Type Sequencing
@Method-Company Dovetail Genomics, Santa Cruz, CA,
    USA
@Method-Sample Healthy
@Method-Note Chicago libraries, more sensitive to
    DNA size

@Code_Ref HiRise
@Code_Ref-Type Assembly of genome scaffolds

@Code_Ref BLAST
@Code_Ref-Type Identify and remove scaffolds of
    non-coral origin
@Code_Ref-Note Searched against databases from
    Symbiodiniaceae, Bacteria, and viruses
```

Figure 2.6: Example of a MEDFORD file describing a coral genomic sequencing experiment.

not be parsed but simply passed along verbatim. For instance, the `@Image-Coverage` tag in the coral image data example below does not specify any units; if this tag were expected by some destination format, units might be assumed by convention, but would otherwise be left to the user (the user could also specify units in plain text in the metadata, e.g. "6.2 degrees").

2.4 MEDFORD Implementation

2.4.1 MEDFORD Parser

The MEDFORD parser, known as `medford`, essentially has two roles. First, it validates the syntax and structure of a provided MEDFORD file as described earlier. Additionally, the parser validates the content of a provided MEDFORD file, such as ensuring date fields are in the correct datetime format. In the future, we plan to support further validation of specific metadata, such as ORCID, geographic coordinates, and grant numbers from various funding agencies. The purpose of validation is to ensure that the file is written in correct MEDFORD format, including major and minor tags, and that each tag is being applied to some data. For example, a

```
@Image 05-01-19_Image3
@Image-Date 2019-05-01T19:20:30.45
@Image-Site LTER 4
@Image-Habitat Outer 10m
@Image-Pole 3-4
@Image-Quadrant 4
@Image-Coral Acropora
@Image-Coverage  6.2

...

@Taxonomy Cnidaria
@Taxonomy-Type Phylum

@Taxonomy Anthozoa
@Taxonomy-Type Class
@Taxonomy-Parent Cnidaria

...

@Region LTER 1 polygon including
    LTER 0 on north shore
@Region-NorthernCoord -17.47
@Region-SouthernCoord -17.49
```

Figure 2.7: Snippet of an example of a MEDFORD file that describes photos taken of a coral alongside taxonomy and region information.

```
@Code_Ref MEDFORD Source Repo
@Code_Ref-Version 1.0
@Code_Ref-URI https://github.com/TuftsBCB/medford
@Code_Ref-Type GitHub
@Code_Ref-Language Python
@Code_Ref-OS Linux MacOS
```

Figure 2.8: Example of a MEDFORD block describing code used as part of the research project. The minor tags OS and Language are novel (not expected by the MEDFORD parser), and thus will be kept but not validated.

user cannot describe an ORCID without having some Contributor name with which to associate it. We note that the current MEDFORD specification is silent as to whether or not to preserve the ordering of tags (for example `@Contributor` does not specify an author order and relies on `@Contributor-Role` to indicate significance). However, the `medford` implementation will preserve order of tags in a future version.

The second role of the `medford` parser is to optionally compile an input MEDFORD file into some destination format. The current `medford` parser specializes in translating a MEDFORD file into a Bag; the `medford` parser can gather all the files referenced in a given MEDFORD file, and creates a Bag following all BagIt specifications. This Bag can then be used to transfer all of the metadata and data of a research effort. The current plans include to add additional output types in the future, such as RDF.

The `medford` parser is written in Python (3.8), relying on the Pydantic parsing module to validate the MEDFORD syntax and structure.

Due to the amount of control a compiler has to have over the input, creating a parser normally causes the vocabulary to become extremely defined and controlled. The `medford` parser, however, was developed specifically to avoid restricting the acceptable vocabulary. While the parser can only validate major and minor tags it is aware of, it will not break on novel inputs, shown in Figure 2.8.

Importantly, the `medford` parser is specifically developed such that the syntactical parsing logic is entirely separate from the vocabulary definition. Given a desire to begin validating the contents of a novel tag, a user can easily add their

own validation without having to interact with the parsing logic. All vocabulary validation definitions are stored entirely independently of the parsing logic, and can be edited with minimal consequences. In this `@Code_Ref` example, a user could implement the validation for the `OS` minor tag to ensure it is some combination of of `Windows`, `MacOS`, and `Linux`.

Given that the `medford` parser is open-source, a research group may add validation to their local copy of the MEDFORD parser without needing to interact with other groups, though we will be welcoming any and all pull requests to add validation that users feel is missing.

2.4.2 Error Handling

MEDFORD errors come in three major forms:

- **Syntax Errors**: errors in the MEDFORD formatting in the provided file, such as multiple uses of the same macro name.
- **Validation Errors**: errors in the content or format of metadata provided for known major-minor tag combinations. For instance, a `@Date` field that does not contain a valid datetime string, or a `@Contributor-ORCID` field whose ORCID is not valid would both constitute validation errors.
- **Missing Data Errors**: Required fields are missing, such as `@Contributor` major tag that has been marked as a `Corresponding Author` without a corresponding `@Contributor-Email` minor tag.

All three types of errors are errors that a standard user is expected to encounter during use, especially during first-time use or novel data type description. Special care has been invested in ensuring these errors will be as human-legible as possible.

For all expected errors, `medford` provides an error text to the user that contains the following information: the line number where the error was encountered, the major-minor tags involved at that line, and an error text description.

Two examples of `medford`'s error messages and their improvements are shown below.

First, a standard Pydantic error contains information that is highly specific to the backend implementation of `medford` parser and irrelevant for standard use.

```
Contributor -> 0 -> 1 -> __root__
  Corresponding Authors must have a provided validated email
  (type=value_error.incomplete_data_error)
```

This has been improved with the addition of the line number in which the error appeared in the `MEDFORD` file, and removal of the implementation-specific array indices.

```
Line 1 : @Contributor has incomplete information:
  Corresponding Authors must have a provided email.
```

Secondly, some major and minor token combinations may have multiple valid formats, and Pydantic's standard error handling will throw a unique error for each failed format validation. The `medford` parser automatically consolidates these errors into a singular error for legibility.

2.5 Availability of `medford` and the Coral RNAseq collection

The `medford` parser is open-source and available under the MIT license at: `https://github.com/TuftsBCB/medford` as well as via `PyPi` as the package `medford`, so it can be installed by invoking `pip install medford`. A specification for the MEDFORD language is available at `https://github.com/TuftsBCB/MEDFORD-Spec`.

Some example files are provided in the parser directory, however, a separate repository is also in development for a larger collection of example MEDFORD files. This repository is available on GitHub at: `https://github.com/TuftsBCB/MEDFORD-examples`. This repository is a collection of primarily Coral RNA-Seq

experiments. This repository also contains partial MEDFORD files for use as templates.

2.6 Discussion and Future Work

This chapter presents MEDFORD, a lightweight metadata format initially targeted at coral reef research data, intended to be easy for researchers without programming expertise to create and maintain. Initially supporting the FAIR principles [WDA+16] of interoperability and reuse, MEDFORD aims to support all FAIR principles.

Currently, MEDFORD relies on editing ASCII or UTF-8 text, but future work will extend the `medford` parser to also be able to extract text content from Microsoft Word files.

One possible critique of MEDFORD is the variety of possible tags. For instance, it may be challenging for a user to remember whether the needed tag is `@Contributor-Association`, `@Contributor-Institution`, or `@Contributor-Location`. A rich template library can mitigate this, by providing examples that a user can simply fill in. A searchable template library portal (similar to LaTeX's CTAN) would enable users to find applicable templates as the template ecosystem grows. In the future, support for the Language Server Protocol [BK19] will allow a user of any compatible text editor to get intelligent suggestions and autocompletions for common tags, even in an offline environment such as at sea. This will also mitigate the likelihood of minor typographical errors in tags causing them to be unrecognized. To further mitigate the likelihood of typographical errors, the `medford` implementation will reject a minor token without an accompanying major token. Further user testing and feedback will result in further enhancements to the MEDFORD language and the `medford` parser implementation.

As a consequence of the MEDFORD parser's compilation use, MEDFORD files have a lifecycle. There are raw, un-validated MEDFORD files, there are validated MEDFORD files, and finally there are MEDFORD files that have been com-

piled (such as in the case of a BagIt compilation). The difference between these is a critical. A researcher needs to know whether or not a MEDFORD file has been validated before they try to submit it to a database.

A major future goal will be output of RDF and support for linked open data. We hope to add the ability to translate a MEDFORD file (and created bag, if applicable) into an RDF, as well as the data-1 compliance this involves.

An unsolved problem is how to handle multiple authors, and conflicting claims of ownership. While there is nothing preventing a MEDFORD file from being passed between collaborators, keeping track of changes is a challenge. How can one researcher be certain that they are editting the most recent version of a MEDFORD file? Perhaps even two researchers are editing their own copies of the same MEDFORD file. Technically both are the most up-to-date in their own facet: one researcher added the coral sample metadata while another added the sequencing pipeline metadata. There exist some solutions to this in external tools such as GitHub, but is it viable to ask MEDFORD adopters to use these tools?

One solution we are in the process of considering for a future version of MEDFORD is to implement the concept of the `include` directive. Rather than restricting a MEDFORD file to a single file, include directives will enable users to work in separate, smaller files that will automatically be combined by the MEDFORD parser. This allows each MEDFORD file to be dedicated to a specific portion of the research project, such as one file for coral sample metadata and another for sequencing pipeline metadata. This partially solves the multiple authorship problem, as each author can be held responsible ensuring all collaborators have the most up-to-date version of the metadata they are authoring.

The "R" in MEDFORD currently represents "reef" as our initial application domain has been coral reef data. However it stands to reason that the "R" might represent "research" in the future.

Chapter 3

Designing a Domain-Specific Language for Expressing Metadata Requirements

3.1 Introduction

In previous chapters, we discussed the development of the MEDFORD language. MEDFORD supports FAIR (Findable, Accessible, Interoperable, and Reusable) research by attempting to circumvent the primary issue of annotating computational biology pipelines: the inherent complexity and difficulty that arises from having a machine-parseable language [WDA+16]. By allowing researchers to define novel metadata types, content, and comments when annotating their data, MEDFORD ensures that researchers can note any metadata they consider important at production time rather than after research completion. Then, when the researcher submits or shares their data, the MEDFORD language is structured enough to allow for the development of the Medford parser that can validate the file for the absence or mis-formatting of required metadata, as well as provide explanations on how to resolve these issues.

