THE UNIVERSITY OF CHICAGO

AUTOMATED METADATA EXTRACTION CAN MAKE DATA SWAMPS MORE NAVIGABLE

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
TYLER J. SKLUZACEK

CHICAGO, ILLINOIS
AUGUST 2022

For Lauren and Baby Skluzacek

“Nothing in life is to be feared, it is only to be understood.

Now is the time to understand more, so that we may fear less.” — Marie Curie

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I have many people to thank for enabling me to successfully write and defend this thesis.

To my advisors, Kyle Chard and Ian Foster, thank you for providing a consistent stream of advice, paper edits, and opportunities. I would not trade my graduate experience for any other; we conducted exciting research, published papers, traveled the world, and experienced the rapid growth of Globus Labs. I have grown immensely as a researcher between asking Kyle "what is metadata" (in week 1) to my recent defense of this thesis (in week 300).

To committee members Michael Franklin and Raul Castro-Fernandez, thank you both for providing key insights that strengthened nearly every part of this work.

To Ryan Chard, Zhuozhao Li, Logan Ward, and Yadu Babuji, thank you for your limitless mentorship, technology support, friendship, and most of all, patience. I do not think it is an exaggeration to say that I learned more from the "Zhuozhao office" than from any course or paper review.

To Roseylne Tchoua, the exceptional student before me whose actions were always as wise as her words, thank you for helping me stay grounded throughout the many oddities faced by an early graduate student.

To fellow Globus Labs students Hong, Tak, Matt, Ricardo, Sam, Aswathy, Greg, Max, and Marcus, thank you for the friendship, homework help, general commiseration, and travel adventures.

To the truly exceptional undergraduate and high school research assistants who did far more than what was asked of them, thank you for your exciting technical contributions to this work: Tyler Groshong, Ryan Wong, Rohan Kumar, Matthew Chen, Paul Beckman, and Erica Hsu. I am confident that each of you will aspire to great things (computationally and otherwise!)

To others who have contributed to my well-being or this thesis in a significant way, thank you: Eamon Duede, Anna Woodard, Nick Saint, Ben Blaiszik, Pranav Subramanian, Jim

Pruyne, Will Brackenbury, Stephen Rosen, the Globus team, and the (late) Computation Institute.

To the most influential educators whose lessons stayed with me well beyond their classrooms, thank you: Ted Doyle, Jen Davidson, and Brian Felten from Montgomery-Lonsdale High School; and Pete Ferderer, Gary Krueger, David Shuman, Vittorio Addona, and Libby Shoop from Macalester College.

To my family, thank you for supporting me in the most important ways. And finally, to Discrete Math—the class so nice, I took it twice.

ABSTRACT

In a science utopia, every research repository would be accompanied by a database of rich, searchable metadata that users can quickly and confidently query to discover, retrieve, and organize the many artifacts of research workflows. In practice, science is far from this utopia; repositories commonly decay into disorganized data swamps that overwhelm scientists and result in crucial research data being inaccessible to those who could use them. To dredge data swamps, I describe an automated metadata extraction system for science—Xtract—that crawls large repositories, dynamically constructs extraction workflows by intelligently mapping extractors to diverse file types, scalably executes these workflows on distributed research cyberinfrastructure, and publishes the derived metadata into a search index. I show via a user study that an Xtract-generated search index drastically increases the speed and confidence with which researchers navigate their science collections. Finally, I highlight the benefits of this approach by applying Xtract to real-world repositories collectively spanning over 6 million files and 1PB of data across materials science, climate science, battery modeling, and spectroscopy repositories.

CHAPTER 1

INTRODUCTION

The research data lifecycle [5] undeniably drives modern science; the acquisition, cleaning, use, publication and preservation of data leads to a steady stream of research artifacts. To fully realize the utility of these artifacts, scientists have begun to adopt the FAIR (Findable, Accessible, Interoperable, Reusable) research data principles [136, 20, 29] that make data broadly reusable and able to be rigorously validated outside the context of their creation. Open, richly populated search indexes have become beacons by which data practitioners can provide to users not only the data, but also information as to the context of their creation and use. Search indexes enable users to swiftly *navigate*—to freely find and discover data within [12]—vast research collections by enabling users to interact with consistent metadata representations of each file. Metadata are especially important in the context of science data, where files can be abundant, large, and in complex formats [63] that are not compatible with modern search index technologies.

In order to populate search indexes, data practitioners require a mechanism to extract metadata from files. Unfortunately, manual methods where data creators manually tag files with underlying attributes are increasingly infeasible due to the steadily increasing size of science data. Rather than expecting humans to manually parse billions of files [45] potentially spanning exabytes [140], *metadata extraction systems* can automatically generate metadata records for repositories.

While metadata extraction systems have gained prominence over the past few decades, the automated extraction of metadata from scientific repositories remains a challenging task. Metadata extraction itself has existed as long as users have navigated files, from the days of early science where data curators would record metadata to better organize cabinets filled with paper files. In the mid-20th century, the rapid digitization of research data led to automated extraction methods; an early example being the retrieval of basic physical file metadata from the UNIX file system [96]. With modern advances in machine learning, users can now automatically extract rich, semantic information from images [61] and text [59], with data potentially spanning many storage systems [100, 74]. Despite well-defined extraction methods, efforts to create a system that enables data navigation for arbitrary scientific communities and data sets have stalled, in part, due to the extreme challenges in creating a one-size-fits-all extraction system. Specifically, challenges exist not only around variety, scale, and decentralization, but also in quantifying the utility of metadata in enhancing users' repository navigation capabilities. A metadata extraction system, in order to adequately process science data, should be capable of addressing each of these challenges.

Unfortunately, no prominent extraction system simultaneously addresses the challenges brought about by the scale and decentralization of data, the variety and 'messiness' of in-the-wild scientific data (i.e., those that might not adhere to a specific schema, file extension, or MIME type), nor does the accompanying literature evaluate metadata in the context of utility [31] (see Table 1.1 for a brief comparison of metadata extraction systems). For instance, Clowder [78] and Tika [79] both present a rich, broad library of *extractors*—lightweight software scripts present in most extraction systems that input a file and output metadata—but lack automated scaling mechanisms that can adequately address distributed data. Science-Search [98] and BDQC [27] extract high-quality metadata, but only from select formats and science domains.

This thesis presents and evaluates a new automated metadata extraction system, Xtract, which can better address the challenges present in scientific repositories, in order to maximize end-users' collective ability to navigate their data. Doing so requires innovation at each level of metadata extraction, from high-level decisions as to the types of metadata to be extracted; to low-level optimizations that intelligently select and prioritize extractors and enable scale on a computing system; to evaluating the utility of outputs via both machine and human feedback. While many extraction systems focus on a specific scientific use case, I build a

Table 1.1: Taxonomy of metadata extraction systems. I illustrate differences in systems' mechanisms for scaling extractions (`Parallel`), whether they require the transfer of data from the edge to a centralized compute resource (`Central`), their strategy for mapping extractors to files (`Mapping`), whether they provide a metadata utility analysis for automatically extracted metadata (`Utility`), and the supported science domains (`Domains`).

System	**Parallel**	**Central**	**Mapping**	**Utility**	**Domain**
Tika [79]	Threads	No	extension, MIME type, byte-matches	None	general
Clowder [78]	Cloud	Yes	MIME type	None	general
BDQC [27]	None	Yes	input schema	None	biomedicine
Constellation [131]	Cloud	Yes	input schema	None	general
ScienceSearch [98]	Cluster	Yes	input schema	None	microscopy
Xtract	Cluster, Cloud	No	FTI	Yes	general

modular, domain-independent extraction system that ensures navigation is achievable across scientific domains, including materials science, climate science, and biomedicine. Further, I have published the software necessary for institutions to leverage the ongoing findings of this work.

This thesis is organized into 5 research goals that I explored when constructing a new metadata extraction system for science:

1. Extraction from diverse file types
2. Intelligent application of extractors to files
3. Automation and scale of extraction workflows
4. Evaluation of metadata on repository navigability
5. Xtract: the metadata extraction system for science

1.1 Thesis Statement

Automated metadata extraction makes science data swamps more navigable. Navigability immediately stems from the metadata itself, and specifically whether the data catalog contains the information necessary for repository stakeholders to accomplish necessary research tasks. I posit that given a properly-populated catalog and some interface for using it, that users should be able to *better* (more correctly, quickly, or confidently) navigate their data than via their best alternative approaches.

Given the challenges of science data, however, specialized methods are needed to even populate the data catalog. Without a metadata extraction system that can simultaneously handle the complexity, decentralization, and scale of science files, constructing such a catalog becomes impossible. I posit that a system capable of efficiently extracting metadata from real repositories that uses scientists' research cyberinfrastructure can reasonably populate data catalogs.

Thus, if I show that automatically extracted metadata promote science navigability *and* the metadata can be reasonably extracted given the real-world challenges of science data, then I validate my thesis.

1.2 Contributions

The primary contribution of this thesis is the creation and evaluation of a decentralized metadata extraction system, Xtract, which enables repository navigability by assembling and executing intelligent, scalable, and customizable metadata extraction workflows optimized for large, heterogeneous, and distributed scientific data. In building, optimizing, and evaluating this system over the past six years, I discovered and addressed research gaps in each phase of metadata extraction as they relate to the navigability of science data, which has rarely been studied in the context of metadata extraction systems. This thesis offers key insights into designing and evaluating extraction systems, both in regard to system performance and users' data utility. I present findings in the context of Xtract, compare and integrate individual components from alternative systems when necessary, and introduce software libraries that allow users to interface with Xtract. In more detail, my contributions for each of the five research points are as follows.

■ **1. Extraction from diverse file types.** I first define and apply a new set of extractor development principles for science data. The requirements of other systems are generally vague, oftentimes mirroring general open source software principles [126] (e.g., commit, review, and be mindful). Therefore, extractors constructed according to these principles can fail to adequately address the intricacies of science data, namely scale, idiosyncracies in data formats [79, 42], and the subjective and ever-changing needs of expanding communities of scientists [87]. All prominent extraction systems have a breadth of extractors that input files of a *type* that can be subjective in scope (e.g., *all images* versus *photographs of maps*) and output information about them. By observing other systems' extractor libraries, I find that there are two main classes of extractors: those that process file types common to most users (e.g., images) and those relevant to a niche community (e.g., Photoshop project files) [78, 79]. Xtract's extractor library also supports this duality of purpose, as I have overseen the design of a broad set of extractors encompassing file types that can be generally applied across disparate science domains (e.g., tabular, free-text, hierarchical data format (HDF), and JSON files) as well as those specific to narrow science use cases (e.g., battery modeling, spectroscopy, materials science, and climate science).

■ **2. Intelligent application of extractors to files.** I next leverage file type identification (FTI) methods and metadata quality metrics [68, 9] to design an extractor scheduler capable of prioritizing the application of extractors that produce ideal metadata, subject to interchangeable definitions of “ideal”. Curators of data catalogs desire metadata that are semantically searchable, uniquely findable, complete, or simply large [51]. Processing most or all files containing valuable metadata is challenging in resource-constrained computing environments as one cannot reasonably execute every extractor on every file; one must prioritize extractions likely to return high-quality metadata. This prioritization is accomplished in two phases: (1) predict which file-extractor pairs are likely to yield *any* metadata, and (2) of those, identify pairs that most likely contain metadata of the highest *quality*.

In the first phase, extraction systems must predict which extractors are likely to yield *any* metadata from each file. Such a problem is well-suited for FTI methods, as one can think of applicable metadata extractors as a file’s “types”. The leading FTI models from previous work [48] generally leverage lightweight statistical learning algorithms (e.g., logistic regression, random forests, support vector machine) trained on quickly accessible physical features of files such as the size and byte samples. Current extraction systems do little more than determine a file’s type via name, MIME type, and byte patterns, which I show in Chapter 4 to be unreliable in recognizing unfamiliar science formats and files of multiple types (e.g., a tabular CSV file with a multiline free-text preamble). FTI has historically been confined to use cases in digital forensics [92] and malware detection [121], so metadata extraction presents a new use case.

In the second phase, I feed predicted file types to an extraction scheduler that prioritizes files based on expected metadata quality (in this thesis, I consider both yield and completeness). I observe that a scheduler optimizing metadata yield per second most capably prioritizes quality extractions (even subject to alternative metrics). Therefore, a metadata extraction system using this scheduler can mine a majority of a repository’s quality metadata objects in a fraction of the total extractor executions or processing time.

■ **3. Automation and scale of extraction workflows.** Applying dozens of extractors to millions of files can require significant computing power. Additionally, the files may be distributed across multiple storage endpoints and be subject to processing on diverse research cyberinfrastructure, such as computing clusters and clouds. This thesis uniquely explores the challenges present in applying extractors to files (1) at scale and (2) distributed across storage systems. To this end, I introduce several optimizations, including grouping files to avoid duplicate between-endpoint transfers (the *min-transfers* algorithm), offloading data elements from congested to idle computing resources, and warming containers on each resource to reduce cold-start costs. To showcase the scale of these methods, I detail an experiment in which Xtract processes the Materials Data Facility data set (2.3 million files; 19TB) in approximately 6 hours using the Theta supercomputer at Argonne National Laboratory. To illustrate Xtract's ability to flexibly handle decentralized data, I illustrate system performance on a continuum of computing scenarios: all data must be computed at the edge, all data must be computed centrally, and hybrid approaches that can be computed anywhere.

■ **4. Evaluation of metadata on repository navigability.** While automated approaches for measuring metadata quality can help evaluate and compare extraction plan schedules, these metrics cannot directly assess whether the extracted metadata provide value to users' research tasks. To evaluate users' perceived value, this thesis explores whether automatically extracted metadata make repositories more navigable, as measured by users' performance and confidence in performing repository search and discovery tasks relevant to their work. I conducted an IRB-approved mixed methods user study on users of large national lab science repositories. I find that automatically extracted metadata, regardless of the search interface, enable users to correctly complete 28% more simulated research tasks, and to perform these tasks significantly faster (>10×, on average) than via their best alternative approaches. Importantly, I find that 100% of participants claim that the automatically extracted metadata are not only helpful, but *more helpful* than their existing approaches in navigating their science data. I also use this opportunity to collect qualitative feedback about Xtract's metadata.

■ **5. Xtract: the metadata extraction system for science.** As a major part of this thesis, I document and publish an initial version of the software that institutions, repositories, science groups, and even individuals can use to leverage Xtract and, by extension, the workflows and optimizations in this thesis. To this end, I create (1) an extractor creation and submission library, (2) a software development kit (SDK) that enables users to connect directly to a hosted Xtract system and to manage the returned metadata attributes, (3) a command-line interface (CLI) that lets users configure their own compute resources as endpoints, and (4) thoroughly documented file type identification and metadata quality toolboxes that can be used to aid extractor selection and evaluation, both in Xtract and otherwise.

1.3 Thesis Organization

Chapter 2 presents fundamental related work in metadata extraction systems. Chapter 3 defines the main software unit of metadata extraction—extractors—and discusses how one should create an extractor library that suits both the needs of users and the machine. Chapter 4 presents an approach for designing intelligent extraction plans that leverage statistical analysis to predict the necessary extractors for each file, which is especially important in the context of resource-constrained computing systems. Chapter 5 introduces and evaluates the Xtract metadata extraction system, which is capable of applying extraction plans to files over remote research cyberinfrastructure. Chapter 6 presents the results of a user study of national laboratories' science storage users in which I evaluated the extent to which Xtract's metadata extraction workflows enable navigability of their repositories. Chapter 7 pivots to discuss how I have packaged Xtract for use by the computational science community. Finally, I discuss future work and concluding remarks in Chapters 8 and 9, respectively.