```
@Contributor Jane Doe
```

```
@Contributor-Role First Author

@Contributor-Association Education University, 415 College Lane, USA
```

As described in Chapter 2, the key structure of MEDFORD is that each line is broken into three parts: a major token, which defines the object being described - in this case, a Contributor to the research project. Optionally following the major token is a minor token, which defines the aspect of the object being described. For example, above, we describe the name (no minor token), the role, and the association of the object defined by the major token, the Contributor. Finally, the final part is the body, or the actual metadata, such as the Role of the Contributor as the First Author.

In essence, MEDFORD is taking an object-oriented approach to metadata by allowing users to define arbitrary metadata Objects and arbitrary metadata Attributes. These objects are validated by the Medford parser and any structural errors are corrected by the author. Then, the Medford parser can either compress the author's research into a BagIt bag [KLM+18], or translate the metadata into a format suitable for database submission, such as for submission to BCO-DMO [BO].

3.2 Motivation

Contrary to the design philosophy of MEDFORD where it aims to be as flexible as possible, in order for MEDFORD to support reusability, it must be capable of enforcing metadata requirements. While flexibility promotes authorship of metadata during research, there must be some process to ensure this metadata contains important information in the correct format. Consider two instances where users would need to define such requirements: first, as a data curator, and secondly as a researcher.

Every database has its own expectations of input data. These expectations are currently often upheld by means of manual labor: specialized data custodians are hired to communicate with researchers after submission, manually updating the input metadata such that it aligns with their database's vocabulary and format-

ting requirements. This process is extremely expensive and time-consuming and at times prone to errors, as data custodians are not capable of being specialists in all of the different kinds of metadata being submitted. On the other hand, if these expectations could be translated such that the Medford parser could automatically search for the presence, absence, and formatting for these requirements, the amount of human hours required to submit data could be significantly decreased.

Secondly, researchers themselves may have an agreed-upon set of expectations of their data. Thus, a given research field may want to make these expectations concrete by turning them into a model expected by MEDFORD. In addition to helping researchers pick major and minor tokens to describe their data, this will also allow researchers to add validation to ensure the field is consistently annotated across projects.

Currently, expectations of metadata are hard-coded into the Medford parser. While allowing for complete flexibility of data requirements and enabling complicated validation, this methodology is unsustainable. Not only must a researcher or data curator have a working understanding of writing Python functions and classes to edit the Medford parser itself, but they must also familiarize themselves with the particular nuances and libraries that are utilized within the parser. These custom validation functions will also only work for that one particular version of the Medford parser; if another implementation becomes popular (such as one written in Rust), researchers and data curators would have to re-write all of their validation to work as part of the new parser.

Finally, there comes the issue of users editing the parser source code. While the parser is open source and the MEDFORD research team fully promotes community development of the parser, requiring all validation to be implemented as edits to the parser paves the way for a significant amount of edit collisions and accidental data overlap. If a research group writes validation that they intend to only use amongst themselves, when an update to the Medford parser is released, it may accidentally overwrite the validation they have written as the source code gets updated. Furthermore, if two research areas have an overlap in vocabulary with the

same term having two entirely different meanings, despite both forms of validation being correct and having widespread use, they cannot both exist in the Medford parser at the same time due to a namespace collision.

Therefore, it is clear that there must be some method for researchers to define custom validation outside of the scope of the Medford parser itself. Allowing users to define custom validation as external files that may then be passed as input to the parser alongside the MEDFORD file itself allows for flexibility and interoperability far better than what would previously be available.

3.3 Design Considerations

As this domain-specific language will be aimed at both data custodians and research scientists, it must be similar in simplicity to the MEDFORD language itself. In the first iteration, we only require the following capabilities:

1. To specify a novel expected major token.
2. To specify a minor token as an attribute of a new major token.
3. To specify whether a specified minor token is required or optional.
4. To specify a minor token as one of a small set of types, including str, int, bool, float, and datetime, as will be declared explicitly in the RFC.
5. To specify uni-directional dependencies based on the presence or absence of another minor token.

A current proposal for how the grammar may look like in use is as follows:

```
Major
- Minor1 str Required
    + len < 5
    + [1] = "A"
- Minor2 int Optional
```

```
    + val > 10
    + val < 30
- Minor3 int Optional (Minor2)
```

This syntax defines three types of lines: a major token definition line, a minor token definition line, and a validation line. The whitespace at the beginning of the line is ignored, such that the user may use tabs, spaces, and empty lines to visually group conceptually similar tokens for their own understanding. Space characters within the line are considered argument separators, as all names and parameters otherwise do not allow for spaces as part of their content, except for `dependencies`, which will be described later and are visually grouped using parenthesis.

3.3.1 Major Token Definition Lines

Major token lines are any lines that do not begin with one of the otherwise defined special characters (`-` and `+`). Their first and only content is a series of alphabetic characters that act as the novel major token name being defined, following the MEDFORD major token name requirements.

3.3.2 Minor Token Definition Lines

A line that begins with the reserved character `-` represents minor token definition lines. Minor token lines can only be used after at least one major token line has previously been defined, as a minor token is always defined as a feature of a specific major token. A minor token line contains the components:

```
- Name type requirement (dependencies)
```

`Name` refers to a series of alphabetic characters representing the name of the minor token, following the MEDFORD minor token naming requirements.

The field `type` refers to which one of the pre-defined types the minor token content expects, defaulting to `str` when unrecognized. The domain-specific language will be developed with the pre-defined types `str`, `int`, and `DateTime`.

The field `requirement` declares whether or not the minor token is required; that is, should the parser accept any Major tokens without this minor token declared (`Optional`), or if the parser should throw an error (`Required`). This field is required, but `Required` may be shortened to `Req` or `R`, and `Optional` may be shortened to `Opt` or `O`.

Finally, the `(dependencies)` parameter allows authors to specify a list of minor token names that this minor token depends on. Depending on the logic specified in this list, some combination of those tokens means that this minor token must be specified. This list of minor tokens may contain alphabetic names, the parenthesis symbols, the ampersand symbol (`&`), and the pipe symbol (`|`). A pipe represents the logical `OR`, while the ampersand represents the logical `AND`. These operators follow the standard associativity and precedence found in mainstream programming languages.

3.3.3 Validation Lines

Any line that begins with the reserved character `+` is considered a Validation Line, which defines some validation that should be performed on a minor token's content.

A Validation Line must directly follow either a Minor Token Definition Line or another Validation line. If a Validation Line follows another Validation line, it affects the same Minor Token Definition Line that the Validation Line does.

Validation Lines have a limited set of reserved words that can be used to represent validation, depending on the `type` of the affected Minor Token Definition Line.

Minor Token Definition Lines with type `str` have the reserved keyword `len` that represents the total length of the string, and the reserved syntax `[n]` that represents the content of the `n`th character of the string. There is also the expanded reserved syntax `[x-y]`, which refers to the content of the string from `x` (inclusive) to `y` (exclusive). This exclusivity is chosen to match Python syntax. The reserved keyword `len` can be used with the mathematical operators `>,<,=`, such that a user can impose restrictions on the length of a minor token string. The character and

substring syntax can only be used with the operator `=`, such that a user can impose restrictions on the content of a string, such as the presence of a specific leading or tailing character or a substring present at an exact location in the string.

Examples of `str` Validation Lines are as follows:

```
Major
 - Minor str Req ()
  + len > 3
  + [0-2] = AB
```

For the type `int`, there is the reserved keyword `val`, which refers to the integer value of the minor token. The mathematical operators `>,<,=` may be used.

An `int` Validation Line follows the pattern:

```
Major
 - Minor int Req ()
  + val > 2
  + val < 5
```

These requirements may also be chained up to twice for brevity, such as below:

```
Major
 - Minor int Req ()
  + 2 < val < 5
```

3.4 Future Work

We are currently in the process of formalizing the requirements of the MEDFORD domain-specific language and grammar as an RFC. In addition, we are developing a parser that will export a parsed JSON file containing a schematic of the defined major and minor tokens and their content requirements. Once this JSON parsing is functional, we will begin authoring a pull request to the Medford parser to accept

definitions in the domain-specific language and write specifications for some of the databases that our collaborators currently use, such as BCO-DMO [BO].

In the future, we may also consider methods to allow for more complex type and dependency validation. However, there is inherently a limit in complexity before the domain-specific language becomes overly complex and inaccessible to researchers.

Chapter 4

Implementing Metadata References in MEDFORD

4.1 Introduction

MEDFORD was designed as a middleware that is simultaneously human-writable and machine-readable while also having been built on top of the BagIt [KLM+18] standard to allow users to package their metadata alongside the research data it describes. Consequently, the MEDFORD language assumes that all relevant metadata would be described in its entirety within a single file and that every object could be described independently from all other research objects. For example, although a photo may contain a particular coral species, a researcher using MEDFORD to annotate their metadata cannot easily link the photo file's metadata to the metadata they transcribed of the coral species.

This behavior is insufficient to describe all the research data MEDFORD is meant to handle. In this chapter, we extend the MEDFORD language in two ways: First, we allow metadata objects to reference other metadata objects. Second, we implement externally defined metadata in MEDFORD and define syntax such that a researcher may reference an external MEDFORD file's metadata contents. This will improve the usability of MEDFORD and promote adoption, as researchers

may describe more complex metadata using cross-object referencing and will not be required to repeat existing metadata described in other MEDFORD files.

4.2 Metadata Inter-connectivity

Initially, MEDFORD was designed to be entirely composed of independent "blocks", where each block represents an object in its entirety. Each minor token line represents an attribute of this object such that the collection of all lines forms a complete description of a single object. However, this independence prevents MEDFORD from being sufficient to describe real research data. Research data is inherently highly interconnected, with certain metadata object descriptions only being complete with context from other metadata objects.

A prime example of this is in the relationship between coral reefs and coral species. Coral species require a description of information such as their scientific name, the current genome construction, and the reef at which the species was studied. Reefs require information such as geo-coordinates, the date of study, and the coral species present in the reef. There is a strong dependency between the description of a coral species and a reef: to completely describe one; there must be some description of the other.

Consider the perspective of a marine biologist interested in the differences between coral species' responses to sediment stress. To these researchers, the coral species are of utmost importance and the reef where the coral was studied is considered an attribute of the species, as shown in Figure 4.1.

Now consider instead a marine biologist interested in the differences in the recovery rate of coral communities at various reefs. To these researchers, the reefs are the primary object being studied and the species present are simply attributes of the reef, as shown in Figure 4.2.

In both of these examples, the researchers had to describe an object as an attribute of the other. In the first, `Reef` became an attribute of `Species`, and in the other, `Species` became an attribute of `Reef`. This is suboptimal – both `Species` and

```
@Species Montastraea cavernosa
@Species-Genome eLife 2016 20-Coral Comparative
@Species_Reef Key West Nursery
@Species_Reef-Location Florida Keys National Marine Sanctuary

@Species Acropora cervicornis
@Species-Genome [..]
@Species_Reef Key West Nursery
@Species_Reef-Location Florida Keys National Marine Sanctuary

@Species Orbicella faveolata
@Species-Genome [..]
@Species_Reef Key West Nursery
@Species_Reef-Location Florida Keys National Marine Sanctuary
```

Figure 4.1: Example of a MEDFORD file as written from the perspective of researchers exploring the differences across coral species. Note how each species describes the same reef, causing repetition of its metadata.

```
@Reef Reef1
@Reef_Coordinates [..]
@Reef_Species Montastraea cavernosa
@Reef_Species-Genome eLife 2016 20-Coral Comparative
@Reef_Species Acropora cervicornis
@Reef_Species-Genome [..]