CHAPTER 2
RELATED WORK

Metadata extraction systems have a strong foundation in the scientific data management literature. In this section, I outline related work that has led to the creation, evaluation, and use of leading metadata extraction systems. I have placed related work in other vital areas—schema inference, file type identification, metadata quality, remote execution systems, and data navigation—into the corresponding chapter(s) to enhance readability.

2.1 Metadata extraction systems

Data, in the absence of information concerning content and relationships, are just assemblages of bytes. This reality has spurred much work on methods for extracting or synthesizing the information that people need to navigate such assemblages. A few information mining analyses can proceed without domain knowledge. For example, file-level deduplication [81], commonly applied in storage systems to reduce storage requirements [52], looks only at byte sequences in files to determine the inter-file relationship "are equivalent." In general, however, semantic information is needed to make sense of data. Manual creation and maintenance of such metadata [66, 95, 135, 139] is time-consuming, even if required expertise is available [123, 19]. In the general case—when scientific data are large, heterogeneous, and potentially distributed—automated methods become necessary.

To alleviate these challenges, researchers have developed methods to extract standard metadata from nonstandard file types and formats [125]. Furthermore, a number of systems have been developed to automatically extract and organize metadata from files. In the following, I review several such systems and compare them to the focal system of this thesis: Xtract.

Infoharness [109, 104] (first published in 1999) represents one of the earliest known metadata extraction systems. They automatically parse structured file formats, and do so in a decentralized manner. They have a finite number of extractors that are automatically applied, including free text, tabular, images, emails, and code artifacts. It is unknown both how Infoharness maps extractors to files and the scaling performance of their approach.

Apache Tika [79] (first published in 2007) is an open source metadata extraction toolkit and extractor library originally constructed to support the Apache Nutch web crawler [88]. Tika has an extensible parser interface for developing custom parsers, and disparate science organizations have adopted it for extraction purposes. With its widespread adoption, Tika's community has developed a rich extractor suite. Tika's default parser libraries can recognize thousands of file formats, making it a rich source of extractors for use within Xtract. A limitation is that the choice of parsers to apply to a file is made primarily on the basis of MIME types, which are often misleading in scientific data sets, where for example MIME type 'text/plain' may be used for both tabular and free text files. The Tika libraries are also used by the GEMMS [94] metadata extraction system to identify parsers for extracting property, structure, and semantic metadata; thus GEMMS suffers from similar limitations to Tika when applied to scientific data.

The Clowder [78] data management system (first published as Brown Dog in 2015) supports data curation and metadata extraction for scientific data. Like Xtract, it uses containers for extensible and scalable metadata extraction. Clowder stores metadata records in an ElasticSearch index to make them discoverable.

Constellation [131] (first published in 2016) is a centralized metadata extraction and storage system that extracts entities—research groups, machines, experiments and files—from scientific data. It provides simple extractors for self-described hierarchical metadata, HPC logs, and file systems. Also relevant to Xtract is work on pay-as-you go information integration systems, which allow for incremental improvements to semantic mappings between data elements, as and when users decide that further investment in data analysis is required [24].

CLAMS [33] (first published in 2016) sweeps over large data lakes to pinpoint errors in raw enterprise data, which can be either unstructured and semi-structured. Unlike BDQC and Xtract, CLAMS does not treat data as immutable objects—it actively loads and transforms them into schema-less RDF triples for storage in HDFS. To identify anomalies across the RDF triples, CLAMS constructs a hypergraph of data subject to constraint rules, and looks for 'violations': combinations of data that cannot co-exist (and likely represent an error). This means, however, that the entirety of all data constraints ought to be known at extraction time, which may not be possible on data lakes with less stringent organizational requirements.

ScienceSearch [98] (first published in 2018) uses machine learning techniques to create metadata for micrographs in a National Center for Electron Microscopy (NCEM) dataset, with additional context derived from associated artifacts, such as file system data and free text proposals and publications. Like Xtract, ScienceSearch allows users to switch metadata extractors to suit particular datasets. However, it too requires that extractions be performed where data reside.

The Big Data Quality Control (BDQC) Framework [27] (first published in 2018) is a centralized bulk metadata extraction system that enables search across large collections of biomedical data. BDQC is similar to Xtract in that it performs systematic 'bulk' sweeps across heterogeneous data and introspects files without regard to their meaning (domain-blind analysis), but with the primary goal of identifying anomalies. BDQC employs a pipeline of extractors to derive the properties of imaging, genomics, and clinical data. While BDQC is implemented as a standalone system, the approach taken would be similarly viable in Xtract.

In the remainder of this thesis, I continuously reference these metadata extraction systems as a basis of discussion for Xtract's design and performance.

CHAPTER 3
EXTRACTION FROM DIVERSE FILE TYPES

As scientific data repositories can contain millions of files spanning multiple petabytes (e.g., Argonne's Petrel [1] offers 3.2 PB of storage capacity), manual metadata annotation becomes virtually impossible. Thus, scientists require automated methods for extracting information from files regarding their content and creation. When creating files, however, scientists generally do not adhere to a universal, well-defined structure, or *schema*. For example, two tabular files (i.e., files resembling a "spreadsheet" with rows and columns) might have different row-column dimensions, inconsistent header-labeling schemes, or non-uniform precision on similar fields. To consistently extract metadata from similar files of various schema, *metadata extractors* should be able to robustly identify and parse minor schema differences.

A ***metadata extractor*** is a specialized program that *derives* and/or *synthesizes* metadata information only from files adhering to a certain set of schema requirements collectively referred to as its ***type***. Each extractor processes files of a single type (e.g., a `tabular` extractor processes files containing row-column 'spreadsheet' formats), but each file can have multiple types (thereby requiring multiple extractors to process). To *synthesize* metadata means to cherry-pick latent information from data—for instance, to find numeric aggregates (means) or the boolean (true/false) existence of the keyword 'precipitation'. Alternatively, to *derive* metadata means to create a new representation of existing data—for instance, using a machine learning model to identify a 'dog' in a photograph, or labelling certain keywords as 'relevant'. Each extractor has two components—(i) a function to perform the derivation and/or synthesis, and (ii) a required virtual environment (including a set of software dependencies)—that input a file or group of files, and output a metadata document. In the following, I refer to one or more files simultaneously processed by a metadata extractor as a **file group**. Metadata extractors vary in engineering complexity; from simple extractors that synthesize information from filenames, to more complex extractors that input file contents into complex machine learning models. In designing a metadata extraction system, I consider the challenges in creating a one-size-fits-all extractor library by outlining key principles by which metadata extractors should abide. I detail these extractor principles in more detail in §3.2.

A primary goal of metadata extraction systems is to construct an ***extraction plan***, or a workflow of extractors that each return non-empty and non-negligible metadata information from each file group. To avoid wasting compute resources, extraction plans should avoid invoking each extractor on every file by identifying the type(s) of a file group, and applying only extractors that map to those types [115, 4]. I outline all file types considered in this work in Figure 3.1. A complex (but common) extraction plan occurs when a file group is of more than one type, thereby emitting non-negligible metadata from multiple extractors. I henceforth refer to these extractions as ***hybrid*** *extractions*. I present two examples of common multi-typed files requiring hybrid extractions in the following:

1. **Tabular and Free-Text**: this multi-typed file is prominent in the sciences, and has been at the center of significant CSV-parsing work [28, 84]. Such files have two clear components: first, a free-text preamble describing the experiment—such as the date an experiment occurred, instrument settings, and citation information (e.g., authors and paper titles)—and second, traditional tabular data (header, rows, and columns) underneath. Figure 3.2 shows a file matching this structure. In this example, I consider the free-text preamble of the file to be of type `freetext` and everything underneath (the header row and row-column contents) to be of type `tabular`. A hybrid extraction plan for this file should include extractors catering to both types.

2. **Image and Photograph**: significant research in entity recognition enables the representation of photographs as hierarchical types—if one can first classify a file to be of type `image`, there can exist unique processing pipelines for each subtype. In the sciences, I consider common sub-types of images to be photographs, maps, and scientific

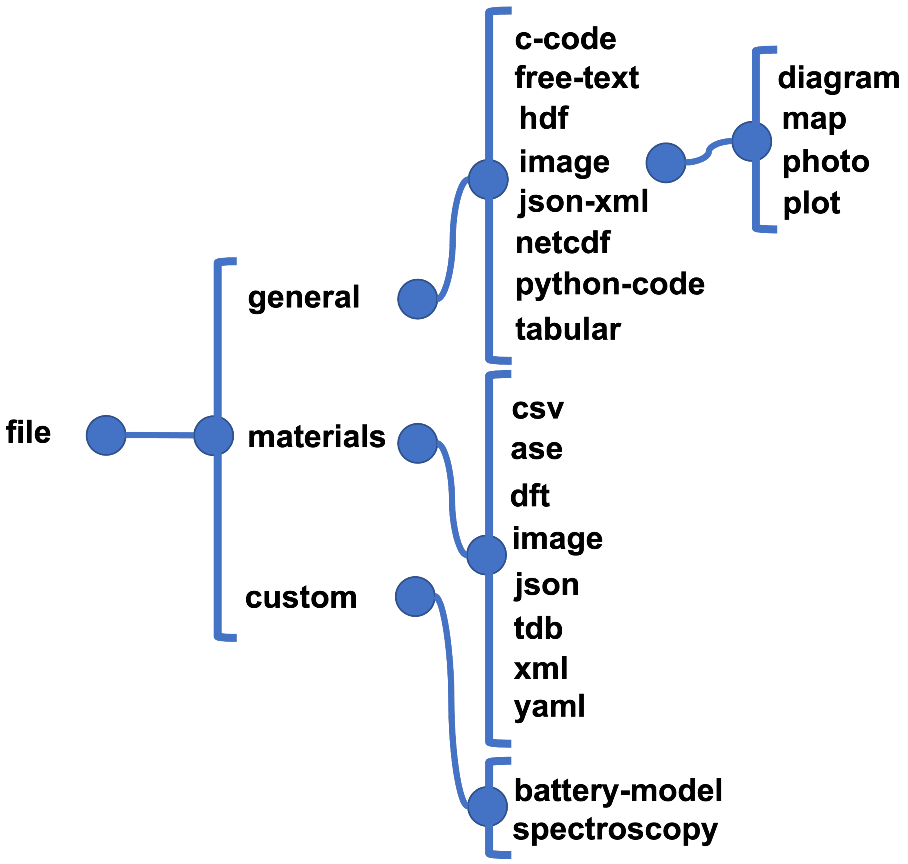
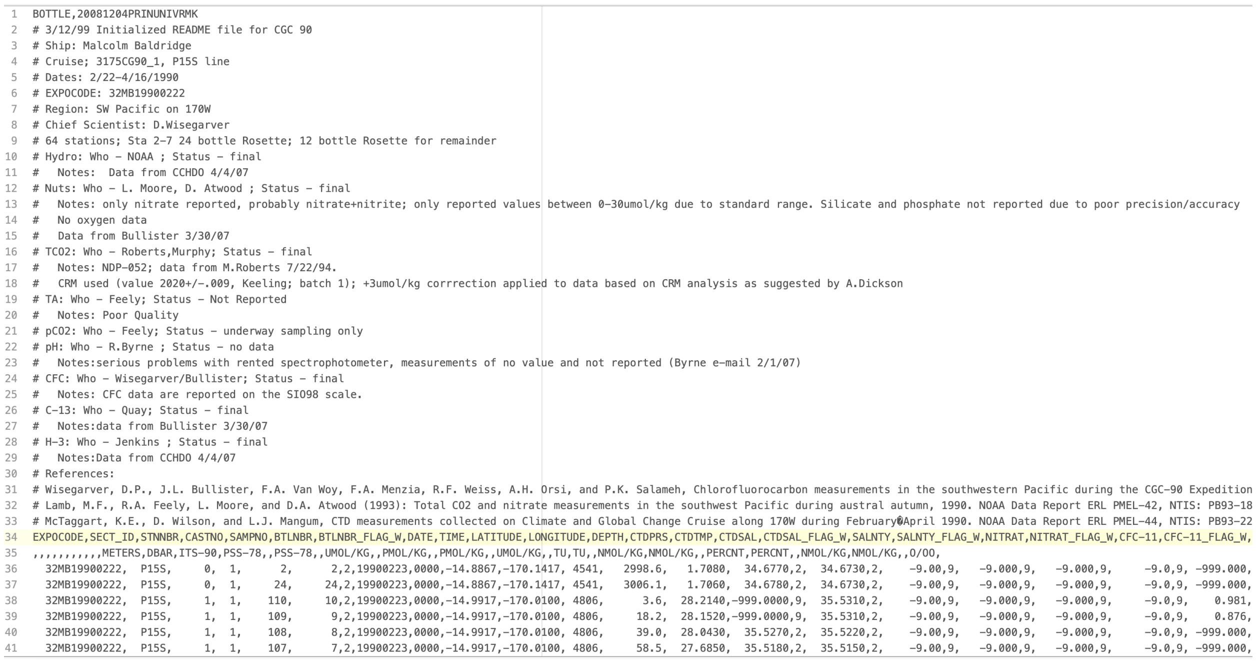


Figure 3.1: **Tree diagram of types able to be processed by the library of metadata extractors discussed in this thesis**. Following the diagram from left-to-right leads to increasingly granular file types, from all files being a `file`, all the way to a specific cross-section of files being a particular sub-type (e.g., `image.map`)



```
BOTTLE,20081204PRINUNIVRMK
# 3/12/99 Initialized README file for CGC 90
# Ship: Malcolm Baldridge
# Cruise; 3175CG90_1, P15S line
# Dates: 2/22-4/16/1990
# EXPOCODE: 32MB19900222
# Region: SW Pacific on 170W
# Chief Scientist: D.Wisegarver
# 64 stations; Sta 2-7 24 bottle Rosette; 12 bottle Rosette for remainder
# Hydro: Who - NOAA ; Status - final
#   Notes:  Data from CCHDO 4/4/07
# Nuts: Who - L. Moore, D. Atwood ; Status - final
#   Notes: only nitrate reported, probably nitrate+nitrite; only reported values between 0-30umol/kg due to standard range. Silicate and phosphate not reported due to poor precision/accuracy
#   No oxygen data
#   Data from Bullister 3/30/07
# TCO2: Who - Roberts,Murphy; Status - final
#   Notes: NDP-052; data from M.Roberts 7/22/94.
#   CRM used (value 2020+/-.009, Keeling; batch 1); +3umol/kg corrrection applied to data based on CRM analysis as suggested by A.Dickson
# TA: Who - Feely; Status - Not Reported
#   Notes: Poor Quality
# pCO2: Who - Feely; Status - underway sampling only
# pH: Who - R.Byrne ; Status - no data
#   Notes:serious problems with rented spectrophotometer, measurements of no value and not reported (Byrne e-mail 2/1/07)
# CFC: Who - Wisegarver/Bullister; Status - final
#   Notes: CFC data are reported on the SIO98 scale.
# C-13: Who - Quay; Status - final
#   Notes:data from Bullister 3/30/07
# H-3: Who - Jenkins ; Status - final
#   Notes:Data from CCHDO 4/4/07
# References:
# Wisegarver, D.P., J.L. Bullister, F.A. Van Woy, F.A. Menzia, R.F. Weiss, A.H. Orsi, and P.K. Salameh, Chlorofluorocarbon measurements in the southwestern Pacific during the CGC-90 Expedition
# Lamb, M.F., R.A. Feely, L. Moore, and D.A. Atwood (1993): Total CO2 and nitrate measurements in the southwest Pacific during austral autumn, 1990. NOAA Data Report ERL PMEL-42, NTIS: PB93-18
# McTaggart, K.E., D. Wilson, and L.J. Mangum, CTD measurements collected on Climate and Global Change Cruise along 170W during February�April 1990. NOAA Data Report ERL PMEL-44, NTIS: PB93-22
EXPOCODE,SECT_ID,STNNBR,CASTNO,SAMPNO,BTLNBR,BTLNBR_FLAG_W,DATE,TIME,LATITUDE,LONGITUDE,DEPTH,CTDPRS,CTDTMP,CTDSAL,CTDSAL_FLAG_W,SALNTY,SALNTY_FLAG_W,NITRAT,NITRAT_FLAG_W,CFC-11,CFC-11_FLAG_W,
,,,,,,,,,,,METERS,DBAR,ITS-90,PSS-78,,PSS-78,,UMOL/KG,,PMOL/KG,,PMOL/KG,,UMOL/KG,,TU,TU,,NMOL/KG,NMOL/KG,,PERCNT,PERCNT,,NMOL/KG,NMOL/KG,,O/OO,
  32MB19900222,  P15S,     0,  1,      2,      2,2,19900223,0000,-14.8867,-170.1417, 4541,   2998.6,   1.7080,  34.6770,2,  34.6730,2,    -9.00,9,   -9.000,9,   -9.000,9,     -9.0,9, -999.000,
  32MB19900222,  P15S,     0,  1,     24,     24,2,19900223,0000,-14.8867,-170.1417, 4541,   3006.1,   1.7060,  34.6780,2,  34.6730,2,    -9.00,9,   -9.000,9,   -9.000,9,     -9.0,9, -999.000,
  32MB19900222,  P15S,     1,  1,    110,     10,2,19900223,0000,-14.9917,-170.0100, 4806,      3.6,  28.2140,-999.0000,9,  35.5310,2,    -9.00,9,   -9.000,9,   -9.000,9,     -9.0,9,    0.981,
  32MB19900222,  P15S,     1,  1,    109,      9,2,19900223,0000,-14.9917,-170.0100, 4806,     18.2,  28.1520,-999.0000,9,  35.5310,2,    -9.00,9,   -9.000,9,   -9.000,9,     -9.0,9,    0.876,
  32MB19900222,  P15S,     1,  1,    108,      8,2,19900223,0000,-14.9917,-170.0100, 4806,     39.0,  28.0430,  35.5270,2,  35.5220,2,    -9.00,9,   -9.000,9,   -9.000,9,     -9.0,9, -999.000,
  32MB19900222,  P15S,     1,  1,    107,      7,2,19900223,0000,-14.9917,-170.0100, 4806,     58.5,  27.6850,  35.5180,2,  35.5150,2,    -9.00,9,   -9.000,9,   -9.000,9,     -9.0,9, -999.000,
```

Figure 3.2: Example multi-typed file with (i) free-text preamble containing descriptive experiment information and (ii) tabular data containing quantitative measurements. From CDIAC's publicly-accessible Global Ocean Data Analysis Project (GLODAP) ([72]).

diagrams. For this example, consider a photograph of household objects, as shown in Figure 3.3. An extractor library might have one extractor to identify an image's type, and once it is determined to be a photograph, a second extractor to identify objects therein. One should note that there is no extraction plan that includes a `photograph` extractor, but *does not include* the more-general image extractor.

For interested readers, I provide a more-formal definition of the artifacts of metadata extraction—for instance, metadata, repositories, and extraction—in Appendix A. The remainder of this chapter describes how I design metadata extractors such that they can be linked into potentially-hybrid extraction plans. §3.1 describes how I select which extractors should be included in an extraction plan, §3.2 describes the key principles (and tradeoffs thereof) to consider when architecting extractors, and §3.3 outlines interesting extractors integrated within Xtract. §3.4 presents related work. Finally, §3.5 summarizes my contributions.

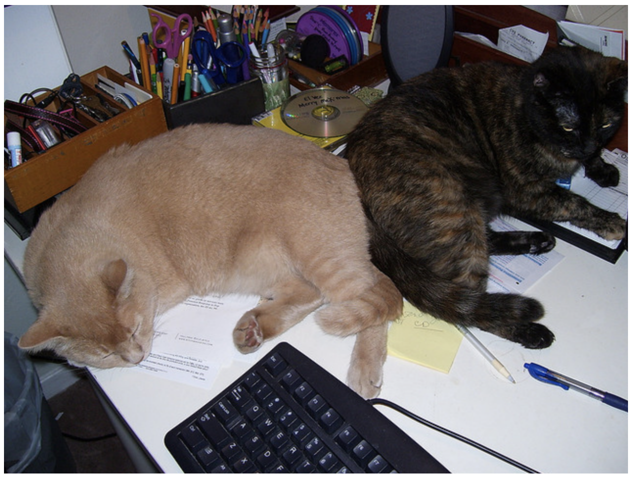


Figure 3.3: Example hierarchically multi-typed file that must (i) be classified as a photograph, and (ii) as a photograph, have entities extracted (e.g., scissors, cats, keyboard). From Common Objects in Common Context (COCO) [76].

3.1 Choosing Extractors for the Library

Determining the extractors to be used in an extraction system is highly dependent on the context: the expected types of files requiring extraction *and* the desired information content of users. As a preliminary heuristic, an extraction system should seek to retrieve some level of rich, descriptive metadata from as many files as possible. Therefore, I formally define ***coverage*** as the percentage of a repository's files validly represented in the resultant metadata corpus, and use this metric to evaluate the usefulness of the extractor library.

Extraction systems generally outline the constitution and role of the community to guide the creation and maintenance of extractors. In one case, Clowder [78] recognizes that "a community will have software that needs to be executed [on files as part of metadata extraction]," and they responded by engaging the community to collaboratively build "a wide variety of extractors" and the PyClowder package for creating and registering custom extractors. Further, Clowder includes a social annotation feature where users manually provide metadata annotations. ScienceSearch [98] creates metadata extractors for mining information from artifacts of a narrow science community, researchers at the National Center for

Electron Microscopy, and they collaboratively design a small set of extractors for (i) the electron microscopy images themselves, and (ii) text data stored near the images. Tika [79] has an active community of users who have contributed over 230 extractors[1] to the system, where users dictate the specific file MIME type on which an extractor should be run. In the case of each system, there exists the need to extract metadata from files by mapping extractors to file types commonly processed by each tool's respective community. To this end, I build upon prior work by designing a suite of independent metadata extractors usable in both community contexts: general science communities (i.e., Tika) and specific science groups (i.e., ScienceSearch).

In designing a suite of metadata extractors, I have leveraged community input to elicit attributes to include in new extractors, and to enhance existing extractors with improved capabilities; from making extractors more scalable or tuning them for lower latency, to enabling extractors to support new schema alternatives for a file type (e.g., add a PDF-parsing library to the `freetext` extractors). Further, I have designed new extractors from existing scientific tooling to bolster existing metadata extraction efforts. For instance, I worked with materials scientists and computer scientists at Argonne National Laboratory to create an extractor suite for materials science data by wrapping the MaterialsIO library's [134] parsers as extractors that were later leveraged to populate searchable attributes within the Materials Data Facility [8].

As an integral part of this thesis, I outline the following community engagement process, inspired by open source community protocols [58], that I have used to construct an extraction system with a robust and useful extractor library:

1. **Interview stakeholders.** Repository stakeholders encompass many roles, including data creators, internal accessors (participants within a research group), external acces-

1. see all applicable Tika MIME types here: `https://tika.apache.org/2.1.0/formats.html#Full_list_of_Supported_Formats_in_standard_artifacts` (230 as of Nov. 30, 2021)

sors (people from other research groups), and curators (those in charge of serving the data to accessors). Each stakeholder has unique *navigability needs* for a repository, and therefore potentially disparate metadata needs. I construct an exhaustive list of user requirements to avoid unnecessary later re-work and to promote better understanding among extractor developers. In full transparency, I bootstrapped the extractor library by designing a number of extractors that I thought would provide value to a general audience (e.g., tabular and free-text), but have recently become a less-biased facilitator of extractor construction and augmentation.

2. **Evaluate fit of existing extractors.** Given the navigability needs of the community, I next evaluate whether the current extractors encompass the underlying metadata requirements. *Do current extractors provide enough decimal points of precision? Are all temperatures represented in both Fahrenheit and Celsius? Do they want to classify a type of image only contained in this repository? Do they need to correctly rank free-text keywords to enable efficient search?* In this step, one should enumerate the ways in which the current extractor library is lacking.

3. **Identify software to enhance metadata.** Are software scripts available to extract or supplement insufficient metadata? Have these software been vetted—by stakeholders or experts—and are known to produce metadata that are both relevant and accurate? Building an extractor requires knowing that the necessary software dependencies (i) exist, and (ii) produce correct results.

4. **Augment current extractors.** Given a list of desired metadata elements and a vetted set of software dependencies capable of solving the problem, can one simply retool existing extractors with the new or improved software in order to provide the necessary metadata? Or must one heavily alter the extractor to a point where it might be better encapsulated as its own extractor (I further discuss the desired scope

of a single extractor in §3.2)? Or perhaps it closely resembles an existing extractor, but has a dependency mismatch that either requires a software compromise *or* a new extractor. Augmenting existing extractors should not alter existing metadata elements, but should instead enable *additional* metadata to be extracted.

5. **Construct new extractors.** If the new software cannot be implemented into an existing extractor, the stakeholders (or I) integrate the new software and dependencies into a new extractor (or if multiple file types, multiple extractors). The extractors are made usable by the existing workflow. Users can do this themselves for their own workflows via the SDK discussed Chapter 7, but inclusion for the entire community will require a proper extractor sharing pipeline.

6. **Evaluate coverage.** To ensure that as many files as possible are represented in a given data catalog, I examine the coverage of metadata elements. After completing extractor augmentation and construction (and after loading the extractor into the extraction system), I perform an end-to-end metadata extraction job of that repository subject to all extractors by counting the number of files that contain non-empty and non-trivial metadata. I create tree diagrams of the uncovered files to examine whether any common data types are unaccounted, or if these files are of specific data types that simply cannot be parsed. Figure 3.4 illustrates a tree diagram of files in the Materials Data Facility (MDF) data repository; in this case coverage can be maximized by constructing extractors for files from large boxes (*.xyz, .tiff, .tif*) to small (*.cif, .sh, .dat*). This process is repeated until metadata coverage is acceptable.

7. **Evaluate navigability (and other quality metrics).** Once it is assumed that the metadata cover a sufficient proportion of a repository's files, the final step is to gauge whether the metadata have value for the community's use cases. I discuss how this can be done via user study in Chapter 6.

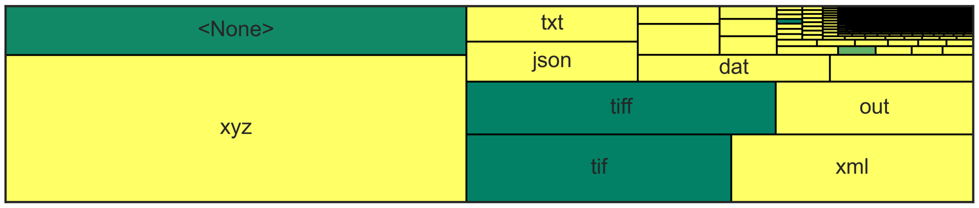


Figure 3.4: Treemap representation of the Materials Data Facility, where the total area of the box is proportional to the number of files of the given extension in the repository. Darker boxes refer to extensions collectively encompassing more disk space.

Extractor	**Description**
Tabular	isolate data elements and compute aggregates for files row-column files
Keywords	identify uniquely descriptive words in unstructured free-text documents
ImageSort	use model to derive an image's type (e.g., photo, map, graph)
ImageNER	utilize Tensorflow ImageNet to retrieve entities in images
JSON/XML	retrieve nesting depth and field information for JSON and XML files
NetCDF	extract dimensional and self-described metadata attributes
Python	retrieve python script attributes: versions, function names, pep8 info
C	retrieve C code attributes: versions, imported headers, docstrings
HDF	extract dimensional and self-described metadata attributes
Maps	extract latitude/longitude and location tags of map images
Spectroscopy	confidential serial crystallography extractor for Argonne National Lab
Materials	suite: atomistic simulations, crystal structures, DFT calculations
Batteries	suite: temperatures, charge/discharge curves, voltage

Table 3.1: Extractor library as of June 2022

In this work, I have identified numerous scientific data repositories and used the aforementioned guidelines to dynamically construct a suite of extractors to address the broad needs of scientists from potentially-disparate domains. I have also gone as far as to calculate the coverage of the extractor library on various data sets—the extractors mine content-specific metadata from over 88% of non-compressed[2] files in a minimally-curated climate science repository [115], 94% of files in a well-curated materials science repository [113], and 91% of files in the personal Google Drive repository of a graduate student [110]. To better illus-

2. A colleague and I address the extraction of compressed files in ongoing work [138], but I consider it to be outside the scope of this thesis.

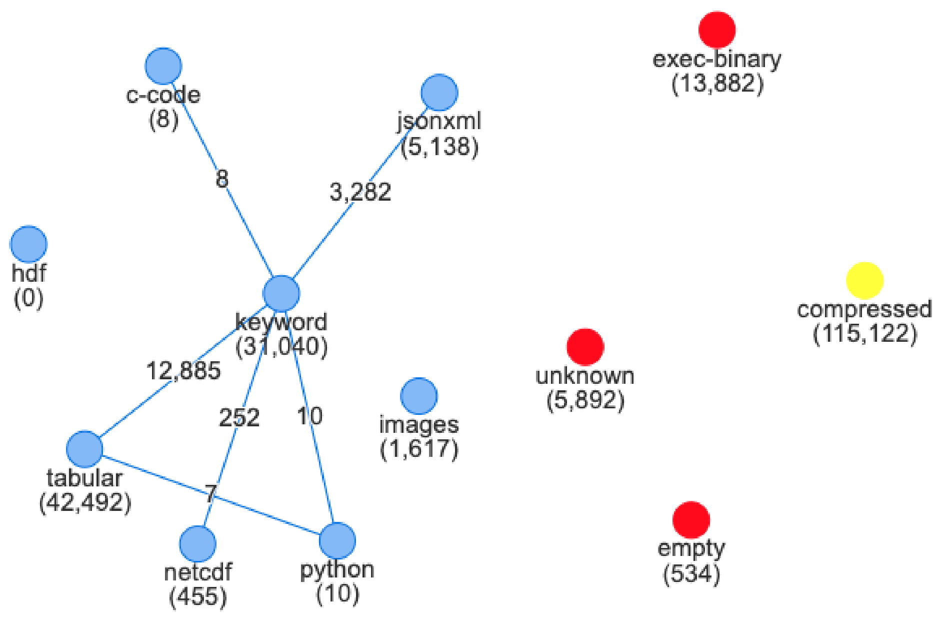


Figure 3.5: Coverage graph of CDIAC data repository (with compressed files), where each node represents a file type encoded as follows (**blue**: extractable type counted in coverage; **yellow**: semi-extractable type that requires file deflation/unzipping in order to process; **red**: un-extractable types. Numbers in parentheses represent the number of files of a given type, and numbers on the edges represent the number of hybrid files that are of the types of both of the adjacent vertices.

trate coverage, I include the graph of all types (extractable; semi-extractable (compressed); unextractable) for CDIAC in Figure 3.5. Over the past five years, I have added over 20 extractors to Xtract's extractor library, illustrated in Table 3.1, that are iteratively edited to encompass new science groups and data.