@Reef Reef2
@Reef_Coordinates [..]
@Reef_Species Montastraea cavernosa
@Reef_Species-Genome eLife 2016 20-Coral Comparative
@Reef_Species Orbicella faveolata
@Reef_Species-Genome [..]
```

Figure 4.2: Example of a MEDFORD file as written from the perspective of researchers exploring the differences across reefs. Note how each reef describes the species present, causing a repetition of metadata.

`Reef` are complete objects that were involved in the research project with various attributes that completely describe them. The solution to make one an attribute of another blurs the line between attributes and objects.

This blurring causes two major issues. First, it forces a hierarchy which cannot be kept consistent between all possible research projects. Consider now trying to submit these examples to the same database: how can the database deconvolute `Reef` and `Species` data from both of these examples at the same time? Either the database would have to choose a "correct" representation that would require a researcher to re-organize their metadata in a way that no longer aligns with their research priorities, or MEDFORD would have to support automatically deconvoluting hierarchical object descriptions automatically.

Secondly, because one of the objects is being treated like an attribute, its metadata is getting repeated. As shown in Figure 4.1, the `Reef` metadata is repeated in every `Species` that references that can be found in that reef. In Figure 4.2, the `Species` metadata is repeated in every `Reef` that contains that coral species, increasing the possibility of a mistake when inputting the metadata of the secondary object. Even if the MEDFORD parser was improved to automatically deconvolute these hierarchies, it still must find a way to resolve conflicts in these repeats.

Therefore, there must be a concept of relationships between metadata objects that is non-repetitive and non-hierarchical.

4.3 Naming Metadata Objects

The first issue with implementing inter-connectivity between metadata objects in MEDFORD is a lack of identifiers. MEDFORD initially had no way to give a unique identifier to a metadata object, as there was never the necessity to reference a specific metadata object. However, consider the valid MEDFORD text:

```
@Species P. Dam
@Species-Construction Pocillopora damicornis genome v1.0
```

```
@Species Species1
@Species-Construction [..]
@Species-Reef @Reef Reef1

@Reef Reef1
@Reef-Coordinates
@Reef-Species @Species Species1
@Reef-Species @Species Species2
```

Figure 4.3: A syntax proposal for referencing MEDFORD objects from other objects. Species1 references Reef1 as its origin reef, while Reef1 references coral species Species1 and Species2 as species that could be found in that reef.

The first line is colloquially called the "description" or `desc` line of the metadata object in MEDFORD documentation. This line is parsed as an arbitrary text description line that is stored as-is in the internal representation of the MEDFORD file and otherwise ignored, meant to be a short description for authors to use for their own reference. This overlaps with the functionality of the `note` minor token, which allows users to arbitrarily annotate a given metadata object with text that will not be processed by the MEDFORD parser, only kept as an annotation for readers. Every block must have a `desc` line for the file to be considered valid MEDFORD.

Therefore, the solution is obvious: the `desc` line can be repurposed into a `name` line. Instead of being a freestyle annotation, the first line of a MEDFORD block will be considered its name. Functionally, this is equivalent – a `name` can be any combination of ASCII characters and may include standard MEDFORD macros, just like a `desc`, but the new name provides more clarity on the line's functionality. It prepares the concept of a name for the purposes of cross-object referencing. In the above example, MEDFORD has the concept of a `@Species` object that is `name`d P. dam. With a name defined, we may now propose a syntax, as shown in Figure 4.3.

4.3.1 Object Name Collisions

A major difference between the concept of a `name` and a `desc` is quickly apparent: since a `name` must refer to a single object for inter-object dependencies to be re-

solvable, `name` lines must be unique. At first, this appears to possibly add a large burden on the user. Consider the specific case where a researcher is studying two corals and has a singular photo of each coral species.

```
@Species P.Acuta
@Species-Construction NCBI BioProject PRJNA812628

@Photo P.Acuta
@Photo-Type JPEG
```

If a researcher would like to describe that a `@Reef` contains the coral species P.Acuta, it appears that there would be a name collision. However, MEDFORD's structure offers an easy solution: when a researcher writes a MEDFORD block, that block is immediately associated with a major token. The major token defines the type of object being declared and if a researcher is referencing a block, they have some expectation of the type of metadata they are referencing. Therefore, the syntax should require the user to declare the major token:

```
@Reef-Species @Species P.Acuta
```

Therefore, name uniqueness in MEDFORD is only required for blocks that share the major token. That is, if the user was to describe a `@Species`, its name must only be unique compared to the other `@Species`. We believe that this is a reasonable expectation for users and would be good practice even if the `name` attribute was not being actively used in MEDFORD logic for readers to be able to quickly differentiate between objects of the same major token. Furthermore, if an author uses an interactive development environment with MEDFORD tooling, the tooling can warn on blocks with repeated names.

It is important to note that the example above separates the minor token from the major token being referenced. This is intentional as it is not guaranteed that the vocabulary used for the minor token and major token are identical. Consider describing a piece of software written by a particular contributor, shown in

```
@Contributor John Doe
@Contributor-Association Institute, Street, City, ZIP

# Proposition 1
@Software Regression Library
@Software-@Contributor John Doe
@Software-Language Python

# Proposition 2
@Software Regression Library
@Software-Developer @Contributor John Doe
@Software-Language Python
```

Figure 4.4: Two propositions for the syntax to reference another metadata object in MEDFORD. In proposition 1, the minor token is assumed to be the major token of the referenced metadata object. In proposition 2, the major token of the referenced metadata object is explicitly defined. Proposition 2 allows for clarity in the attribute of the object described – in this case, it is clear that the contributor John Doe was responsible for developing the library.

Figure 4.4. Assuming the minor token matches the major token of the object being described forces the authors to use a particular minor token which may or may not be informative of the relationship between the two objects. By instead requiring the author to explicitly declare the major token being referenced, MEDFORD's standards of flexibility are maintained and an informative minor token may be used.

4.4 Referencing Metadata from Other Sources

In developing the syntax for internal referencing, other circumstances where a researcher would want to reference data quickly became apparent. The proposed reference syntax is developed solely to reference metadata within the file a researcher is editing, but consider the case where multiple research groups are studying the same set of coral species. It is important to maintain consistent annotation of these coral species across these research groups, but currently, this metadata must be manually copied from the reference MEDFORD file into the researchers' own files. We decided to extend our proposed reference functionality to add the capability to reference metadata stored in external MEDFORD files.

This requires two more additions to the MEDFORD syntax: the capability to declare that referenced metadata exists in an external file, and the capability to declare the location of an external file that will be referenced.

There have been a myriad of approaches to the concept of declaring that an object is imported from an external source. Much like how a user may either use `Import module` or `From module import Function` in Python, MEDFORD must either completely import the external metadata and place all of its objects into the current environment, or allow the user to import particular metadata objects into the current context.

Considering how often a user may want to reference external files, and the relative small size of metadata objects in MEDFORD, we have decided to implement the former. While parsers may optimize to only import relevant metadata for efficiency, requiring researchers to manually import every referenced metadata object would be greatly detrimental to the efficacy of MEDFORD as a user-friendly object and increase the possibility of user error.

However, this causes a new problem: if two separate MEDFORD files both define a `@Species P. dam`, but one references an older genome construction than the other, the MEDFORD language must have a way to deconvolute sources to avoid placing the responsibility on parsers to determine how to handle collisions. This is resolved by a feature we have just introduced: every object in MEDFORD must have a `Name`, and therefore, the syntax to define an external MEDFORD file reference must also include a `Name`.

4.4.1 External Reference Syntax

We now propose concrete syntax for the two required additions described above. First, the syntax to declare that an attribute is from an external reference is shown below. We imitated Python import syntax, as its plain English syntax made it easier to understand from a quick glance:

```
@Reef Reef1
```

```
@Reef-Species from CoralsMFD: @Species P.Dam
```

Even without experience using Python or MEDFORD, a new reader could glean that `@Species P.Dam` comes from some object named `CoralsMFD`.

Next, to provide this nickname `CoralsMFD` as some link to an external source, we propose the following syntax to declare an external MEDFORD metadata source. The example below declares that MEDFORD metadata should be imported from the file `corals_metadata.mfd` to be available for reference anywhere in the MEDFORD file using the nickname `CoralsMFD`.