In this section, I outlined the importance of community in designing an extractor library that can process as many files as possible in a scientific data repository. In the following section, I discuss how—given community input—one can design extractors such that they are maximally effective for both the community and the extraction system.

3.2 Principles for extractor design

The process by which scientists create metadata extractors is generally open-ended—extractors, by definition, are simply (i) a function that inputs a file group and outputs a single metadata document, and (ii) a virtual environment containing a collection of dependencies. To ensure a consistent, quality, and performant metadata extractor library I have compiled the following list of principles by which each extractor abides:

- **P1. Relevant:** extractors should produce metadata that enable an end-user (e.g., search indexes, automated experimental laboratories, human observers) to accomplish a given task. Metadata are relevant if they contain attributes necessary for all users to accomplish tasks, and irrelevant if they do not. For instance, when searching through images to identify a body of diagrams from academic papers, relevant pieces of metadata can include image type, file extension, creation date, or even a thumbnail representation of the image. Irrelevant metadata could include the weather when the diagram was constructed in an editor, the 1991 World Series champions, or the number of vowels in the filename.

  Metadata not useful to one context, however, may be useful to another; I formalize metadata relevance as follows. If M_e is the universe of metadata information outputted from an extractor $e \in E$, $s \in S$ the community stakeholders using an extractor, and $extract(e_s)$ the metadata deemed relevant to a use case for stakeholder s from extractor e, then $\bigcup_{s \in S} extract(e_s) = M_e$. This means that the potential metadata attributes for an extractor are strictly the union of necessary attributes desired by the extractor's stakeholders. I show as an example a metadata object in Listing 1, where one stakeholder (stakeholder s_1) requires 10 unranked, representative free-text keywords about a COVID-19 paper, and another (stakeholder s_2) requires 20 ranked keyword

3. here I use the information theoretic *relevance* first described in 1976, but later implemented in TF-IDF modeling (as is used in the `keyword` extractor), that describes the amount of 'weight' a word or phrase should have in describing a document (i.e., more weight = more relevance). This is also commonly referred to as the *usefulness*.

(weighted by relevance score[3]). The extractor in question is the keyword extractor, denoted $e_{keyword}$. In this example, the metadata is relevant to both stakeholders' needs, because it includes 20 ranked keywords (and by default, also contains the top 10 keywords, as s_1 can choose to ignore the rankings).

```
{
    'keywords': {
        'coronavirus': 283,
        'saliva': 210,
        'volume': 153,
        'venue': 150,
        'middle': 118,
        'assay': 118,
        'specimen': 115,
        'specimens': 100,
        'latex': 88,
        'washington': 82,
        'laboratory': 81,
        'school': 77,
        'affiliation': 75,
        'paired': 72,
        'agreement': 63,
        'original': 61,
        'collection': 60,
        'detection': 56,
        'medicine': 54,
        'positive': 53
    },
    'extract_time': 0.8125491142272949
}
```

Listing 1: Output from **keyword** extractor when invoked on an academic paper about COVID-19. Each keyword is assigned a score that demonstrates its importance, or relevance, in describing the full text document. The paper in question is about the efficacy of using saliva swabs (as compared to nasal swabs) to detect SARS-CoV-2.

- **P2. Correct:** the ability to guarantee metadata correctness is a vitally important feature of a metadata extractor, but such correctness is difficult to manually validate when extractions are automatically conducted at scale. Xtract's extractor library, similarly to Apache Tika, requires that all public metadata extractors include a representative set of files and unit tests on which correctness can be evaluated with each code or dependency change to the extractor. If one could theoretically guarantee a universally representative set of unit tests and test files, then each extractor's metadata would be 100% correct. However, given constant changes to file types, computation methods, and user communities supported by an extractor, achieving consistently high correctness is difficult. Additionally, it may prove too stringent to have a rigid definition of correctness; *if an extractor uses a machine learning model that accurately assigns a metadata label 95% of the time, how does one know if that is correct-enough?*

To combat the aforementioned difficulties in maintaining correctness, Xtract leverages the expertise of domain experts who collectively determine whether an extractor achieves an acceptable level of correctness. I [115] and others [123] have shown in prior work that enlisting domain experts to manually confirm a subset of model results can ease much of the ambiguity of correctness determination. If metadata are deemed to be correct, the extractor is added to the library; otherwise, the domain experts outline ways in which the extractor could be improved, and the process is repeated.

- **P3. Lightweight:** extractors should optimize for time, whenever possible, to avoid wasting scientific computing resources. There are multiple ways an extractor can achieve acceptable, or even minimal, execution latency. First, concurrency should be used to avert waiting for blocking processes when additional work can be done, but should avoid aggressive fan-out parallelism if resources are scarce.[4] For instance, one can load a machine learning model into memory while executing a compute-intensive sequential scan over a file. Second, approximate computing methods (e.g., input data sampling [3]) should be used when possible when full file scans are not absolutely necessary, and the metadata's relevance and correctness are not violated. Third, one can

4. in Xtract, extractors are scaled to match the number of compute cores, and scaling too aggressively (e.g., with Python's `multiprocessing` library) in one extractor instance can take necessary resources from another [110].

simply terminate extraction processes that cross a timeout threshold. This is especially useful as, oftentimes, extractors might take inordinately long when accidentally applied to the wrong type of file (e.g., trying to tokenize keywords in a binary executable). Finally, each extractor should open files *once* to avoid extraneous I/O overhead [93]. As shown in Figure 3.7, opening large files in the CDIAC data set can take upwards of 5 seconds; incurring this cost multiple times is especially wasteful when the accompanying extractor executes in relatively less time (some complete in fractions of a second). Therefore, file I/O should be minimized.

To achieve an acceptable level of extractor performance, I have designed extractors that integrate each of these four latency-reducing mechanisms. Every extractor opens a file just once. The `keyword` extractor uses separate threads to load the `nltk` NLP toolkit, and another to load the file into memory (shown in Figure 3.8b). Additionally, users can (via keyword argument) have the tabular extractor collect approximate aggregates; based on the use case, one may opt to only process a certain percentage of rows from the beginning of the file, or those selected at random from throughout. Finally, the `keyword` extractor terminates execution on a file after a 120-second timeout, and recomputes keywords on the file's first 10KB.

Some extractors, by nature, are faster than others. For instance, an extractor to compute something simple (e.g., a file extension) will take significantly less time than an extractor that computes line-by-line aggregates and feeds said aggregates as features into a machine learning model. In Figure 3.6 I illustrate the distribution of execution times for all general extractors on each file in the 428 000 file Carbon Dioxide Information Analysis Center (CDIAC) dataset, when using the Theta supercomputer at Argonne National Laboratory. One will see that the heavier `keyword` extractor takes more than an order of magnitude longer than the much simpler `netcdf` and `jsonxml` extractors that do not require complex tokenization.

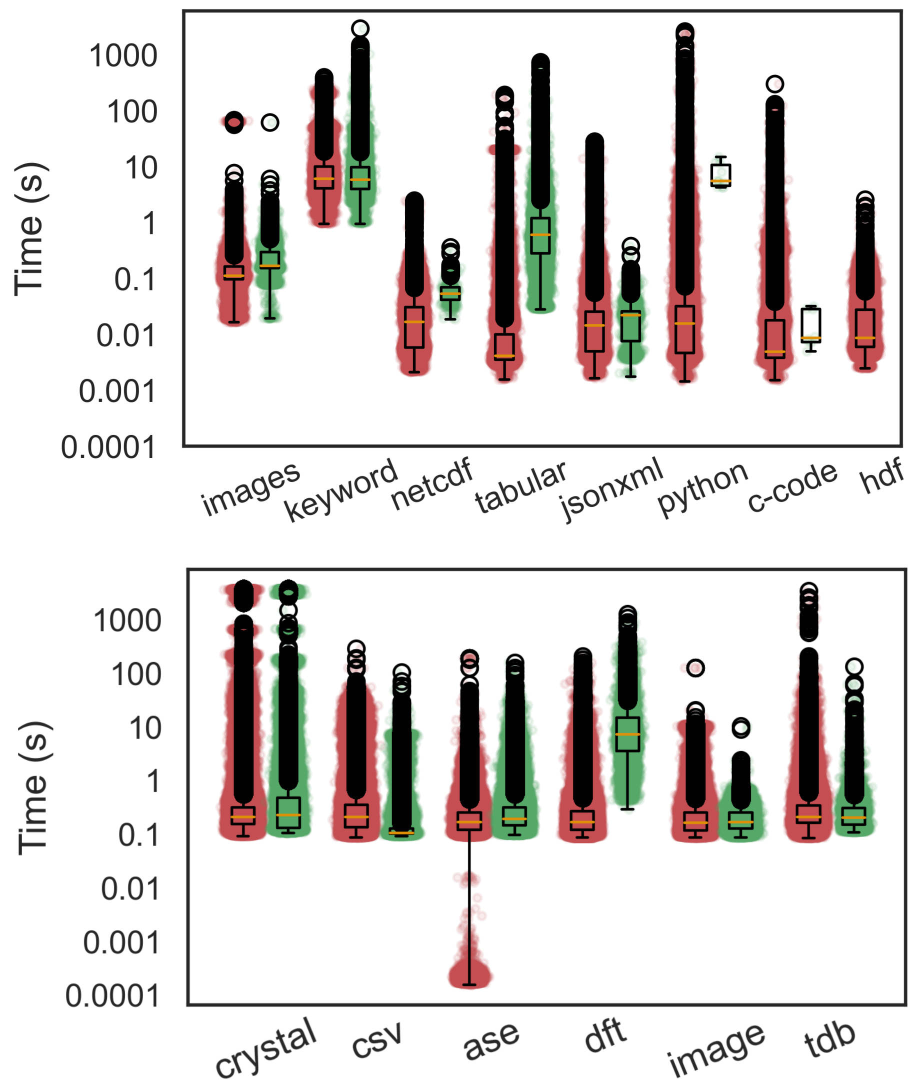


Figure 3.6: Extractor execution time by extractor type: CDIAC (top) and MDF (bottom). The red box plot illustrates the time taken to invoke that extractor on a file yielding *negligible* metadata; the green box plot is invocation time over files successfully yielding *non-negligible* metadata. The red and green points represent the processing time of individual files.

- **P4. Flexible:** metadata extractors should parse a *type* of data rather than a specific schema. For example, a `keyword` extractor should try to parse words from as many well-known formats as possible (e.g., *.pdf* vs. *.doc* vs *.txt*), rather than including separate extractors for each possible file extension. I use one of two design approaches for handling multiple schema in one extractor: (1) code adaptation, and (2) schema conversion. To explain both, I first provide a figurative example:

  > *A man is trying to eat a very large hamburger that is too big for his mouth. At first the man, wanting to enjoy the burger as quickly as possible, turns his head sideways, then upside-down, in order to hopefully* ***adapt*** *his mouth for the burger. He eventually gives up, and* ***converts*** *the burger into smaller pieces that he is more comfortable processing, even if cutting the burger takes time.*

  If the man is a metadata extractor and the burger is a file, the former example of trying to adapt his head to the burger without sullying the burger is *code adaptation*—it avoids changing the input. However, this is not always possible, so sometimes the input requires conversion to a common, more-easily-processable format. This is an example of *schema conversion.* Code adaptation occurs when instructions for processing a file can decipher differences in schema. An example of this occurs in a tabular file with a free-text preamble; in this case I first conduct a binary search over a file to find the first and last lines (the range) of the preamble before processing everything outside of that area as tabular data. Conversely, schema conversion occurs when each file is converted to a 'common' schema and then processed uniformly. This is generally helpful when the extractor uses well-known conversion libraries (e.g., Python libraries for loading *.pdf*, *.doc*, or *.md* as an in-memory ASCII-string), and adapting the code to handle schema differences is infeasible.

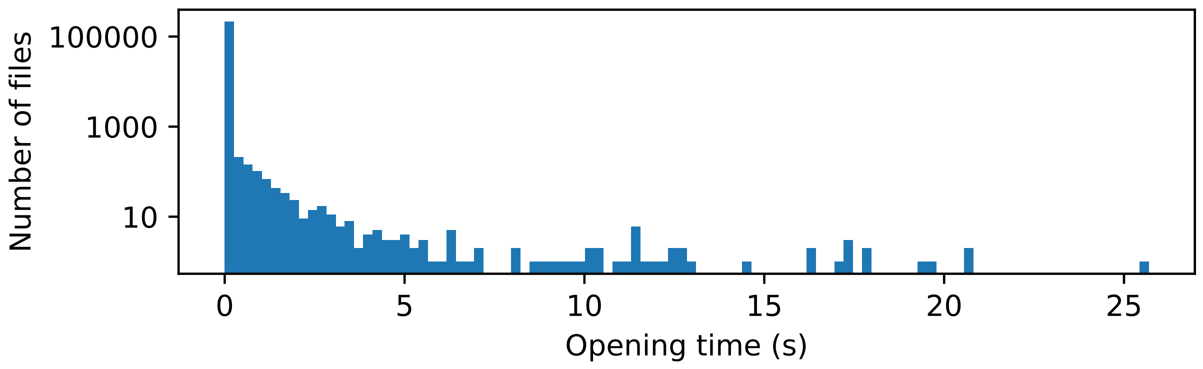


Figure 3.7: Histogram representation of time required to ‘open’, or load into memory, each file in the CDIAC repository. Observe that there is a long-tail of opening times that can adversely affect end-to-end extraction time.

- **P5. Modular:** each extractor should extract one type of data, and files containing multiple types of data should be processed by multiple extractors. For instance, consider the multi-type file with a free-text preamble followed by tabular data shown previously in Figure 3.2. Rather than having one *hybrid-freetext-then-tabular* extractor, there are instead separate `freetext` and `tabular` extractors that can individually isolate relevant data and return necessary, respective metadata elements.

In creating a list of principles, I opted to exclude some that might otherwise seem reasonable. For instance, at no point do these principles require that extractors be stateful, despite the obvious benefits of checkpointing progress for processing by another extractor. For instance if I process a PDF file and discover that it contains images, it would be useful to save the image tags so that the next extractor can more-quickly access these embedded image objects. However, this model is inherently heavyweight (requires reading/writing to/from disk multiple times *and* a communication mechanism for alerting the extraction system as to which extractor to send next), and makes it difficult to process files across machines subject to resource load (i.e., state eliminates the ability to freely move a file between endpoints). The principles also do not require that extractors produce metadata with consistent fields, as this would be difficult to harmonize between use cases (i.e., different communities have different metadata requirements and schemata).

It is important to note that it is up to the creators (i.e., developers) of extractors to uphold these principles; the community should determine the extent to which each extractor should abide by each principle. For instance, must a lightweight extractor terminate in 10 milliseconds, 10 seconds, or 10 minutes? Should an extractor mining for program information (e.g., Python or C) be flexible enough to also mine Java code? These decisions remain with the applicable communities and stakeholders therein.