```
@Import CoralsMFD
@Import-File ~/shared/corals_metadata.mfd
```

4.5 Conclusion and Future Work

We have proposed an addition to the MEDFORD language that enables metadata objects to reference other objects, such that research data that is intrinsically linked can be accurately documented in MEDFORD. We have also proposed syntax for MEDFORD files to be able to reference other MEDFORD files, such that metadata can be re-used. We intend to implement both of these functionalities as an extension to the current MEDFORD parser and submit a pull request to add this functionality to the official release.

We believe this syntax can be extended to even more powerful functionality. For example, photos taken by modern digital cameras automatically store metadata such as the date and GPS location of the photo. Researchers should not be expected to repeat this metadata in the MEDFORD file by hand. Theoretically, this could be implemented using nearly identical syntax: simply point the `Import` block at the external metadata file. However, while MEDFORD files can easily be referenced since any MEDFORD parser is expected to be able to parse MEDFORD, the same cannot be said for external metadata formats such as EXIF. Therefore, once internal and external MEDFORD referencing has been implemented and tested, we

intend to research methods of adding universal plug-ins to MEDFORD parsers to allow researchers to reference external formats the same way they reference external MEDFORD files.

Chapter 5

Consolidating Metadata in High Performance Compute Environments

5.1 Introduction

Previous chapters covered the development of MEDFORD, a metadata description language aimed towards assisting researchers in storing their research metadata. In this chapter, we discuss the development of a suite of metadata collection tools geared towards higher complexity compute environments, specifically research performed using high performance compute (HPC) systems. Scientific research utilizing HPC systems increasingly uses workflows to launch simulations, communicate with schedulers and automate resource allocation. These workflows strive to take advantage of the multitude of resources available to them on these powerful compute systems: tasks may be massively parallelized, memory-intensive scripts may run without restriction, and complex mathematical calculations can be offloaded onto GPUs. Under the latest generation of systems, the runtime environments of these workflows have become extremely heterogeneous. A given workflow can be run on many different types of hardware at once, different workflows require different levels

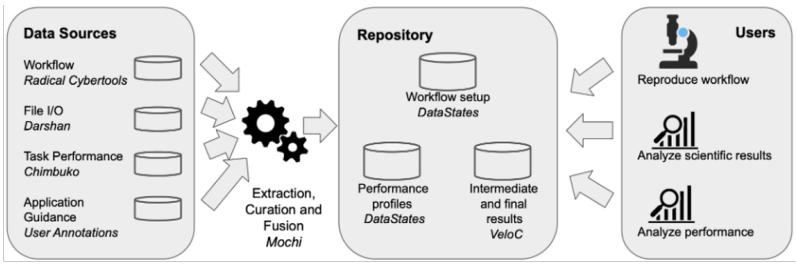

Figure 5.1: Overview of several components that comprise a RECUP pipeline to collect data and metadata about the performance of a workflow run on the Polaris system. Data and metadata is collected within the data sources, extracted and combined, and then stored in data repositories. Researchers then study the data from the repositories.

of parallelization, and even the data accessed by the workflows themselves could be spread across different file systems. This heterogeneity increases the complexity of metadata annotations for a given workflow, much of the metadata that can be captured is outside a researcher's control, and the processes to capture and establish the provenance of workflow execution are themselves not unified. They may even be missing.

We offer an approach from the software perspective to ensure that metadata is collected throughout the life cycle of a workflow running on a US DOE Leadership Computing Facility (LCF). To illustrate our approach we take the example of the RECUP[1] workflows, executed on the Polaris system hosted by Argonne National Laboratory (Fig. 5.1) [NIR+23].

Importantly, these workflows generate massive amounts of metadata. While MEDFORD was developed to support researcher authorship of metadata, the scale of metadata in this use case is too extreme to leave as a task for the researchers. Furthermore, while many of these pipelines are developed specifically for HPC, some of the metadata that determines their performance may require extensive HPC knowledge that falls out of the domain of expertise of researchers operating on these systems. Therefore, metadata must be automatically generated and collected by the HPC workflows while the payload application is running.

[1]https://sites.google.com/view/recup-reproducibility/home
https://github.com/RECUP-DOE/

To collect metadata from a running workflow system, there are two approaches: collect all the metadata using a singular collection software or collect the metadata individually at their source using multiple specialized scripts. The first approach is straightforward and easy to tailor to a particular workflow system, but it is extremely fragile. Given any change in the workflow system, this massive metadata consolidation software would have to be updated and any attempt to apply this software in a new environment would require significant portions of it to be updated if not outright replaced. Instead, for the purposes of supporting Findable, Accessible, Interoperable, and Reusable (FAIR) metadata [WDA+16], it is more appropriate to take the second approach.

By modularizing the metadata capture system, this approach is highly generalizable and can be easily modified or updated to support different systems. Furthermore, it can be maintained and adjusted by other interested developers to suit their needs. There will still have to be a software to collect the metadata generated by these scripts but this software can be far more generalized in the way it accepts inputs.

However, simply capturing and storing the metadata generated by these workflows is insufficient. Due to the heterogeneity of the pieces that make up an HPC pipeline, providing the raw metadata will be nearly as overwhelming to researchers as annotating it themselves. Therefore, this metadata must also be condensed into a format that is easy for researchers to analyze and process. Therefore, the metadata that is annotated by these specialized scripts and collected by the final software must also be consolidated in a uniform, approachable format.

5.2 Metadata Sources

Three components of the RECUP pipeline were chosen as initial metadata capture targets: workflow metadata, I/O metadata, and anomalous event metadata. These three sources provide a brief overview of the workflow by describing the workflow's spread across compute nodes, the I/O performed by these nodes, and the anomalous

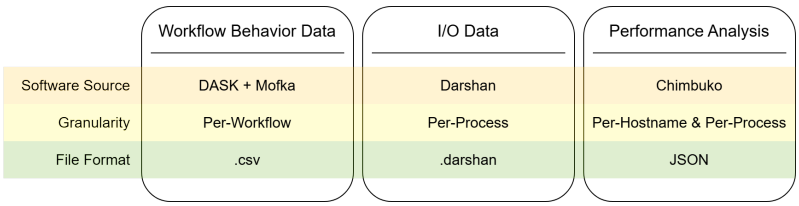

Figure 5.2: Table of the differences between the three chosen metadata sources: workflow metadata, I/O metadata, and anomalous event (performance analysis) metadata. Not only are the software that generate these metadata and their output formats unique to the metadata type, metadata generation occurs at different rates.

events during the workflow's life cycle. Importantly, these metadata sources are highly heterogeneous in source, granularity, and metadata format, shown in Figure 5.2. These sources thus act as a significant proof of concept of the feasibility of collecting metadata from the components of a RECUP pipeline rather than from the completed pipeline.

5.2.1 Workflow Metadata

Running workflows on HPC systems provides unique challenges in reproducibility. Researchers often have little to no control over the particular nodes used to run a step. Differences in nodes may cause changes in results or performance and thus are critical to annotate for workflow reproducibility.

In RECUP, two major workflow systems are used: Radical Cyber Tools (RCT) Pilot and Dask [Das16, BJMT19]. RCT-Pilot and Dask are both Python packages that serve as pilots, or systems that automatically manage task dependencies, task parallelization, and assignment of tasks to nodes. Researchers define their workflows as Python code and import RCT or Dask functions to take advantage of HPC resources.

Dask was chosen as an initial workflow metadata source due to its widespread use and existing technologies to capture communications between the Dask scheduler and workers, called the Dask-Mofka coupler [GPI+24]. The Dask-Mofka integration causes every entity in a running Dask workflow to send a message on any event to a Mofka instance spun up for this particular workflow. These messages are

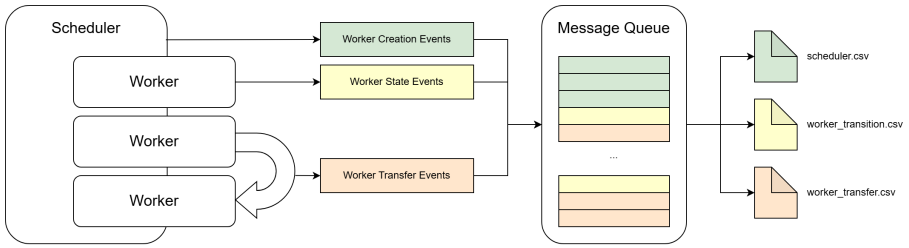

Figure 5.3: Schematic diagram of metadata collected during the run of a Dask workflow. There are three distinct types of messages from the Dask workflow, each of which partially describe different parts of a task's life cycle or a node's workload.

collected using a `consumer.py` script and organized into 3 files: one for the scheduler state transitions, one for worker scheduler state transitions, and one for worker file transfers (Fig 5.3).

The scheduler file contains metadata regarding the state of tasks from the perspective of the scheduler. A given entry in `scheduler.csv` describes metadata such as the start and end state of a task during a transition, when the transition occurs, and what worker the task was assigned to.

The `worker_transition.csv` file describes the transition of event states from the perspective of a worker. It contains the identifier for the task, its start state, end state, the ip address of the worker, and the timestamp of the event.

The `worker_transfer.csv` file describes all file transfer events between workers. These events describe the source and destination workers, the size of the file being transferred, the related tasks, and the time this transfer occurred.

It is important to note that each of these files contain metadata from the perspective of events. Events are a pilot-unique perspective. Researchers do not define their workflow by event but rather by operation or by task, where tasks are a set of at least one operation. An example of the differences between an event and a task are shown in Figure 5.4. Each event contains a subset of the metadata of a task's life cycle – for example, a worker transition event contains information on what type of node a task was processed on and at what time its computations were completed.

To gather this metadata into a format more accessible by researchers, we de-

```
# (a)
normalized_images = (images - da.min(images)) * (255.0 / (da.max(images)
                    - da.min(images)))