3.3 From Principles to Product: Example Extractors

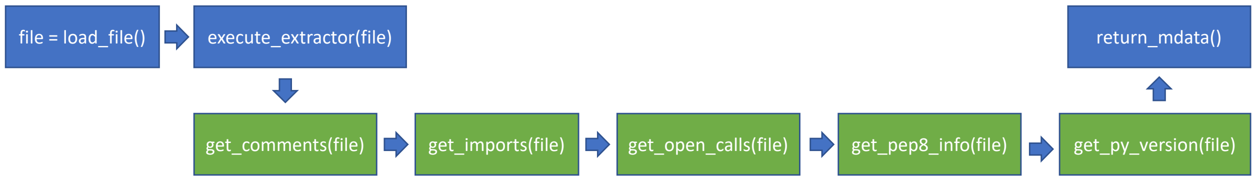
In this section, I illustrate how one can create extractors from the principles described in the previous section. As a running example, I outline the creation of two extractors— (1) the `python` extractor—which inputs a file and outputs searchable attributes of Python code files, and (2) the `keyword` extractor. One should note that all current extractors are written in Python and execute in Linux containers (via Docker, Singularity, and Shifter). By no means is this a requirement, as one could write extractors in other languages and virtual environments with marginally more effort.

3.3.1 ***Example 1:*** *Python Code Extractor*

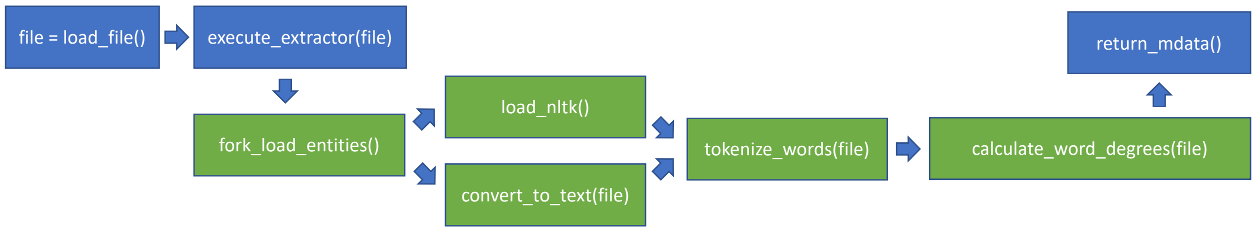
The `python` extractor inputs documents and extracts relevant, searchable metadata from them. Of interest are compatible Python versions, names of functions, free-text comments, required imports, and whether the script is likely to generate additional files. I developed this extractor in response to learning that there are 34 970 unindexed *.py* files located in Argonne National Laboratory's Petrel Petabyte-scale research data repository [1], and per community recommendations to process files directly linked to the scientific *process* and not just results. In the following, I discuss how I architected the extractor to adhere to the P1–P5 extractor engineering principles discussed in the previous section. I illustrate a high-level architecture diagram of the the `python` extractor in Figure 3.8a, with functionalities common to all extractors in blue boxes and those specific to the given extractor in green. All extractors must load a file, execute an extractor and return metadata, but this extractor is unique in fetching code-comments, imports, the number of calls to Python's *open* function, PEP8 style information, and compatible Python versions.

The `python` extractor exhibits each of the design principles from P1–P5, as follows. For guarantees of relevance (P1) and correctness (P2), I leverage a community consisting of three research software engineers to help assure the quality of the metadata outputs for the broader community. These domain experts outlined metadata attributes useful to consider when navigating their own and others' Python scripts, including imported libraries and the compatible Python version (by leveraging the `vermin` library [70]). Further, through proper software engineering channels (multi-engineer code review; unit testing), I have constructed a dynamically updated suite of unit tests on known Python files to validate extractor outputs. I set an aggressive latency limit (P3) of 20 seconds per extraction, because Python documents, by nature, are small (tens-to-thousands of short lines) and largely predictable (well-defined line-by-line schema of the scripting language). One strategy used to ensure low average extraction latency is to quickly reject files that are not of a valid Python format. I use multiple heuristic measures, such as no 'import' statements in the first 1000 lines of a file.[5] Flexibility (P4) is achieved by leveraging code adaptation approaches—I use regex to discern the different ways libraries can be imported (`import csv` versus `from csv import reader, writer`) or executed (`print x` in Python2 versus `print(x)` in Python3). Finally, modularity (P5) is enabled by only processing the Pythonic elements of the data (e.g., the extractor does not tokenize keywords in code comments; the heavier-weight `keyword` extractor isolates and processes those elements). In future work I will explore additional attributes to be included from the Abstract Syntax Tree (AST) library [39].

5. This is despite the uncommon, yet obvious, edge case here where one has a Python script without a single import statement.



(a) Python Extractor Diagram



(b) Keyword Extractor Diagram

Figure 3.8: Workflow diagrams for `python` and `keyword` extractors. Functionalities present in all extractors (as part of the extractor creation library) are represented by blue boxes; extractor-specific functionalities in green.

3.3.2 ***Example 2:*** *Keyword Extractor*

As I used the `keyword` extractor as a frequent example in §3.2, I will discuss only the differences of this extractor as compared to the `python` extractor. In prior work [115] I formally gauged relevance (P1) and correctness (P2) by tasking a panel of three graduate students with rating the keywords from a randomly selected subset of the files containing valid keyword metadata. I digitally provided each with 250 files and accompanying metadata documents, and tasked them with the the binary question, are the keywords in the metadata descriptive of the metadata? The extractor, according to the panel, exhibited 94% accuracy. The `keyword` extractor is lightweight (P3) due to its three-minute timeout on full-file processing and its concurrent dependency loading. Further, in my experience, the keyword extractor can fail quite slowly—the mechanism by which the `nltk` library (the natural language library that tokenizes bytes into words and sentences) processes files is vulnerable to taking a long time on binary executable files, as it furiously attempts to tokenize binary 'gibberish' into written English. To this end, I provide a heuristic based on file extension to terminate processing after 3 minutes, unless the file's extension is *README*, *txt*, *pdf*, or *doc(x)*. Further, the keyword extractor leverages parallel processing to load the `nltk` library as well as the file, as shown in Figure 3.8b, as each can take on the order of seconds. Finally, as previously discussed, the `keyword` extractor uses schema conversion techniques (P4) to convert text to a string (from *pdf* and *doc(x)* formats) before processing commences, and can isolate and tokenize keywords among punctuation and whitespace (P5).

3.4 Related Work

Related work in extractor design and creation spans two primary areas not already discussed in this chapter: other systems' extractor libraries and schema inference. The former is discussed as part of the greater related work on extraction systems in Chapter 2; thus in this section, I focus on work related to programs inferring files' schema such that they can be uniformly processed among other files of similar schema.

Schema inference is the process of discovering the structure of arbitrary data, including fields, types, null values, and delimiters. Current methods focus on well-structured data with greater uniformity than many general scientific data sources. Schema inference is an imperative initial step of metadata extraction; each extractor in the metadata extractor library needs to identify the correct parsing strategy for a given file in order to accurately discover or calculate metadata attributes. Otherwise, valuable metadata may be overlooked. In the following, I outline multiple such tools.

Tools such as RecordBreaker [21], PADS [34, 35], and Catamaran [44] represent solutions to the problem of identifying columns. Their models, while simplistic, are surprisingly accurate. The basic approach uses a set of possible delimiters and attempts to parse file rows using this set. The tools calculate the number of columns per row and overall variance across many rows. If the variance is below a predefined threshold, the column proposition is accepted. For each column, the tools then analyze individual values to determine the column's type. A histogram of values is created. If the values are homogeneous, a basic type can be easily inferred from the histogram. If the values are heterogeneous, then the column is broken into a structure, and the individual tokens with the structure are analyzed in a recursive process. I have replicated many such methods in the `tabular`, `python`, and `keyword` extractors.

Hybrid files add complexity to schema extraction, since data in scientific repositories are often concatenations of various types. For instance, a free-text preamble followed by a row of header labels and data will be miscategorized using the aforementioned models. There is prior work in using multi-hypothesis delimiter separated file parsing to extract a schema from untidy data when nulls and table structures are unknown [28]. *Predictive user interaction* affords a system high-accuracy in identifying multiple schemata, but does not allow full automation [53]. Google's Web Tables [10] harnesses advanced table interpretation at scale, but assumes a great deal of table structure, including the existence of a header row.

Some have even gone as far as to reverse-engineer input formats with a rich information set, including the record values, types, and constraints on the input [22]. Their approach (1) uses generative code (e.g., scripts that generate data) to reverse engineer field sequences, (2) applies clustering to records into a small set of types based on the steps used to process the record, and (3) infers constraints by tracking symbolic predicates from dynamic analysis of data flow.

3.5 Summary

In this chapter I outlined a vital tool for extracting rich metadata information from files of heterogeneous types and schemata: metadata extractors. I discussed how files can be of one or more types, and users' collective ability to build extractors capable of processing the types present in a repository enable the extraction system to maximize the coverage—the percentage of files emitting non-negligible metadata—of the extractor library onto each repository.

Additionally, I outlined the importance of the scientific community in crafting extractors; from determining the relevant, granular elements to be extracted from files; to evaluating the correctness of extractor outputs. Beyond relevance and correctness, I outlined principles P1–P5 for architecting metadata extractors, ensuring that extractors are lightweight, modular, and flexible. To put these principles into context, I presented a 'behind-the-curtain' look at two extractors: `python` and `keyword`.

Future work in extractor design encompasses two main areas: automation and community. Having a test suite that automatically computes an extractor's coverage, latency, and flexibility against a robustly-annotated corpus of data would alleviate concerns that extractor shortcomings not addressed in unit tests could only be discovered "in production". Future work focuses around better leveraging the community, for instance, improving the ways by which the relevance and correctness of extracted metadata are measured by better automating the review process. An ideal scenario would have a broad range of scientists contributing to the conversation, even to the point of having an automated chat-bot system [11] that can flag "questionable" metadata for review by a user and interactively query the user for input.

In the next chapter, I discuss multiple mechanisms by which extractors can be applied to files by using simple multi-output statistical learning models exhibiting high F1 scores.

CHAPTER 4
INTELLIGENT APPLICATION OF EXTRACTORS TO FILES

Metadata extraction systems can promote repository navigability by automatically populating rich, searchable data catalogs. These systems [110, 79, 78, 27, 131] generally follow a common structure, as illustrated in Figure 4.1, in which the following steps are performed in order: (A) iterate over all files in a repository; (B) identify the *type(s)* of each file (e.g., `tabular` or `image`); (C) invoke one or more *extractors* (sometimes called *parsers*) on each file to obtain metadata; and (D) perform an action with the resulting metadata (e.g., populate a search catalog). However, different metadata extraction systems focus on different use cases, data types, and communities, and therefore apply different approaches at each stage.

This chapter focuses primarily on step (B): inferring a file's type(s)—an important, but relatively underresearched step in metadata extraction systems. Challenges present in type inference are especially prominent in scientific data, as the broad nature of scientific inquiry often leads researchers to store data in esoteric formats, without regard for schema or file extensions, and data are often encoded in multidimensional file formats that integrate various data types into single or multiple files.

The growing volume and velocity of scientific data leads researchers to closely consider the resources used when extracting metadata. Naively applying all extractors to each file is not only inefficient, but may also lead to incorrect or irrelevant metadata. Figure 3.6 in the last chapter illustrates execution times when exhaustively invoking a library of eight extractors on every file in the Carbon Dioxide Information Analysis Center (CDIAC) data set [32]. The figure shows that while most extractors fail quickly, significant compute time is wasted; I estimate that successful invocations consume 130 core hours, whereas applying incorrect extractors (e.g., a NetCDF extractor on a Python script) consumes 670 core hours while returning no valid metadata. When mapping files to extractors, even the most advanced extraction systems do little more than map a MIME type, extension, or byte regex to a

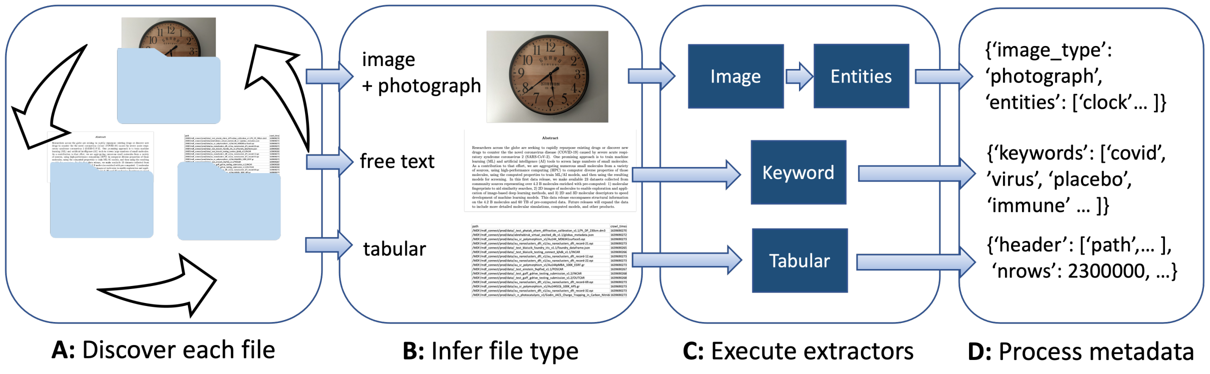


image
+ photograph
free text
tabular
Image
Entities
Keyword
Tabular
{'image_type': 'photograph', 'entities': ['clock'...]}
{'keywords': ['covid', 'virus', 'placebo', 'immune' ...]}
{'header': ['path',...], 'nrows': 2300000, ...}
A: Discover each file
B: Infer file type
C: Execute extractors
D: Process metadata

Figure 4.1: **Automated metadata extraction steps:** (A) find all files in repository, (B) infer each file's type such that it can be mapped to applicable extractors, (C) execute one or more extractors, (D) post-process metadata.

single extractor. However, when scientists create data in bespoke formats or store diverse data types within a single file, these modes of mapping extractors to files often fail.

In this chapter, I present an intelligent extractor scheduler for the Xtract metadata extraction system [110] that addresses many of the challenges in applying extractors to science data. I focus here on addressing file diversity by leveraging prior research in file type identification (FTI). I construct statistical learning models that, when used as part of a scheduler, can prioritize the application of extractors to collections of files; thereby maximizing some quality metric of the obtained metadata information. Furthermore, I evaluate the efficacy of these methods via a set of automatically derived metadata quality metrics. The contributions of this chapter are:

- Parameterization and evaluation of new multilabel, multi-output FTI methods on science data.
- Comparative evaluation that shows that this work's best FTI models outperform a state-of-the-art tool (libmagic [41]) in mapping extractors to files by 35% on CDIAC.
- Application of FTI methods on three large, uniquely diverse scientific data repositories: the heterogeneous CDIAC, the homogeneous COVID-19 Open Research Dataset

(CORD) [132], and the semi-curated Materials Data Facility (MDF) [8].

- The application and evaluation of multiple scheduling strategies applied to the CDIAC data, both in simulation and on a supercomputer.
- An automated metadata quality analysis toolkit capable of evaluating extracted metadata, regardless of their schema.

The remainder of this chapter is as follows. §4.1 presents related work in FTI and metadata quality. §4.2 outlines the automated metadata quality metrics explored in this thesis. §4.3 presents the algorithms, learning models, and quality metrics to be evaluated. §4.4 contains the evaluation of this work on three uniquely diverse scientific data repositories. Finally, §4.5 summarizes lessons learned.

4.1 Related Work

In this section, I review related work in metadata extraction systems, file type identification, and metadata quality.

4.1.1 Metadata Extraction Systems

When evaluating the breadth of open source metadata extraction systems (as illustrated in Table 1.1), I observe recurring research gaps: most systems do not cater to the scale and decentralized nature of modern scientific data; none consider the quality of returned metadata; and most have rigid schema constraints (i.e., only process a handful of file types) or manually map file MIME types to extractors, and therefore cannot support files of multiple types (e.g., a tabular CSV file with a free-text header). To the best of my knowledge, no prior system prioritizes extractors based on the expected value of metadata. This chapter strictly focuses on designing an FTI-based extractor scheduler for Xtract.