# (b)
5564,"('chunk_min-5a09aab47a69898004101a2c5b6e1b96', 137, 0, 0, 0)",
        released,fetch,tcp://10.201.0.212:34699,1713455691.4646766
```

Figure 5.4: Example of the difference between a user-defined Task or Operation and a Worker Event. Example `(a)` shows an example operation written by the user in their workflow that calculates the minimum and maximum value of an image's numerical values using the Dask version of the `max` and `min` operations. Example `(b)` shows one of the worker events associated with this operation, containing the IP address of the worker that completed this instance of the parallelized task.

veloped a tool that re-organizes the data generated by the Dask-Mofka coupler. This script collects the events across all three `.csv` files and creates `Scheduler-Event`, `Worker-Event`, and `Worker-Transfer-Event` objects to represent the metadata. In addition, the script has an object representation of a `Task` which it aims to populate using the information contained in the `-Event` objects.

As events are processed from the three files, the tool generates a name for its associated `Task` based on the information available. If there does not already exist a `Task`, it creates a new object and immediately associates the event with that object. If one already exists, the current event is added as an additional child to the existing `Task`.

As events are added to the `Task` object, attributes are added and updated to accurately represent the data contained within. For example, a `Task` contains the attributes `start_time` and `end_time`. These attributes are used for search and potentially indexing purposes: if a workflow took abnormally long to complete, researchers can look at a chart of the total run duration of tasks (perhaps even grouped by task operation type, such as `min`) to discover which instances took abnormally long to complete. From there, they can delve deeper into the particular attributes of the task and events within to diagnose the cause.

This object representation allows for easy addition to or reorganization of

the data contained within these three files. Currently, the script supports named indexing of `Tasks` and sorting by time. Additional sorting options and the addition of filters may be easily added, as well as the option to group data by compute node.

5.2.2 I/O Metadata

I/O metadata is collected from every process during the lifetime of a workflow by a tool running on the compute cluster called Darshan [CHA+11]. Darshan generates binary log files containing modules describing the different categories of I/O operations, such as POSIX or STDIO. These log files can only be read using the tools provided by the Darshan developers in `darshan-util` [LSR+23]. To support researchers working with Darshan log data, we develop scripts to pre-process these logs into a more universal and condensed format.

The first challenge in reading Darshan logs is the requirement of `darshan-util` availability. While Darshan is often installed on HPC systems, it is not necessarily installed on analysis machines. Furthermore, different workflows may use different versions of Darshan, such as experimental versions that include additional information in logs that cause the stable branch of `darshan-util` to crash. Therefore, we develop Docker images that serve as quickstart wrappers for viewing Darshan logs or running the preprocessing scripts. One image exclusively contains the tools necessary to run `darshan-util` and a second "dev" image contains additional python tools for hands-on analysis such as Jupyter notebooks.

To begin pre-processing the logs, we use the package `PyDarshan`, provided by the the Darshan team, a python module that interfaces with `darshan-util` to provide support for reading Darshan logs using Python. `PyDarshan` provides support for translating Darshan log modules into Pandas dataframes [WM10]. However, `PyDarshan` performs one-to-one translation: every module is exported into its own dataframe – and in several cases, a single module generates two dataframes that are structurally identical.

To aid researchers in analyzing this data from a workflow perspective, we restructure the dataframes generated by `PyDarshan` by collapsing related dataframes

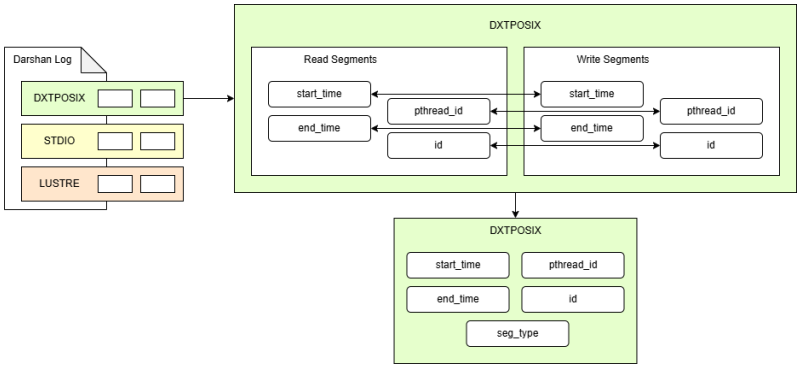

Figure 5.5: Visualization of the paired-dataframe consolidation of Darshan log data.

and combining dataframes across processes.

The first level of complexity reduction was to collapse paired-dataframe Darshan modules, visualized in Figure 5.5. Several modules, such as DXTPOSIX, STDIO, and LUSTRE contain two independent dataframes with structurally similar data. In the DXTPOSIX module, which represents the extended fine-grain traces of system calls, there are separate dataframes to represent read operations and write operations. These dataframes contain identical columns, such as *start_time*, *end_time*, *pthread_id* and *id*. To simplify, we collapsed these dataframes into a single dataframe, with an added annotation named *seg_type* that represents whether it is a *read* or *write* segment.

This simplification was repeated for all modules that contained two dataframes with identically structured data.

The second level of complexity to reduce was the separation according to process. For every process, Darshan generates a log file containing modules with each of its categorized I/O events. To better support a workflow-oriented perspective, we reorganized this metadata to be organized by workflow and module rather than process and module.

This is done by attaching unique process identifying information (job ID, job UID) to all entries within a Darshan log. Then, the modules are extracted and condensed as mentioned in the previous section. Finally, each instance of a given module across all Darshan logs are combined into a single dataframe (Fig 5.6).

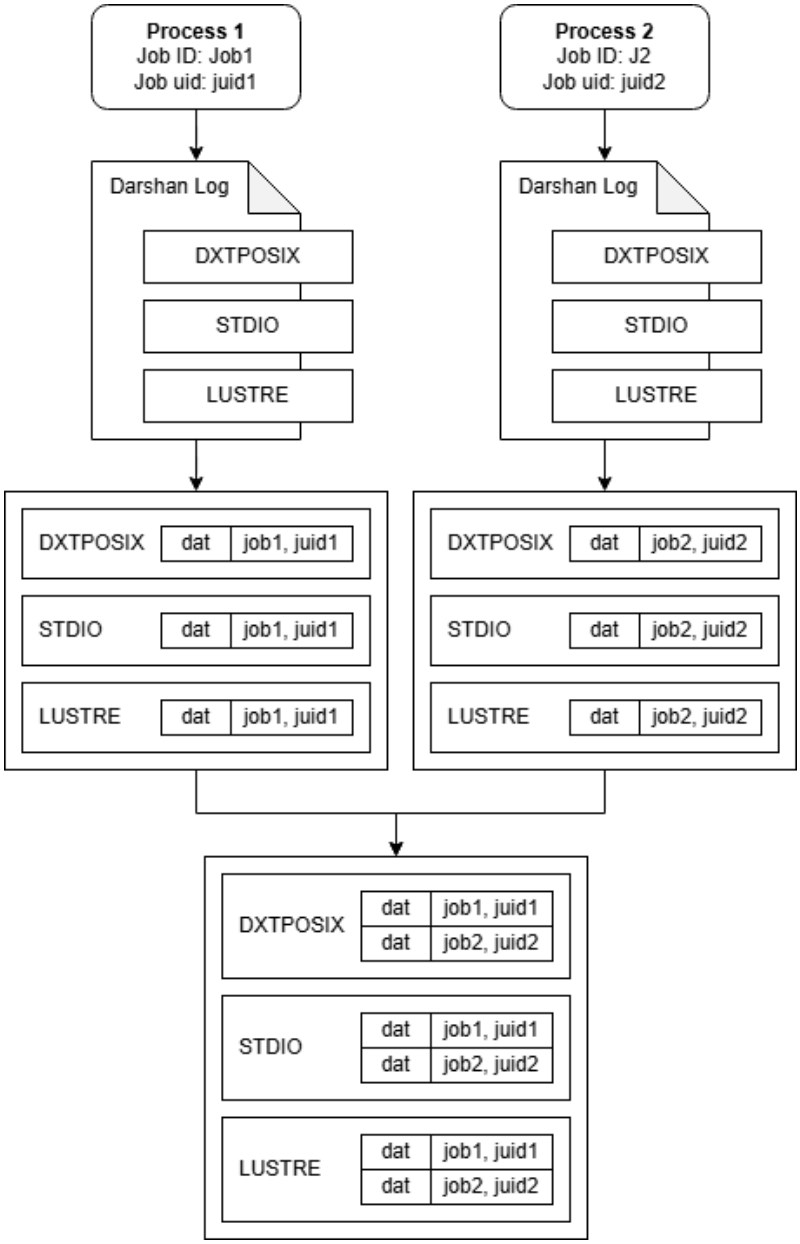

Figure 5.6: Reorganization of the data contained in different Darshan logs from the same workflow. This restructuring massively reduces the number of files – in this case, as all pictured modules are dual-dataframe modules, we reduce from 12 dataframes to 3.

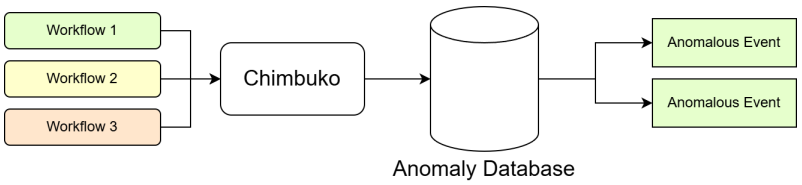

Figure 5.7: Schematic diagram of the Chimbuko anomaly detection process. Note that Chimbuko captures anomalous events from multiple workflows and these events must be filtered to only include the workflow of interest.

On completion, this tool outputs several consolidated dataframes containing all of the IO metadata from the workflow run.

5.2.3 Anomalous Event Metadata

The final category of collected metadata is performance analysis metadata. The source chosen is Chimbuko, an anomaly detection and storage database framework [KHH+20]. Chimbuko monitors running processes on a system, identifying and annotating anomalous execution events according to a rolling baseline of workflow performance. On anomaly detection, it stores the anomalous events in a JSON docstore database. As shown in Figure 5.7, a single instance of Chimbuko runs on a cluster, observing multiple running workflows.

As Chimbuko stores anomalous events to a database, the initial version of this tool operates off a database file itself. This ensures that even if Chimbuko is not currently running and serving endpoints for its database, researchers can still obtain data from logged events. This import is straightforward and thus this tool was used to test potential output formats for metadata consolidation.

5.3 Output Consolidation and Format

The three described tools have three separate output formats. The workflow metadata tool outputs custom Python objects, the I/O metadata tool outputs Pandas Dataframes, and the anomalous event tool output JSON. To fully support ease of analysis, these three output formats must be unified in a format that is easiest for researchers to use.

```
"@graph": [
{
    "@id": "ro-crate-metadata.json",
    "@type": "CreativeWork",
    "about": {"@id": "./"},
    "conformsTo": [
        {"@id": "https://w3id.org/ro/crate/1.1"},
        {"@id": "https://w3id.org/workflowhub/workflow-ro-crate/1.0"}
    ]
},
```

Figure 5.8: Example of root attributes of an RO-Crate. This snippet declares that it is describing an object with the name `ro-crate-metadata.json`, and that this file conforms to the standards of version 1.1 of RO-Crate and version 1.0 of the workflow RO-Crate profile.

Initially, the intent of this project was to output Research Object Crates, or RO-Crates [SRSC+22]. As discussed in Chapter 1 in the "Metadata-Specific Formats" section, RO-Crate is a packaging format that became a community standard. By outputting metadata in an existing format, our metadata capture tools can easily interoperate with existing software and development environments.

In addition to acting as a container for data files (known as a "crate"), RO-Crates support the use of a RO-Crate Profile. RO-Crate Profiles act as specifications for a `ro-crate-metadata.json` file, which is a file that contains metadata describing the research wrapped within the crate. To support FAIR research, many scientific communities have defined RO-Crate profiles that define strict expectations for the metadata in crates related to their field, and RO-Crates that are shared within that community are expected to conform to these profiles. An example of the RO-Crate Profile JSON-LD structure structure is shown in Figure 5.8.

However, RO-Crates are designed with an assumption of completeness. An RO-Crate is meant to be a complete, whole description of a crate's data. This is incompatible with the approach of these tools – if each component of a workflow generates its own metadata, the metadata it generates is by definition only a subset of the metadata describing the entire workflow.

Furthermore, existing RO-Crate Profiles describe workflows from a different

perspective. While the workflows in RECUP are operation or task-focused to best use HPC resources, many RO-Crate Profiles are designed around the movement of files. For example, the RO-Crate Profile for Workflow Run Provenance places a heavy emphasis on input and output files and their connection to software steps, but in the metadata captured from RECUP workflows this must be derived from a combination of the workflow and I/O metadata.

Therefore, RO-Crate cannot be used as an output from the component-specific metadata collection tools. An intermediary format should be used to output metadata from these component-specific tools such that all the metadata can be collected by a single generalizable script and compiled into an output format, which may include RO-Crate.

5.4 Conclusion

We developed a set of tools to collect metadata from three separate heterogenous sources in the RECUP pipeline and reorganize the output for ease of use. We collect workflow metadata from Dask, I/O metadata from Darshan, and anomalous event metadata from Chimbuko to gain a varied perspective of possible pipeline metadata. These software reorganize the collected metadata to various levels, experimenting with formats and levels of granularity to begin determining what output would most assist researchers in studying the performance of their workflows. All software is open source and available on GitHub in the RECUP organization: `https://github.com/RECUP-DOE`.

Chapter 6

Conclusion and Future Work

In this thesis, I have described two approaches to supporting FAIR metadata.

6.1 MEDFORD

The first approach, MEDFORD, focuses on supporting FAIR metadata from the author's side. Working under the assumption that the metadata was small in scale and set by the researcher, it aims to provide a straightforward syntax that was easy to write. MEDFORD can be written using any text editor because it uses purely plaintext ASCII characters, with minimal formatting requirements that were designed to be as unintrusive as possible.

We also developed a set of tools – such as the MEDFORD parser – to help researchers ensure that their written MEDFORD follows all syntax requirements. These syntax requirements ensure that MEDFORD can be easily parsed by automatic systems and this support for automatic parsing enables the development of tools and vocabulary to describe content requirements. In turn, the support for content requirements gives a concrete way of ensuring certain content is present in the MEDFORD file, thus reducing the amount of manual validation that had to be done by data curators between a researcher submitting their metadata and their target database accepting it.

To improve the usefulness of MEDFORD, I explored two extensions to the

language: the ability to define the content expectations of a MEDFORD file and the ability to import external metadata, explored in chapters 3 and 4 respectively.

6.1.1 Error Handling

MEDFORD was designed to be approachable for researchers without a significant amount of programming experience. MEDFORD itself can be written without any external tool, allowing users to write without a constant barrage of errors or warnings. However, eventually the written metadata must eventually be passed to the MEDFORD parser for processing and the parser must communicate issues to the author.

There are three major types of errors:

1. Syntax errors
2. Type errors
3. Requirement errors

Syntax errors are errors in the MEDFORD formatting itself. For example, if a user omits the Name line when writing a MEDFORD block, this would be considered a syntax error (Fig 6.1), as it prevents the MEDFORD parser from processing the file itself. These errors are caught before the file finishes processing, and may even force the parser to terminate early.

Type errors occur when the content of the MEDFORD line do not match the expected formatting. For example, we plan to implement ORCID validation – if an ORCID line is provided, it will be validated against expected ORCID formatting, and the parser may pull additional information using that ORCID. Were a researcher to provide a link to the contributor's ORCID page instead of their ORCID itself (Fig 6.2), this would fail type checking as it is not in a format that the parser could easily validate. In the current version of the MEDFORD parser, written in Python and using the Pydantic package for type definitions, these errors are thrown by Pydantic

```
#Incorrect
@Contributor-Email polina.shpilker@tufts.edu
@Contributor-Affiliation Tufts University

#Correct
@Contributor Polina Shpilker
@Contributor-Email polina.shpilker@tufts.edu
@Contributor-Affiliation Tufts University
```

Figure 6.1: Example of a Contributor block missing a Name line. This would be a Syntax error, as all MEDFORD blocks require a Name line to identify the metadata object being described.

```
#Incorrect
@Contributor Polina Shpilker
@Contributor-ORCID https://orcid.org/0000-0002-6761-7326

#Correct
@Contributor Polina Shpilker
@Contributor-ORCID 0000-0002-6761-7326
```

Figure 6.2: Example of an inappropriately formatted ORCID line. In the first line, the author provided a URL to the contributor's ORCID page instead of the ORCID itself. The parser expects the ORCID itself to process.

as it attempts to align the provided content with the provided types and content validation functions.

Requirement errors are errors that happen due to the requirements imposed upon the metadata by databases. Unlike the previous two types of errors, these errors do not necessarily occur directly from the content of the MEDFORD file. An error could occur due to the content, as shown in Figure 6.3, but it is also possible for these errors to occur from the absence of content.

In the current implementation of the Python MEDFORD parser, these errors are also defined in and thrown by Pydantic. However, once the work described in Chapter 3 of this thesis towards the development of a domain specific language for defining content expectations in completed, these errors will be found and thrown using an entirely different system.

Currently, these errors are not handled. They are thrown as exceptions,

```
#Incorrect
@Contributor Polina Shpilker
@Contributor-Role Corresponding Author

#Correct
@Contributor Polina Shpilker
@Contributor-Role Corresponding Author
@Contributor-Email polina.shpilker@tufts.edu
```

Figure 6.3: Example of a requirement error. For contributors with the role 'Corresponding Author', an Email must be provided. This is an example of an error caused by content missing from the MEDFORD file, rather than content within the file itself.

causing Python to crash and print the error message to the terminal. Some work has been done to define MEDFORD-specific errors, such as the `MissingDescError`, which represents a missing Name line, or `MissingContentError` which represents a line where no payload was provided (only a major token and a minor token.)

A future development to improve MEDFORD's usability and adoptability would be to add a better handling system for errors to the current parser. Rather than causing python to exit with an exception, there should be options for collecting and providing the errors to users at different granularities. For example, in the case of an IDE, errors should be provided as they occur so that the IDE can present them at the lines that caused them. In another case, a user using the parser from the command line can choose to have the parser process the entire file before providing all errors as a single output. This output should also be sortable by line number or by error type.

By capturing the various types of errors and making their output a part of the MEDFORD parser's functionality, we are also able to re-word the error messages to make them more intuitive for users. Rather than expecting them to be able to debug Python error messages (or Pydantic error messages, in the case of Type errors), we can present the issue in a more straightforward manner to make it easier for them to ensure their MEDFORD fulfills all requirements to parse.

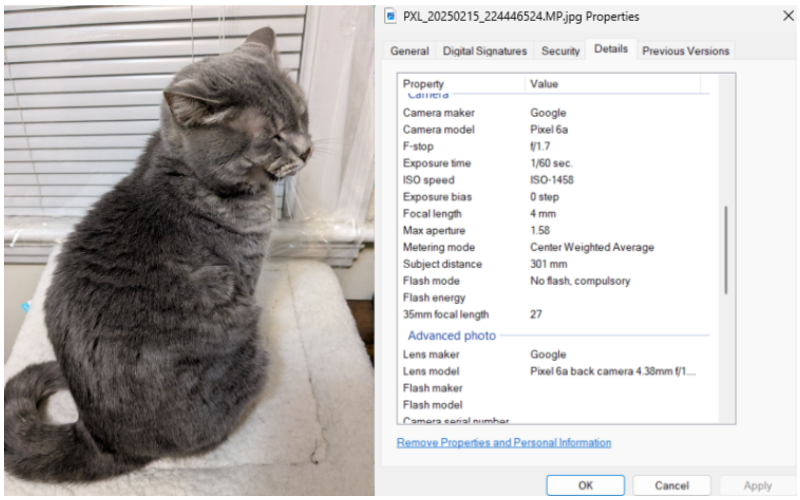

Figure 6.4: Example photograph alongside its EXIF metadata. Note the extensive and photograph-specific metadata described for this photo.

6.1.2 Metadata Cross-Referencing and Import

In chapter 4 of this thesis, I described the addition of logical links between metadata. By changing the implicit `desc` line into a `name` that must uniquely name the block amongst other blocks of the same major token, we take the implied metadata objects and turn them into proper named entities. This enables blocks to then associate one of their attributes with another block, allowing metadata to point to other metadata without trying to claim ownership or imposing a hierarchy.

This development was actually a milestone on the path to a larger goal: the ability to import and reference external metadata from other sources. Specifically, our first goal was the capability to reference metadata from images, called Exchangeable Image File Format (EXIF) metadata, as our initial coral research use case involved a large number of coral photographs.

EXIF metadata includes a wide variety of metadata, ranging from photograph-specific metadata such as ISO speed to generic but detailed metadata such as GPS coordinates (if enabled on the device that took the photograph), shown in Figure 6.4. Especially in the case of GPS coordinates, this metadata can be critical for further research, but its complexity makes it difficult for researchers to manually write into a MEDFORD file. Requiring MEDFORD file authors to manually enter

these values into their files adds immense possibility of error as they must enter two eight-digit numbers per photograph. Furthermore, it takes up their valuable time for what is an extremely menial task of copying numbers from one location to another, significantly decreasing the likelihood that the researcher decides to finish writing their MEDFORD file.

Therefore, it is important to add the capability to import metadata from external files in MEDFORD. By naming blocks within MEDFORD to allow cross-block referencing, we can now support the concept of importing metadata, but there are still two major components missing: importing metadata from other files and translating other metadata formats into MEDFORD.

6.1.2.1 External MEDFORD Files

The first proposed improvement related to external metadata is the capability to import metadata from other MEDFORD files. There are two major use cases for this capability:

First, consider the existence of a "Reference" MEDFORD file. A research lab has a single MEDFORD file that contains the `@Contributor` information for every participant in the lab. When an individual researcher prepares a MEDFORD file for their project, they should be able to import the contributor data for all of their contributors from this other MEDFORD file. This can be considered a Many-to-One relationship; this Reference MEDFORD file is never meant to be submitted as is and instead is used by an arbitrary number of MEDFORD files.

The second case of importing a MEDFORD file is in the case where a MEDFORD file either references existing research or extends upon it. In this case, the MEDFORD file has been submitted to a database and is in itself a complete declaration of the metadata of a research project, and the novel MEDFORD file should copy blocks of its metadata for the portions of its data it used in its own research.

These two cases are extremely similar, but the difference in the purpose of the original MEDFORD file adds some complexity that needs to be considered. In the first example, the copied metadata should be considered novel in the final

MEDFORD file. In the second example, the copied metadata should be treated as a copy unless the research has been directly reproduced by the researchers themselves.

This leads to a larger question of differentiating between "Copy" and "Primary" metadata, which was briefly discussed in the original MEDFORD publications but were ultimately not yet expanded upon in favor of making an initial working version of MEDFORD. Along with the development of a vocabulary to use for referencing metadata in another MEDFORD file, this topic should be explored, as well as some way of differentiating between a MEDFORD file that has been validated, a MEDFORD that will be validated, and a MEDFORD file that is never meant to be validated for compilation into another format and only used as a reference.

6.1.2.2 External Metadata Files

The second improvement is the capability to import metadata from external files of other types. As previously mentioned, one of our most important examples is importing metadata from images. This adds two points of complexity: first, the external metadata is in a different format that must be processable by MEDFORD. Secondly, we assumed in the previous discussion that the MEDFORD files were correct. What if the camera used to take the photographs did not have its time zone updated to represent the correct time for the location the photo was taken at? Unlike MEDFORD files, which are still plain text and can be corrected, external metadata may be difficult to access or edit.

The first problem is partially an update to the MEDFORD language. In the case of external metadata that is attached to the data itself (such as with EXIF), it would be unwieldy to separately import the metadata when the author already had to define the file path. Instead, there must be some syntax to declare that there is metadata attached to the data file, as well as the format of the metadata. Then, it becomes a parser implementation problem – every external metadata type will need a converter that can translate it into MEDFORD. Then, back in MEDFORD itself, there must be some syntax to declare what metadata fields it wants from that

external file, and what minor tokens to associate those fields with.

Finally, there must be some syntax to then declare that the imported metadata is incorrect. Note that this is important in this case and not the external MEDFORD import because the metadata import from external metadata is a bulk import, while the described external MEDFORD import is attribute-by-attribute. Now that external metadata import has been solved in a bulk fashion, it opens the door to importing MEDFORD metadata in bulk as well.

6.1.2.3 Bulk MEDFORD Import

A final note on this topic is the concept of bulk importing MEDFORD metadata. The previously mentioned method, which is on a minor token by minor token basis, is not the optimal solution. Now that we have proposed the concept of correcting metadata, consider instead the possibility of importing entire MEDFORD blocks at a time. Now, when declaring a MEDFORD file to import, the author can also pass a series of blocks to import with the ability to adjust them.

Consider the example of a contributor: in the reference file, they must have been described without a `Role`, as there is no meaning for `Role` in a MEDFORD file that does not describe research. Even beyond that, there is no content for `Role` that could be universally applicable. An author must import the applicable `Contributor` block and correct the block to have the `Role` minor token according to their role in that particular research project.

This also leads to another new complexity: replicated metadata. While an issue even in a single MEDFORD file, it becomes far more likely as researchers begin to import other reference MEDFORD files. It is possible that two MEDFORD files have blocks with identical names that represent entirely separate objects. This may be resolved by assigning all imported MEDFORD blocks a prefix according to the name of the `@Metadata` block it was imported with, but this may add complexity to names that only escalates as MEDFORD files containing imported data are, themselves, imported. This may also be resolved using the correction syntax, but it may require improving the correction syntax to support correcting names.

```
@Species Pocillopora damicornis
@Species-Loc Sabago Isthmus, Panama
@Species-ReefCollection [..]
@Species-Cultured University of Miami Coral Resource Facility
@Species-CultureCollection [..]
```

Figure 6.5: Example of a MEDFORD template. Most of the metadata for this block is filled out, except for two minor tokens, which are filled with the text `[..]`, which represents missing data that must be filled in on template use.

6.1.3 Templates

A feature that was defined during MEDFORD's original specification was the concept of Templates. Templates were intended to be starting points for rapidly filling out consistent MEDFORD blocks. An example from Chapter 2 is repeated in 6.5. The initial version of templates were plain-text blocks of MEDFORD that users copy and paste to re-use. The tooling around MEDFORD would also support users using template blocks – the IDE highlights any `[..]` template markers left in the open file, and a custom error should be added to the MEDFORD parser to warn the user that they left template markers in the provided file.

While straightforward to use, these templates are difficult to discover and bulky to use. Early in MEDFORD's development, a repository of community-developed templates was proposed. While still very useful, this suffers from the very same problem mentioned in the first chapter of this thesis – as more templates are collected, they must be organized or have some metadata connected to them to make them searchable and discoverable.

The method of copy and pasting templates may work for minor use cases, such as filling in one or two coral species, but can quickly become an impediment when working with larger sets of data. Consider the case where over a hundred photographs were taken with the same camera of the same subject over the course of a day in a research project. Optimally, the user should not have to re-enter the metadata that is consistent across all one hundred photos. An example of the pseudo-MEDFORD is shown in Figure 6.6.