4.1.2 File Type Identification

File type identification (FTI) aims to automatically classify files by using their inscribed physical contents and is commonly used in digital forensics [92] and malware detection [121]. FTI methods traditionally rely on easily attainable features from the file (bytes, extension, size). However, science data creates unique challenges as file creators do not adhere to common file extensions, MIME types, or schema [115].

Leading FTI literature focuses on a single-label, single-output problem, with the implicit assumption that each file is of one type. In each given application context, this problem formulation is natural; in malware detection, for instance, it is sufficient to flag a file as one type of malware. Thus, researchers have explored common methods such as byte frequency profiles [80], centroid detection [75], support vector machines [48], logistic regression, random forests [65], and kNN [17]. Generally, the "success" of these models is measured via precision, recall, and F1 scores. In the context of metadata extraction, however, a file may map to more than one extractor—for instance, in the case of multi-typed files or files that otherwise have a natural chain of extractors applied (e.g., a photograph of a dog having the `image`, then `photograph` extractors applied). To this end, multilabel, multi-output models can naturally provide such classification flexibility, but have unfortunately been marginally explored in the context of FTI. In this work, I benchmark existing methods against tree-based ensembles: decision trees [86], random forests [57], extra tree, and extra tree*s* [107].

Without explicitly using statistical models, libmagic [41], the off-the-shelf tool used in Linux and Tika as a fast FTI utility, has *keep going* capabilities where it attempts to identify as many types as possible from a file. In this work, I benchmark libmagic's prediction performance against leading FTI methods for the extractor classification problem.

4.1.3 Metadata Quality

In quantifying the quality of metadata, the FAIR principles [136] of data management—which broadly state that research data should be **F**indable, **A**ccessible, **I**nteroperable, and **R**eusable—recently gave way to FAIR metrics [137, 50], a list of specific dimensions upon which one can validate data FAIRness. Table 4.2 contains the 14 FAIR metrics (as summarized by Kiraly [68]). These metrics encompass two broader categories: metrics of (i) administration and (ii) metadata. Metrics of *administration* score the quality of certain data policies and metadata practices used by administrators or curators of a data repository (e.g., digital resource librarians or system administrators). For instance, identifier persistence (F1.2) is obtained by delineating a clear plan as to what should happen when an identifier-type is deprecated. Access control (A1) requires administrators to design and enforce a clear set of rules regarding which data can be accessed and by whom. Depending on the context, questions surrounding provenance (R2) and community standards (R3) also generally enforce certain administrative procedures. Conversely, FAIR metrics regarding *metadata* score the constitution of a metadata search index. In practice, such metrics require that metadata contain certain attributes (F1.1, F2, F3, F4); enforce how metadata change over time, especially in the case of file updates and deletions (A2); and describe whether the metadata themselves are both generalizable and FAIR (I1, I2). These metadata metrics could be automated and used in a scalable metadata extraction workflow, especially when manual annotation is not feasible.

4.2 Metadata Quality Determination

Ultimately, the goal of metadata extraction systems is to derive useful metadata; however, current extraction systems do not consider the utility of extracted metadata for either individual files or entire data collections. Metadata quality metrics are thus necessary to illuminate the value of applying a given extractor to a file, and by extension, enables users

Label	FAIR metric	Description
F1.1	Identifier Uniqueness	Can I uniquely identify resource?
F1.2	Identifier Persistence	What happens when identifier is deprecated?
F2	Readability of Data	Are the data parse-able by humans and machines?
F3	Resource Identifier in Metadata	Does metadata contain identifier?
F4	Indexed in a Searchable Manner	Can the resource be found via search?
A1.1	Access Protocol	Are there limitations for using data?
A1.2	Access Authorization	Is there protocol for accessing restricted content?
A2	Metadata Longevity	Do metadata persist after data amended/removed?
I1	Knowledge Representation Language	Is there formal, shared, and broadly applicable KRL?
I2	FAIR vocabularies	Are metadata values and relations also FAIR?
I3	Qualified References	Are intra- and inter-metadata relationships clear?
R1	Accessible Usage Licenses	Is there a license document for data and metadata?
R2	Detailed Provenance	Is there insightful data history information?
R3	Community Standards	Are data certified by external entity?

Figure 4.2: Table of FAIRmetrics.

to evaluate the efficacy of FTI methods and extraction systems. While there is some prior work in metadata quality metrics [68], I specifically seek out metrics to *automatically* quantify the utility of a metadata corpus. I identify the following metrics that measure various dimensions of utility: yield, completeness, entropy, and readability.

Yield. Metadata yield is the total amount of metadata, measured as the number of bytes of metadata produced. While a simple measure, yield is useful for understanding the context of the other metrics, and is easy to obtain. For instance, how do 5 "readable" bytes compare to 1000 that are less readable?

Completeness. A primary criticism of the FAIR metrics is that they do not explicitly score metadata completeness. Metadata are complete if they contain all possible attributes that could be obtained. In practice, and especially in the presence of diverse schemata, some metadata attributes may be left empty. The simplest completeness metric [89] simply divides the number of metadata elements by the total number of elements that could be obtained (i.e., a percentage). I call this metric *simple_completeness* and define it in Equation (4.1), where N is the number of possible attributes and $P(i)$ is 0 if the i^{th} metadata attribute is null, and 1 otherwise:

$$simple\_completeness = \sum_{i=1}^{N} \frac{P(i)}{N} * 100 \tag{4.1}$$

A benefit of this completeness metric is that it is simple to calculate, but difficult to generalize across heterogeneous sources with hierarchical metadata models. For instance, a metadata model for images might naturally have completely disparate metadata elements for images that are photographs (e.g., entities, camera-type, or location) versus those that are scientific plots (e.g., statistical distributions, axis-labels, plot type). To avoid this heterogeneity issue, others [60] have developed scoring systems that similarly compute an 'absence' penalty for missing metadata elements, but provide weights to each metadata element, such that there are higher weights for relevant or mandatory metadata elements. I illustrate *weighted_completeness* in Equation 4.2, where a_i represents the weight of metadata field i:

$$weighted\_completeness = \sum_{i=1}^{N} \frac{a_i * P(i)}{\sum_{i=1}^{N} a_i} \tag{4.2}$$

Beyond this, others have extended completeness metrics to reward multi-value fields (e.g., lists) containing more information [77] and ranking each tier of hierarchical data structures [60].

I use *simple_completeness* as a sufficient and automatable proxy-measure in this thesis.

Entropy. Metadata entropy [106] is the degree to which metadata presents information that is different from other metadata. A common approach is to apply Term Frequency-Inverse Document Frequency (TF-IDF) to determine the entropy of a metadata document. TF-IDF for a metadata document provides an importance score for all words therein, relative to all documents in the corpus. Some [89] have proposed a score built on TF-IDF that produces an entropy score for a metadata document as is shown in Equation (4.3), where N is the number of text attributes, $attribute_i$ the i^{th} attribute of metadata, and $sum\_tf(attribute_i)$ the sum of TF-IDF scores for a given attribute (within a document):

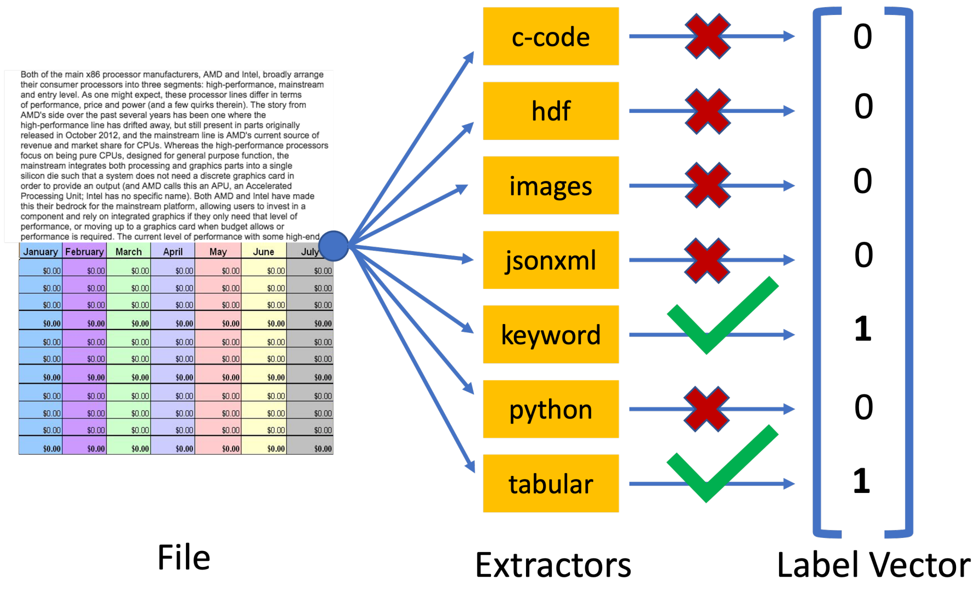
$$entropy = \log(\sum_{i=1}^{N} sum\_tf(attribute\_i)) \tag{4.3}$$

Readability. Readability measures the ability of humans to semantically interpret metadata. In this work, I leverage the Flesch Index [122]—a document score that compounds the complexity of words and sentences onto a scale where documents scoring below 0 are unintelligible to most human readers and those scoring above 100 are broadly understood. For metadata documents in a search index, I ideally give higher semantic weight to metadata containing searchable words, thereby penalizing number-dominated metadata. To accomplish this, I weight the Flesch index by the proportion of characters (n_char) that are not numbers (n_num): $W_s = (1 - \frac{n\_num}{n\_char})$. To account for decimal points potentially misrepresenting the ends of sentences in numeric metadata, I remove all mid-numeric decimal points prior to tokenizing. I then define a weighted Flesch index $WFlesch$, where n_word, n_sent, n_syl are the number of words, sentences, and syllables, as follows:

$$WFlesch = \overbrace{(206.835 - 1.015(\frac{n\_word}{n\_sent}) - 84.6(\frac{n\_syl}{n\_word}))}^{\text{original Flesch Index}} * W_s \tag{4.4}$$

4.3 Methodology

I now describe the process of using statistical learning models to identify applicable extractors for each file in a science repository. Specifically, I describe label and feature generation, model selection, and the steps to leverage model outputs as input to the extraction scheduler. In this section, extractor selection is envisioned as a mlti-label, multi-output learning problem.


Both of the main x86 processor manufacturers, AMD and Intel, broadly arrange their consumer processors into three segments: high-performance, mainstream and entry level. As one might expect, these processor lines differ in terms of performance, price and power (and a few quirks therein). The story from AMD's side over the past several years has been one where the high-performance line has drifted away, but still present in parts originally released in October 2012, and the mainstream line is AMD's current source of revenue and market share for CPUs. Whereas the high-performance processors focus on being pure CPUs, designed for general purpose function, the mainstream integrates both processing and graphics parts into a single silicon die such that a system does not need a discrete graphics card in order to provide an output (and AMD calls this an APU, an Accelerated Processing Unit; Intel has no specific name). Both AMD and Intel have made this their bedrock for the mainstream platform, allowing users to invest in a component and rely on integrated graphics if they only need that level of performance, or moving up to a graphics card when budget allows or performance is required. The current level of performance with some high-end

January	February	March	April	May	June	July
$0.00	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
$0.00	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
$0.00	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
$0.00	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
$0.00	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
$0.00	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
$0.00	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
$0.00	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
$0.00	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
$0.00	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
$0.00	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00

Figure 4.3: Visual representation of the automated muiltilabel generation process. Each extractor executes on every file in the training set; if an extractor returns valid metadata for a file, it is assigned a '**1**' in its label position.

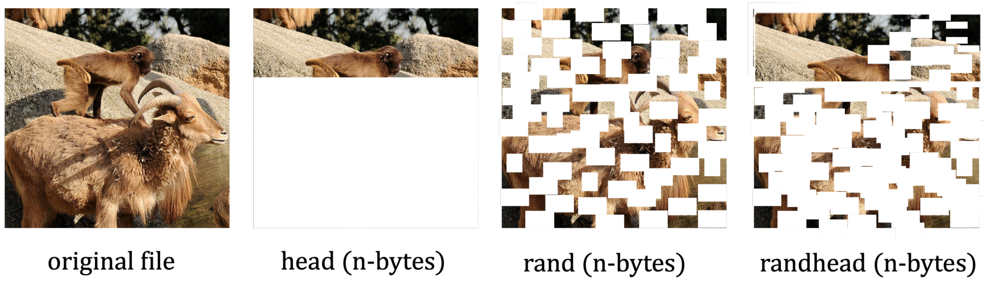
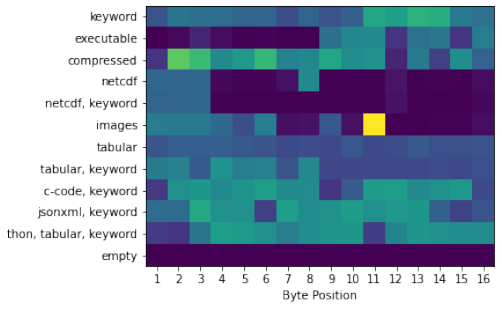
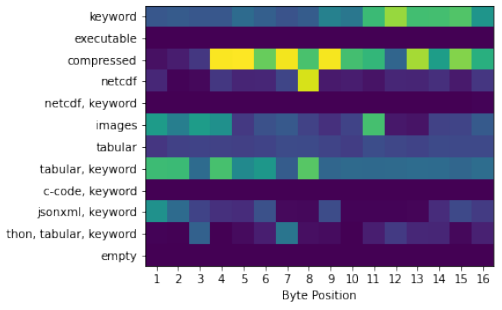


Figure 4.4: Feature illustration for head, rand, and randhead.

Feature selection. I create input feature vectors containing (i) file size and (ii) 16–512 byte samples from the file. As illustrated in Figure 4.4, byte samples are fetched from the following locations in the file: the header (head), randomly throughout (rand), or a combination of both (randhead). The implicit assumption in choosing to use bytes as features is that each file type has a unique byte profile. For visual purposes, I illustrate the mean and standard deviation of 16 head bytes from all files in CDIAC in Figure 4.5.



(a) Byte Value Mean



(b) Byte Value Standard Deviation

Figure 4.5: Visual representation of CDIAC 16 head byte features: (a) the mean and (b) standard deviation of the byte values at each position. **Darker**=smaller.

Model selection. I train models to accomplish the following: given a file $f \in F$, I want to train a model $m \in M$ such that $m(f)$ generates a probability distribution $P(f) = [p(f, e_1), p(f, e_2), ..., p(f, e_n)]$, where $p(f, e)$ is the probability that f should map to extractor $e \in E$. In this work, I attempt to find a model that outperforms the best model configuration of my own prior work [114]—a random forests model. In §4.4, I compare the performance of the random forests model and its tree-based alternatives: decision tree (dtc), extra tree classifier (t_etc), and extra tree*s* classifier (e_etc).

I evaluate the performance of these models using samples-weighted F1, precision, recall, and training time. I also examine specific muiltilabel metrics such as coverage error, LRAP, ranking loss, and normalized discounted cumulative gain (NCDG) [43, 25]. In my previous work, I accounted for potential overfitting by evaluating the models on both imbalanced (all data) and balanced (subset of the data) classes. In this work, I do not balance classes as tree-based ensemble models (rf, e_etc, and t_etc) are unlikely to succumb to severe overfitting given sufficient hyperparameter tuning [91, 38]. I also investigate individual class-label performance via multiple confusion matrices.

Extraction Scheduler. The primary goal of this work is to design an extraction scheduler that converts FTI model outputs into a queue of file/extractor pairs to execute. I explore scheduling strategies that optimize on two criteria: *expected yield* and *expected completeness*. For each, I first train lightweight regressions $R(e, size(f))$ that use a file's size and assigned extractor to predict metadata yield $Y(e, size(f))$ or completeness $C(e, size(f))$. I automatically select a regression based on which has the better correlation score between a linear and nonlinear model [133], and fit the corresponding model. Given the probability vector of file-extractor mappings, $P(f) = [p(f, e_1), p(f, e_2), ..., p(f, e_n)]$, the size of a file $size(f)$, the average extractor execution time t_e and a +1 Laplace smoothing constant, I introduce an objective function to compute predicted metadata yield or completeness over time $\alpha(f, e)$:

$$\alpha(f, e) = log(\frac{R(e, size(f)) * p(f, e) + 1}{t_e}) \tag{4.5}$$

For both schedules, I prioritize extractor execution by loading a priority queue in descending order of alpha score (i.e., the system maximizes the expected yield or completeness over time). I evaluate both schedules on the Carbon Dioxide Information Analysis Center (CDIAC) climate science data set by simulating the returned metadata over time, and use the best model in a time-constrained extraction job using 8 nodes of the Theta supercomputer at Argonne National Laboratory.

4.4 Evaluation

I first analyze the feature and model performance of the FTI methods and compare the prediction performance with libmagic [41]. I then use the predictions to construct, execute, and evaluate quality-based schedules—both via simulation and on the Theta supercomputer—of CDIAC.

Science repositories. I evaluate the models in the context of three distinct scientific repositories. I primarily focus on CDIAC, which represents a multi-group conglomeration

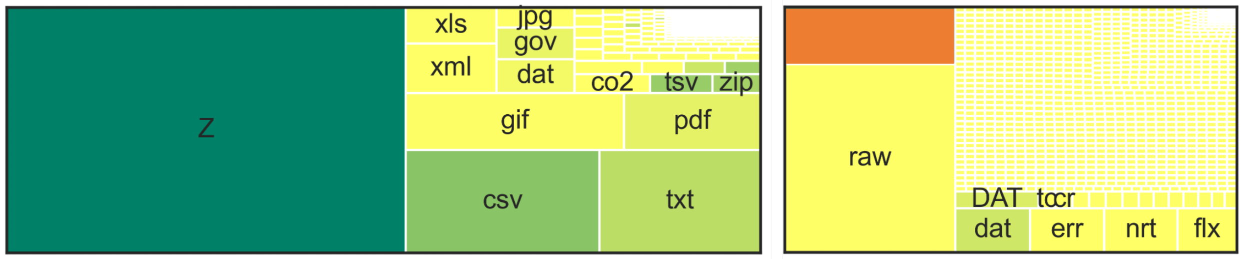


Figure 4.6: **Treemaps of CDIAC:** (left) the unedited repository, (right) all decompressed files. Each box's *area* is the proportion of files of that extension, and *darkness* is the relative total size (darker=bigger). The orange box on the right represents files with no extension.

of carbon dioxide data. I copied these data from their now-defunct FTP server in 2017. These data, whose extensions are visualized as a treemap in Figure 4.6, have a high degree of variety—there are over 150 unique file extensions spanning 428 000 files, and many of the files are in difficult-to-parse formats (e.g., deprecated Windows installers, Hadoop error logs, and desktop shortcuts) [115]. These files include both the unedited file formats consisting of many compressed files (e.g., *.Z*) and their decompressed contents. Second, I examine the more homogeneous COVID-19 Open Research Dataset (CORD) containing 517 000 JSON-formatted COVID-19 research papers spanning 2019–2021. Finally, I explore the Materials Data Facility (MDF), which contains 2.3-million semi-curated files from materials science. For scheduling purposes, I focus on the CDIAC data set as it is the most heterogeneous of the three.

Experimental Testbed. I perform all experiments on ALCF Theta, an 11.7-petaflop Cray XC40 supercomputer with second-generation Intel Xeon Phi "Knight's Landing" (KNL) processors. Each node has a 64-core processor and 166 GB MCDRAM, 192 GB DDR4 RAM, with a shared Lustre file system. For model training, I use a single compute node; for scheduling experiments, I use 8 nodes.

4.4.1 FTI Modeling

Features. I first search for the best byte structure (head, rand, randhead) and number of bytes (16–512) to use as features in this analysis. In prior work [115], I use a standard 70%/30% train/test split, and in this work, I use a 10%/90% train-test split. This disparity is due to the nature of the work at different times. In the prior work, I broadly evaluated whether FTI methods were feasible for extractor selection. In this work, however, I have started to construct models that could minimize the total time spent generating labels as part of an end-to-end extraction job. Fortunately, this work is not meant to provide an apples-to-apples comparison of linear and tree-based learning models—I simply aim to find the best-performing model for the extraction scheduler.

Table 4.1: Model performance for 16 and 512 bytes for logistic regression (logit), random forests (rf), and support vector classifier (svc) on CDIAC and CORD; 70%/30% train/test split.

Repository	**Header Bytes**	**Model**	**Train Time (s)**	**Precision**	**Recall**	**F1 Score**
CDIAC	16	logit	403	0.839	0.836	0.837
		rf	2.29	0.890	0.896	0.893
		svc	1010	0.856	0.867	0.861
	512	logit	1140	0.930	0.936	0.933
		rf	4.50	0.939	0.938	0.938
		svc	9240	0.875	0.885	0.880
CORD	16	logit	17.0	1.00	1.00	1.00
		rf	3.56	1.00	1.00	1.00
		svc	418	1.00	1.00	1.00
	512	logit	183	1.00	1.00	1.00
		rf	4.25	1.00	1.00	1.00
		svc	464	1.00	1.00	1.00

Figure 4.7 illustrates the range of model scores for the different byte structures across models borrowed from prior FTI work: logistic regression (logit), random forests (rf), and support vector classifier (svc). For the CDIAC data, the figure shows that head bytes out-

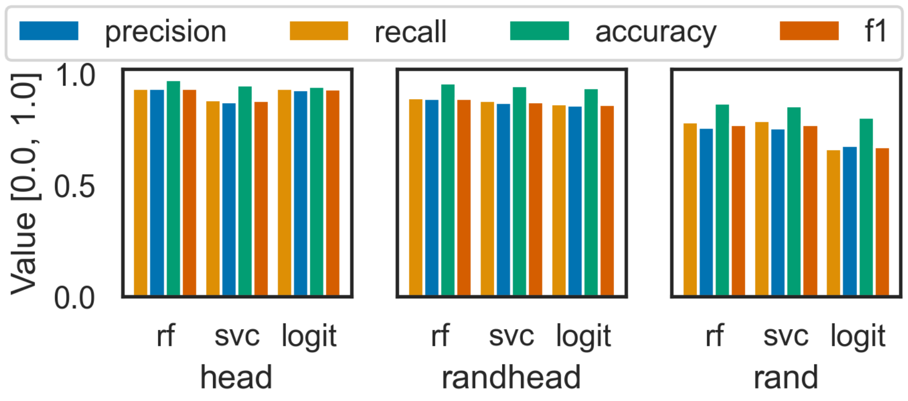


Figure 4.7: Model scores for multiple 512-byte feature configurations (CDIAC).

perform rand and randhead in every statistical metric. Next, I identify the optimal number of head bytes; as illustrated in Table 4.1, 512 head bytes outperforms 16 head bytes across all models when applied CDIAC. To investigate whether there is significant benefit beyond 512 head bytes, I compare F1 improvements when doubling from 16 to 32 and 256 to 512, respectively. The relative F1 difference when increasing from 16 to 32 bytes for (logit, rf, svc) is (+0.1, +0.06, +0.07), but the difference between 256 and 512 bytes is only (+0.02,+0.01,-0.02). Therefore, I use 512 head bytes, since additional bytes would likely have marginal benefit. Due to its homogeneity, CORD can be processed well by any of the feature configurations. Due to the obvious advantage of head bytes over either random configuration, I exclusively use 512 head bytes in the following analysis.

Models. I next look to robustly compare model performance to select a classifier for the CDIAC extractor scheduler. Using the models from my previous work, I find that the random forests model outperforms logit and svc across all presented feature types (see Figure 4.7) and sizes (Table 4.1). The imbalanced class (full-repository) experiments shown in Figure 4.8 show that the random forests model can adequately identify most file types in the CDIAC repository without evidence of severe overfitting when compared with the balanced classes.[1] The balanced CDIAC subset was manually constructed by randomly selecting 200 files from each single-label class in the training set (and omitting those classes with fewer than 200 files). The dark diagonals on the confusion matrices show that the model exhibits both high precision and recall in identifying individual file types, and the precision-recall curve, as expected, exhibits high precision, regardless of the recall level, for each label type. Interestingly, one can see in both diagrams that the most difficult class for the random forests model to identify is the "unknown" class.

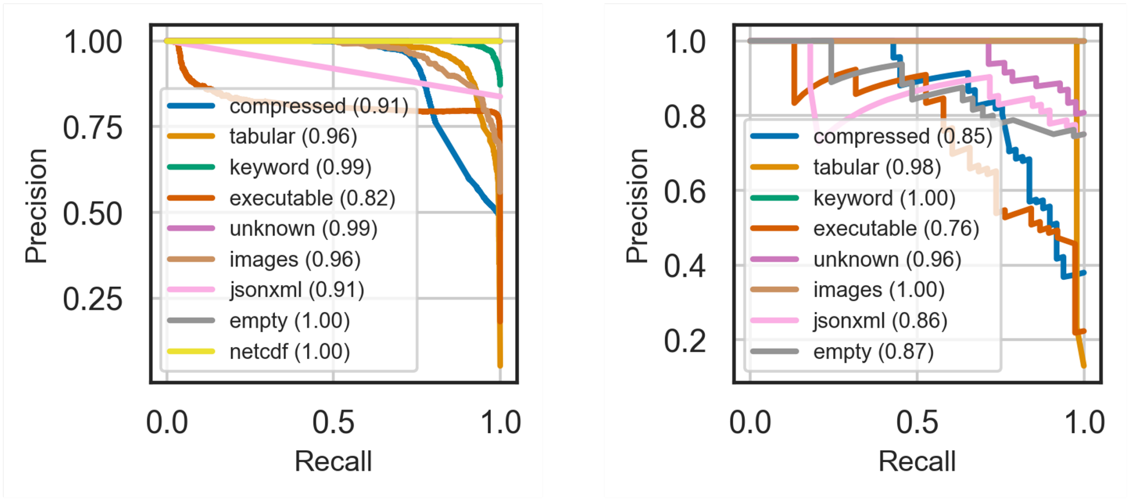
To study how the original random forests model performs when individual files contain multiple content types, I analyze the output probability distributions for all multi-typed files, and investigate whether each type is represented at the top of the file's label probability distribution. Table 4.2 shows that, for CDIAC, most multi-typed files share a type with the `keyword` extractor, and the model correctly identifies the `keyword` type 80% of the time and the other type 96% of the time.

Table 4.2: Analysis of files of both types Type 1 and Type 2, and how many of each type are included in the top-2 entries of the probability distribution.

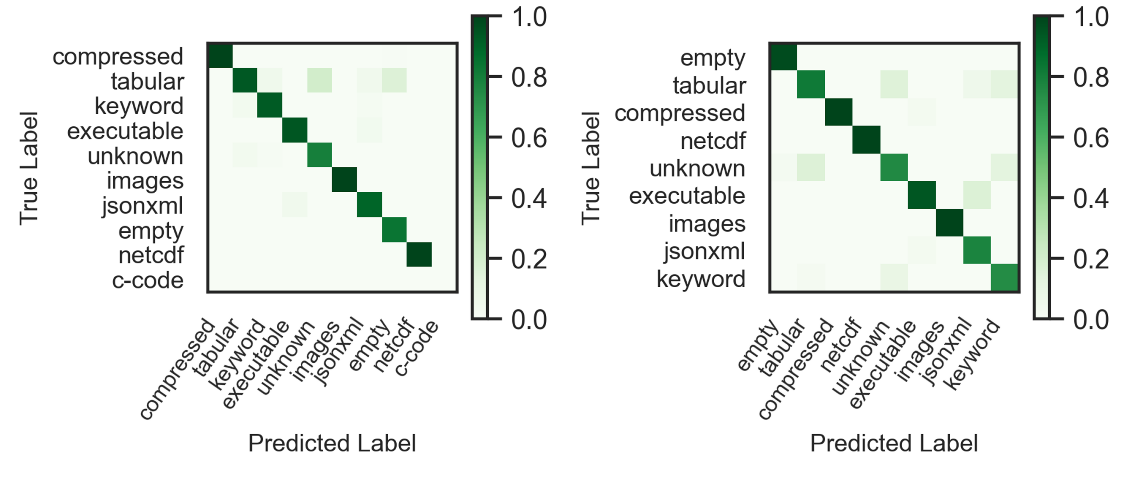
Repository	**Type 1**	**Type 2**	**Count**	**Type 1 Included**	**Type 2 Included**
CDIAC	keyword	tabular	12 878	10 966	12 415
		jsonxml	3282	1954	3109
		netcdf	252	205	252
		c-code	8	8	3
		python	3	3	3
	tabular	python	7	7	7
CORD	jsonxml	keyword	517 900	517 900	517 900

I next examine whether another nonlinear, tree-based model can exhibit even higher F1 scores in the extractor selection problem across three repositories: CDIAC, CORD, and

1. without loss of generality, I assume that the CORD repository, by its homogeneous nature, is already balanced.



(a) Class-weighted precision-recall curves (left imbalanced, right balanced)



(b) Prediction-normalized confusion matrices (left imbalanced, right balanced)

Figure 4.8: **Overfitting Analysis (CDIAC):** for random forests model trained on 512 head bytes. There is marginal change in overall model performance between imbalanced and balanced classes, which means that severe overfitting is unlikely.

MDF. Using the same 512-byte head features, I train the following models: decision tree (dtc), extra tree (e_etc), extra tree*s* (t_etc), and a new random forests model (rf2) with 30 estimators and a maximum tree depth of 4000. Prediction performance of each model on each repository is given in Table 4.3. For CDIAC, all models perform similarly well, with e_etc performing slightly better than the alternatives. Despite the smaller training set, e_etc exhibits an F1-score improvement over rf: from 0.938 to 0.975. The CORD model, similarly to before, performs perfectly due to the homogeneity of the repository. Finally, the best model for MDF, like for CDIAC, is also e_etc. Fortunately across all three repositories, the similar performance between dtc and its multi-tree counterparts demonstrates dtc's probable overfitting avoidance.

Table 4.3: Classifier performance metrics for multilabel models trained on 512 head bytes for each file in specified repository; 10%/90% train/test split.