```
@Photo PDamPhoto1
@Photo-File photos/pdam/photo1.raw
@Photo-Species @Species Pocillopora damicornis
@Photo_External EXIF
@Photo_External-Date 2025-04-01T17:20-05:00
```

Figure 6.6: Pseudo-MEDFORD block mocking a description of a `Photo` that has external EXIF metadata. The external EXIF metadata field Date has been corrected.

```
@Photo [..]
@Photo-File photos/pdam/[..].raw
@Photo-Species @Species Pocillopora damicornis
@Photo_External EXIF
@Photo_External-Date 2025-04-01T[..]:[..]-05:00
```

Figure 6.7: The pseudo-MEDFORD block shown in Figure 6.6 adjusted to be a template. Regions that may differ between copies of the block have been replaced with the template marker, `[..]`.

A reasonable response would be to turn this block into a template, shown in Figure 6.7. This is theoretically sufficient: the user can copy the template, paste a copy, edit the associated fields, and repeat. However, this process is tedious and repetitive, making it easy for authors to make mistakes.

This difficulty may first be alleviated by updating tooling to have rich template support. If language server protocols (LSPs) developed for MEDFORD are aware of available templates downloaded by the user, it may support inserting an entire template block in a few keypresses. Furthermore, it may make it easier for users to edit templates by supporting tab navigation through template markers remaining in the template.

A longer-term solution would involve adjusting the metadata import syntax to have a similar tactic for importing templates. Templates are, in fact, simply metadata blocks that are guaranteed to be incorrect! Therefore, it stands to reason that templates could be treated similarly as imported metadata that requires corrections. However, in the external metadata import discussion, it was assumed that entire attributes would be correct at a time. In the example presented in Figure

6.7, most of the attribute is correct except for a particular sub-region denoted by the template marker. Were we to treat these like external metadata and force the author to replace the entire attribute, the strength of the template would be lost. Therefore, while templates can be treated similarly to imported external metadata, they will require unique syntax to allow authors to replace only the regions denoted by template markers.

6.1.4 Tutorials

Despite how much effort has been done to design MEDFORD to be as user-friendly as possible, it is still a novel format for researchers to write their metadata in and requires some knowledge of how the tooling that surrounds it works. Therefore, the creation of a series of tutorials on how to use MEDFORD would greatly support adoption.

The main tutorial that needs to be made is a walkthrough that walks the reader through writing a MEDFORD file and putting it through the MEDFORD parser. It should also point users to useful resources, such as the Visual Studio Code MEDFORD extension and a link to download the MEDFORD parser.

This tutorial must highlight the fact that there exists an expected vocabulary, while also stressing that users are free to create their own tokens as necessary. In lieu of having a particular compilation target in mind and supported, it should also walk through users creating a BagIt Bag [KLM+18] from their MEDFORD file.

6.1.5 Defining Content Expectations with a Domain-Specific Language

Chapter 3 of this thesis described a foundation for a Domain-Specific Language (DSL) for defining MEDFORD content expectations. The purpose of this language is to provide a method for data curators to define what they expect from any MEDFORD files submitted to their databases, reducing the amount of correction communications between submitters and curators and taking advantage of the error system discussed in the Error Handling system discussed above.

This language must be implemented and tested. Implementation into the current python parser should be fairly straightforward using tree-based parsing tools, replacing the object-oriented module currently used. After an initial version is developed, the content validation specification syntax must be tested to determine what level of complexity can be achieved without becoming too difficult to adopt by data curators. This syntax should then be used to describe example database requirements to confirm its capability to describe these requirements and then provided to collaborators for user testing.

Furthermore, while the DSL is initially made as an intermediary between MEDFORD and existing database systems, the DSL itself may act as a foundation for a database. Since the DSL describes the shape of the expected metadata, it has all the information it needs to make the database tables and store the relationships between files in a research project.

6.2 RECUP

Three metadata collection tools were described in the RECUP work described in Chapter 5. The purpose of these tools is to collect the metadata generated by the workflow stages, I/O events, and anomalous events of a workflow running on a high-performance compute (HPC) cluster, as this metadata is critical for understanding workflow performance and provenance. As mentioned previously, this metadata is far too massive and complex for researchers to annotate by hand and is often automatically generated by the workflow. The purpose of these tools is the reformat and collect these metadata into a format that can be as accessible as possible for researchers.

Currently, the tools output their own individual collections of metadata in partial RO-Crates [SRSC+22], Pandas dataframes [WM10], and Python objects. This metadata should be collected and presented to researchers as a singular unit, in a format that is most accessible for their analytical pipelines. In order to do so, there must exist a consolidation script.

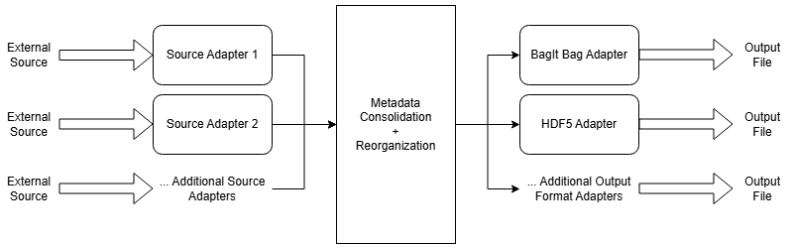

Figure 6.8: Diagram of the intended structure of the consolidation script. External sources communicate with components of the consolidation script called adapters that are developed to accept the input from that source. The consolidation script then collects the metadata and uses one of several output adapters to output to a format of choice.

We propose that this consolidation script is modular. Much like how we developed individual scripts to capture the metadata from individual components of a workflow to support changes in the workflow, this script should be easily adjustable to accept metadata from a novel capture script. It should exist as three distinct layers, shown in Figure 6.8: a set of input adapters, a middle consolidation layer, and a set of output adapters. Input adapters are added, removed, or adjusted as the workflows change, and the output adapters are added as novel output formats are supported.

The input adapters – or the original collection scripts themselves – should adjust the metadata to be represented as dataframes. All metadata currently being captured is structured sufficiently to be represented by dataframes easily and dataframes are widely used in data science and thus should easily fit into existing analytical workflows. These dataframes are then collected by the adapters and by the consolidation layer.

From here, there are two output formats currently in consideration: a BagIt Bag [KLM+18] and HDF5 [HDF]. Both should eventually be supported, but we propose initially developing the BagIt Bag output. This process would involve simply compressing the dataframes collected by the consolidation layer, but this alone makes it considerably easier for researchers to share their workflow's metadata. HDF5 is a far more complex target, but HDF5 is often used in machine learning analysis pipelines and would be optimal to support researcher analysis.

Bibliography

[BCHK+22] Michelle Barker, Neil P. Chue Hong, Daniel S. Katz, Anna-Lena Lamprecht, Carlos Martinez-Ortiz, Fotis Psomopoulos, Jennifer Harrow, Leyla Jael Castro, Morane Gruenpeter, Paula Andrea Martinez, and Tom Honeyman. Introducing the FAIR Principles for research software. *Scientific Data*, 9(1):622, October 2022. 10.1038/s41597-022-01710-x.

[BGJK16] Alex Ball, Jane Greenberg, Keith Jeffery, and Rebecca Koskela. Rda metadata standards directory working group: Final report. RDA recommendation, Research Data Alliance, 2016.

[BJMT19] Vivek Balasubramanian, Shantenu Jha, André Merzky, and Matteo Turilli. Radical-cybertools: Middleware building blocks for scalable science. *CoRR*, abs/1904.03085, 2019. 10.48550/arXiv.1904.03085.

[BK19] Hendrik Bünder and Herbert Kuchen. Towards multi-editor support for domain-specific languages utilizing the language server protocol. In *International Conference on Model-Driven Engineering and Software Development*, pages 225–245. Springer, 2019. 10.1007/978-3-030-37873-8_10.

[BMN11] Thomas CG Bosch and Margaret J McFall-Ngai. Metaorganisms as the new frontier. *Zoology*, 114(4):185–190, 2011. 10.1016/j.zool.2011.04.001.

[BO] Biological and Chemical Oceanography Data Management Office.

[BWF+00] Helen M. Berman, John Westbrook, Zukang Feng, Gary Gilliland, T. N. Bhat, Helge Weissig, Ilya N. Shindyalov, and Philip E. Bourne. The Protein Data Bank. *Nucleic Acids Research*, 28(1):235–242, 01 2000. 10.1093/nar/28.1.235.

[CGK+16] C. L. Chandler, R. C. Groman, D. Kinkade, et al. BCO-DMO: Stewardship of marine research data from proposal to preservation. *American Geophysical Union*, 2016:OD24B–2457, 2016.

[CHA+11] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. Understanding and improving computational science storage access through continuous characterization. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14, 2011.

[CMP17] Nicolas CARPi, Alexander Minges, and Matthieu Piel. elabftw: An open source laboratory notebook for research labs. *Journal of Open Source Software*, 2(12):146, 2017. 10.21105/joss.00146.

[Das16] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016.

[DRH17] Simon D Donner, Gregory JM Rickbeil, and Scott F Heron. A new, high-resolution global mass coral bleaching database. *PLoS One*, 12(4):e0175490, 2017. 10.1371/journal.pone.0175490.

[FAJS05] Eric H Fegraus, Sandy Andelman, Matthew B Jones, and Mark Schildhauer. Maximizing the value of ecological data with structured metadata: an introduction to ecological metadata language (EML) and principles for metadata creation. *Bulletin of the Ecological Society of America*, 86(3):158–168, 2005. 10.1890/0012-9623(2005)86[158:MTVOED]2.0.CO;2.

[GPI+24] Amal Gueroudji, Chase Phelps, Tanzima Z. Islam, Philip Carns, Shane Snyder, Matthieu Dorier, Robert B. Ross, and Line C. Pouchard. Performance characterization and provenance of distributed task-based workflows on hpc platforms. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 2032–2039, 2024. 10.1109/SCW63240.2024.00254.

[Gro09] Michael Grobe. Rdf, Jena, SparQL and the "semantic web". In *Proceedings of the 37th annual ACM SIGUCCS fall conference: communication and collaboration*, pages 131–138, 2009. 10.1145/1629501.1629525.

[HBB+17] Terry P Hughes, Michele L Barnes, David R Bellwood, Joshua E Cinner, Graeme S Cumming, Jeremy BC Jackson, Joanie Kleypas, Ingrid A Van De Leemput, Janice M Lough, Tiffany H Morrison, et al. Coral reefs in the Anthropocene. *Nature*, 546(7656):82, 2017. 10.1038/nature22901.

[HDF] HDF Group. Hierarchical Data Format, version 5.

[JdMAJ+20] Annika Jacobsen, Ricardo de Miranda Azevedo, Nick Juty, Dominique Batista, Simon Coles, Ronald Cornet, Mélanie Courtot, Mercè Crosas, Michel Dumontier, Chris T. Evelo, Carole Goble, Giancarlo Guizzardi, Karsten Kryger Hansen, Ali Hasnain, Kristina Hettne, Jaap Heringa, Rob W.W. Hooft, Melanie Imming, Keith G. Jeffery, Rajaram Kaliyaperumal, Martijn G. Kersloot, Christine R. Kirkpatrick, Tobias Kuhn, Ignasi Labastida, Barbara Magagna, Peter McQuilton, Natalie Meyers, Annalisa Montesanti, Mirjam van Reisen, Philippe Rocca-Serra, Robert Pergl, Susanna-Assunta Sansone, Luiz Olavo Bonino da Silva Santos, Juliane Schneider, George Strawn, Mark Thompson, Andra Waagmeester, Tobias Weigel, Mark D. Wilkinson,

Egon L. Willighagen, Peter Wittenburg, Marco Roos, Barend Mons, and Erik Schultes. FAIR principles: Interpretations and implementation considerations. *Data Intelligence*, 2(1-2):10–29, January 2020. 10.1162/dint_r_00024.

[KHH+20] Christopher Kelly, Sungsoo Ha, Kevin Huck, Hubertus Van Dam, Line Pouchard, Gyorgy Matyasfalvi, Li Tang, Nicholas D'Imperio, Wei Xu, Shinjae Yoo, and Kerstin Kleese Van Dam. Chimbuko: A workflow-level scalable performance trace analysis tool. In *ISAV'20 In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ISAV'20, page 15–19, New York, NY, USA, 2020. Association for Computing Machinery. 10.1145/3426462.3426465.

[KLM+18] J. Kunze, J. Littman, E. Madden, J. Scancella, and C. Adams. The BagIt File Packaging Format (V1.0). RFC 8493, 2018. 10.17487/RFC8493.

[LAV16] Yi Jin Liew, Manuel Aranda, and Christian R Voolstra. Reefgenomics. org-a repository for marine genomics data. *Database*, 2016, 2016. 10.1093/database/baw152.

[LL22] Alexander L R Lubbock and Carlos F Lopez. Microbench: automated metadata management for systems biology benchmarking and reproducibility in Python. *Bioinformatics*, 38(20):4823–4825, 08 2022. 10.1093/bioinformatics/btac580.

[LNH+20] Jeremy Leipzig, Daniel Nüst, Charles Tapley Hoyt, Stian Soiland-Reyes, Karthik Ram, and Jane Greenberg. The role of metadata in reproducible computational research. *CoRR*, abs/2006.08589, 2020. 10.48550/ARXIV.2006.08589.

[LSR+23] Jakob Luettgau, Shane Snyder, Tyler Reddy, Nikolaus Awtrey, Kevin Harms, Jean Luca Bez, Rui Wang, Rob Latham, and Philip Carns. En-

abling agile analysis of i/o performance data with pydarshan. In *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 1380–1391, New York, NY, USA, 2023. Association for Computing Machinery. 10.1145/3624062.3624207.

[MHC+16] Joshua S Madin, Mia O Hoogenboom, Sean R Connolly, Emily S Darling, Daniel S Falster, Danwei Huang, Sally A Keith, Toni Mizerek, John M Pandolfi, Hollie M Putnam, et al. A trait-based approach to advance coral reef science. *Trends in Ecology & Evolution*, 31(6):419–428, 2016. 10.1016/j.tree.2016.02.012.

[MNV+17] Barend Mons, Cameron Neylon, Jan Velterop, Michel Dumontier, Luiz Olavo Bonino da Silva Santos, and Mark D. Wilkinson. Cloudy, increasingly FAIR; revisiting the FAIR data guiding principles for the european open science cloud. *Information Services & Use*, 37(1):49–56, March 2017. 10.3233/isu-170824.

[NIR+23] Bogdan Nicolae, Tanzima Z. Islam, Robert Ross, Huub Van Dam, Kevin Assogba, Polina Shpilker, Mikhail Titov, Matteo Turilli, Tianle Wang, Ozgur O. Kilic, Shantenu Jha, and Line C. Pouchard. Building the i (interoperability) of fair for performance reproducibility of large-scale composable workflows in recup. In *2023 IEEE 19th International Conference on e-Science (e-Science)*, pages 1–7, 2023. 10.1109/e-Science58273.2023.10254808.

[QBG12] Jian Qin, Alex Ball, and Jane Greenberg. Functional and architectural requirements for metadata: Supporting discovery and management of scientific data. *International Conference on Dublin Core and Metadata Applications*, page 62–71, 9 2012. 10.23106/dcmi.952136606.

[Res25] Research Space. RSpace, 2025. 10.5281/zenodo.14628346.

[SFM+22] Polina Shpilker, John Freeman, Hailey McKelvie, Jill Ashey, Jay-Miguel Fonticella, Hollie Putnam, Jane Greenberg, Lenore Cowen, Alva Couch, and Noah M. Daniels. Metadata format for open reef data (medford). In Emmanouel Garoufallou, María-Antonia Ovalle-Perandones, and Andreas Vlachidis, editors, *Metadata and Semantic Research*, pages 206–211, Cham, 2022. Springer International Publishing. 10.1007/978-3-030-98876-0_18.

[SPA+22] Michael C. Schatz, Anthony A. Philippakis, Enis Afgan, Eric Banks, Vincent J. Carey, Robert J. Carroll, Alessandro Culotti, Kyle Ellrott, Jeremy Goecks, Robert L. Grossman, Ira M. Hall, Kasper D. Hansen, Jonathan Lawson, Jeffrey T. Leek, Anne O'Donnell Luria, Stephen Mosher, Martin Morgan, Anton Nekrutenko, Brian D. O'Connor, Kevin Osborn, Benedict Paten, Candace Patterson, Frederick J. Tan, Casey Overby Taylor, Jennifer Vessio, Levi Waldron, Ting Wang, Kristin Wuichet, Alexander Baumann, Andrew Rula, Anton Kovalsy, Clare Bernard, Derek Caetano-Anollés, Geraldine A. Van der Auwera, Justin Canas, Kaan Yuksel, Kate Herman, M. Morgan Taylor, Marianie Simeon, Michael Baumann, Qi Wang, Robert Title, Ruchi Munshi, Sushma Chaluvadi, Valerie Reeves, William Disman, Salin Thomas, Allie Hajian, Elizabeth Kiernan, Namrata Gupta, Trish Vosburg, Ludwig Geistlinger, Marcel Ramos, Sehyun Oh, Dave Rogers, Frances McDade, Mim Hastie, Nitesh Turaga, Alexander Ostrovsky, Alexandru Mahmoud, Dannon Baker, Dave Clements, Katherine E.L. Cox, Keith Suderman, Nataliya Kucher, Sergey Golitsynskiy, Samantha Zarate, Sarah J. Wheelan, Kai Kammers, Ana Stevens, Carolyn Hutter, Christopher Wellington, Elena M. Ghanaim, Ken L. Wiley, Shurjo K. Sen, Valentina Di Francesco, Deni s Yuen, Brian Walsh, Luke Sargent, Vahid Jalili, John Chilton, Lori Shepherd, B.J. Stubbs, Ash O'Farrell, Benton A. Vizzier, Charles Overbeck, Charles

Reid, David Charles Steinberg, Elizabeth A. Sheets, Julian Lucas, Lon Blauvelt, Louise Cabansay, Noah Warren, Brian Hannafious, Tim Harris, Radhika Reddy, Eric Torstenson, M. Katie Banasiewicz, Haley J. Abel, and Jason Walker. Inverting the model of genomics data sharing with the nhgri genomic data science analysis, visualization, and informatics lab-space. *Cell Genomics*, 2(1):100085, 2022. 10.1016/j.xgen.2021.100085.

[SRSC+22] Stian Soiland-Reyes, Peter Sefton, Mercè Crosas, Leyla Jael Castro, Frederik Coppens, José M. Fernández, Daniel Garijo, Björn Grüning, Marco La Rosa, Simone Leo, Eoghan Ó Carragáin, Marc Portier, Ana Trisovic, RO-Crate Community, Paul Groth, and Carole Goble. Packaging research artefacts with ro-crate. *Data Science*, 5:97–138, 2022. 10.3233/DS-210053.

[Var14] Mary Vardigan. The DDI matures: 1997 to the present. *IASSIST Quarterly*, 37(1-4):45–45, 2014. 10.29173/iq501.

[WDA+16] M. Wilkinson, M. Dumontier, IJ Aalbersberg, et al. The FAIR guiding principles for scientific data management and stewardship. *Scientific Data*, 3(1), 03 2016. 10.1038/sdata.2016.18.

[WHN+19] Anna J Woodhead, Christina C Hicks, Albert V Norström, Gareth J Williams, and Nicholas AJ Graham. Coral reef ecosystem services in the anthropocene. *Functional Ecology*, 33(6):1023–1034, 2019. 10.1111/1365-2435.13331.

[WK00] Stuart L Weibel and Traugott Koch. The Dublin core metadata initiative. *D-lib magazine*, 6(12):1082–9873, 2000. 10.1045/december2000-weibel.

[WLD+17] Nick Weber, David Liou, Jennifer Dommer, Philip MacMenamin, Mariam Quiñones, Ian Misner, Andrew J Oler, Joe Wan, Lewis Kim,

Meghan Coakley McCarthy, Samuel Ezeji, Karlynn Noble, and Darrell E Hurt. Nephele: a cloud platform for simplified, standardized and reproducible microbiome data analysis. *Bioinformatics*, 34(8):1411–1413, 10 2017. 10.1093/bioinformatics/btx617.

[WM10] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010. 10.25080/Majora-92bf1922-00a.

[YLL+20] Liying Yu, Tangcheng Li, Ling Li, Xin Lin, Hongfei Li, Chichi Liu, Chentao Guo, and Senjie Lin. SAGER: a database of Symbiodiniaceae and Algal Genomic Resource. *Database*, 2020, 07 2020. 10.1093/database/baaa051.

[YWF+18] Jasmine Y Young, John D Westbrook, Zukang Feng, Ezra Peisach, Irina Persikova, Raul Sala, Sanchayita Sen, John M Berrisford, G Jawahar Swaminathan, Thomas J Oldfield, Aleksandras Gutmanas, Reiko Igarashi, David R Armstrong, Kumaran Baskaran, Li Chen, Minyu Chen, Alice R Clark, Luigi Di Costanzo, Dimitris Dimitropoulos, Guanghua Gao, Sutapa Ghosh, Swanand Gore, Vladimir Guranovic, Pieter M S Hendrickx, Brian P Hudson, Yasuyo Ikegawa, Yumiko Kengaku, Catherine L Lawson, Yuhe Liang, Lora Mak, Abhik Mukhopadhyay, Buvaneswari Narayanan, Kayoko Nishiyama, Ardan Patwardhan, Gaurav Sahni, Eduardo Sanz-García, Junko Sato, Monica R Sekharan, Chenghua Shao, Oliver S Smart, Lihua Tan, Glen van Ginkel, Huanwang Yang, Marina A Zhuravleva, John L Markley, Haruki Nakamura, Genji Kurisu, Gerard J Kleywegt, Sameer Velankar, Helen M Berman, and Stephen K Burley. Worldwide protein data bank biocuration supporting open access to high-quality 3d structural biology data. *Database*, 2018, January 2018. 10.1093/database/bay002.