Model	**Repo**	**Train Time (s)**	**Precision**	**Recall**	**F1**	**Coverage Error**	**LRAP**	**Ranking Loss**	**NDCG**
dtc	CDIAC	52	0.977	0.970	0.973	1.475	0.967	0.031	0.984
rf2	CDIAC	56	0.981	0.968	0.974	1.533	0.968	0.033	0.984
t_etc	CDIAC	55	0.977	0.970	0.973	1.485	0.967	0.032	0.984
e_etc	CDIAC	109	0.980	0.970	0.975	1.491	0.969	0.031	0.985
dtc	CORD	129	1.000	1.000	1.000	2.000	1.000	0.000	1.000
rf2	CORD	125	1.000	1.000	1.000	2.000	1.000	0.000	1.000
t_etc	CORD	136	1.000	1.000	1.000	2.000	1.000	0.000	1.000
e_etc	CORD	132	1.000	1.000	1.000	2.000	1.000	0.000	1.000
dtc	MDF	189	0.980	0.995	0.988	1.14	0.998	0.001	0.999
rf2	MDF	206	0.980	0.996	0.989	1.16	0.998	0.001	0.999
t_etc	MDF	187	0.986	0.997	0.991	1.14	0.998	0.001	0.999
e_etc	MDF	316	0.994	0.998	0.996	1.14	0.999	0.001	0.999

I next evaluate how the best-performing model, e_etc, performs on labeling each file. To first measure how well the model predicts each individual label type (i.e., splitting up multilabel sets), I create a confusion matrix for each label as illustrated in Figure 4.9. In this case, the model adequately labels most file types besides the underrepresented `c-code` and `python`. An even stronger measure of the model is its efficacy in selecting exact multilabel sets for each file. In Figure 4.10, one should observe that the model does struggle on some multilabel sets (e.g., `tabular,keyword` and `jsonxml`), but performs well in general.

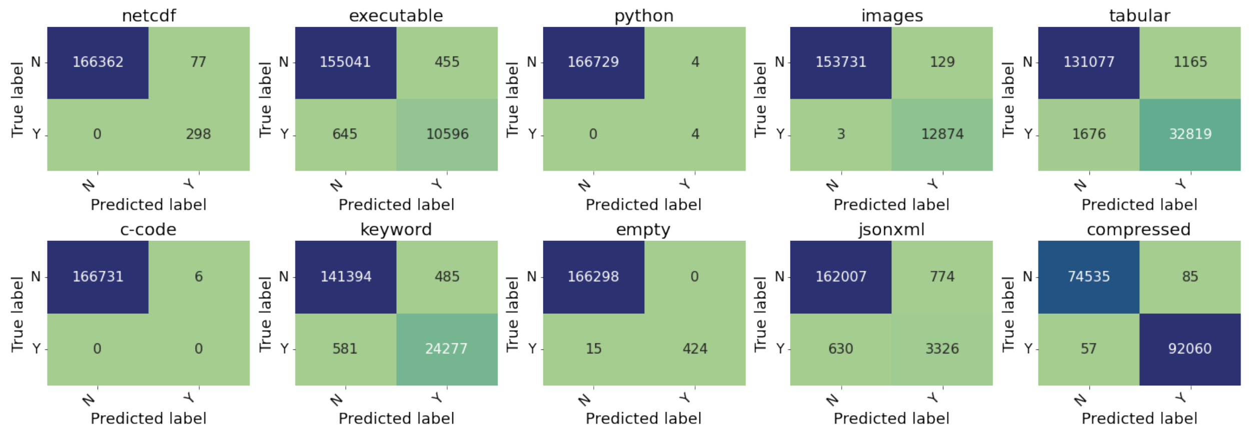


Figure 4.9: **Multi-Confusion Matrices (single-label)**: e_etc performance on each label.

Despite the strong performance on predicting multilabel sets, I am left with the following question: given the presence of multi-typed files whose contents are spatially separated throughout the file (e.g., tabular files with free-text headers), why can head bytes adequately predict these files? After exploring many of these file types, the answer became clear: these files, despite containing multiple types, have commonly structured free-text preambles, thereby making them uniformly parsable.

Libmagic Comparison. I next compare both the original random forests (rf) and the new extra trees classifier (e_etc) models with the libmagic FTI tool using the balanced CDIAC data set. As libmagic types do not directly map to Xtract's extractor library, I manually map libmagic outputs to my own label set. Some mappings are obvious (e.g., empty:empty, compress'd:compressed) while others require consulting libmagic documentation (e.g., data:unknown). Figure 4.11 shows the result of comparing each libmagic output to the ground truth extractor labels. Overall, libmagic performs significantly worse than the new FTI methods, as it consistently misclassifies tabular and keyword data. Even in this favorable experiment, libmagic only accurately identifies 65–72% of files (cf. rf and e_etc correctly identify 86-88% and 91–93%, respectively). The ranges are the observed accuracy values when both including and excluding executable files, as having ground-truth labels for

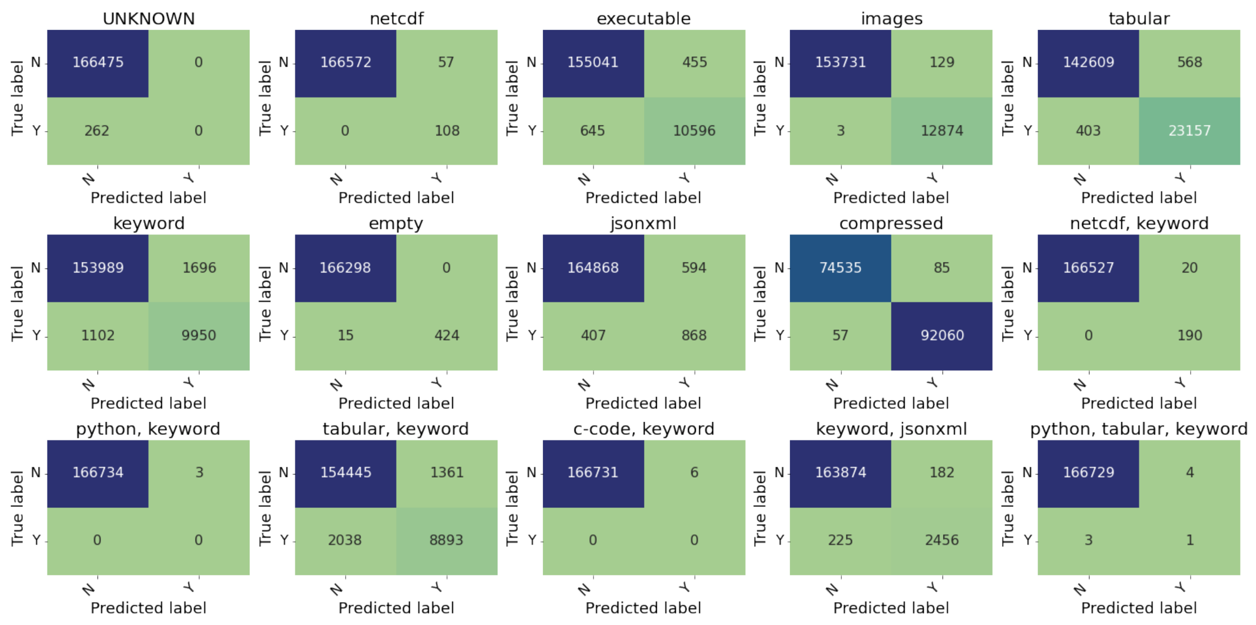
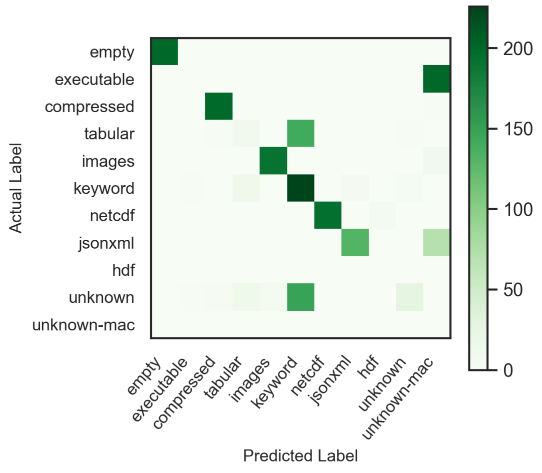


Figure 4.10: **Multi-Confusion Matrices (multilabel)**: e_etc performance on all observed label combinations.

these would require executing each file individually. In other contexts, however, libmagic has empirically shown to be 98% correct [7], so this likely warrants future study on additional file sets.

4.4.2 Extractor Scheduler and Metadata Analysis

The relative utility of metadata extraction is measured by counting "interesting" metadata documents as defined by the quality metrics: `semantic` metadata are those that contain searchable words (measured as files with positive readability scores), `near-full` metadata contain complete data (measured as files with over 80% completeness), `high entropy` metadata add unique information to the corpus (measured as the top-20% of semantic files, in terms of entropy score), and `high yield` metadata exceed 500 bytes. These metrics are used to measure the efficacy of the scheduling algorithms in the following. For CDIAC, the interesting metadata documents produced by each extractor are shown in Table 4.4; the



(a) Confusion Matrix

Type	Pr.	Re.	F1
empty	1.00	1.00	1.00
executable	0.00	0.00	0.00
compressed	0.98	1.00	0.99
tabular	0.25	0.06	0.10
images	0.96	0.96	0.96
keyword	0.44	0.91	0.59
netcdf	1.00	0.97	0.98
jsonxml	0.96	0.65	0.78
unknown	0.84	0.13	0.23
unkn.-mac	0.00	0.00	0.00

(b) Precision, Recall, F1

Figure 4.11: **Libmagic (CDIAC):** confusion matrix and performance metrics for mapping extractors to files using the libmagic FTI tool.

`keyword` and `tabular` extractors dominate the quality file counts. This is sensible for two reasons: (i) files in CDIAC, as shown back in Figure 5.8, tend to successfully generate metadata when having the `tabular` and `keyword` extractors applied, and (ii) the contents of the `tabular` extractor scale to the number of columns in the file (i.e., each column has a data type, mean, median, mode, etc.). For purposes of illustrating the observed quality profiles of extractors, I illustrate in Figure 4.12 the normalized log value of the median value as a percentage of the 75th percentile, which allows for visualizing the data without outliers. On these CDIAC data, it is apparent that different extractors generate unique metadata quality profiles: tabular metadata exhibit high readability and entropy, keyword metadata exhibit high readability and yield, and image metadata exhibit high completeness.

Scheduler #1: yield-over-time. Figure 4.13a shows the total invocations by extractor type on the left and the interesting files found over those invocations on the right. The

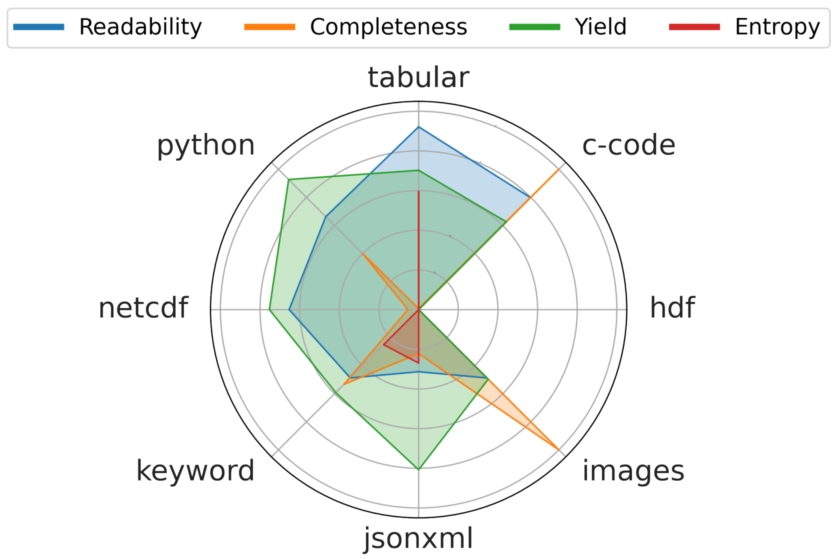


Figure 4.12: Spider plot representation of successful metadata extraction metrics on CDIAC. The distance between the center and each extractor is the log-scale range of each metric, and the placement of the colored line represents the median of the corresponding metric.

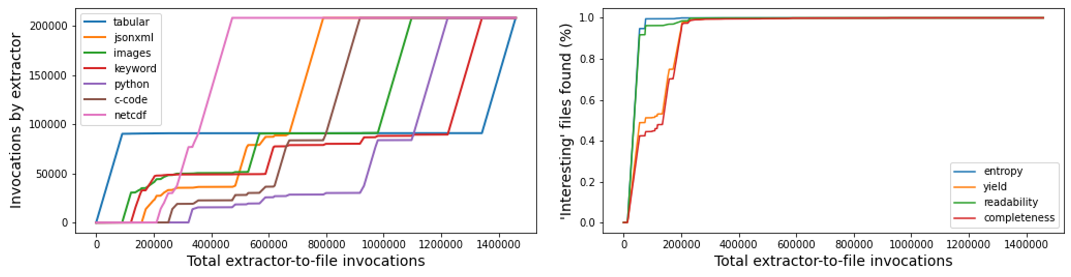
Table 4.4: Number of interesting files by extractor (CDIAC).

Extractor	**High Yield**	**High Entropy**	**Semantic**	**Near-Full**
c-code	6	0	6	6
images	13636	0	0	13636
jsonxml	2770	38	223	2770
keyword	29501	0	471	16515
netcdf	209	1	199	191
python	10	0	6	10
tabular	36311	12336	31770	17161
All	**70897**	**12377**	**33003**	**38616**

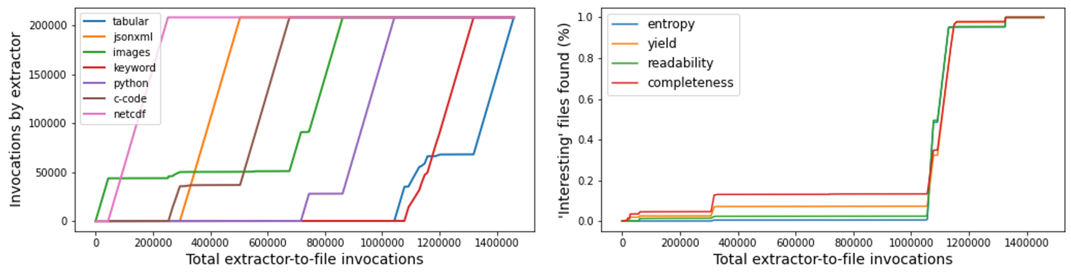
model performs well; a majority of the interesting files are found in the first 250 000 extractor invocations. Given that the metric of optimization is yield-over-time, the model naturally prioritizes extractors that generate high-yield metadata *or* those that execute quickly. Table 4.4 shows that a majority of high-yield files stem from the `tabular` and `keyword` extractors. The immediate jump in successive `tabular` invocations shows that the expected yield of `tabular` extractors is relatively high, and the subsequent jump in `image` invocations around total invocation 10 000 is due to the low latency of the `image` extractor. Given that this scheduler prioritizes `tabular` and `keyword` invocations, a large proportion of interesting metadata can be immediately generated. Therefore, this scheduler (when applied to these data) could add significant value for organizations looking to create a semantically searchable index with high information content with a limited compute budget. The stark vertical lines above $y = 100000$ in the extractor invocation graph are the result of invocations having "near-zero" probability of belonging to the corresponding extractor; alpha is determined solely by the average execution time in the denominator.

Scheduler #2: completeness-over-time. Figure 4.13b presents a less effective choice for CDIAC: expected metadata completeness-over-time. First, the expected completeness clearly has stark divisions between extractors that cause each extractor to be successively applied. Second, some extractors regularly exhibit 100% completeness (e.g., `images`), whereas others have varying levels of 'miniscule' completeness (e.g., `keyword`, `tabular`). This scheduler could provide value to an organization whose most complete extractors (e.g., `images`) yield the most interesting metadata (by their measures).

Finally, I implement the expected yield-over-time scheduler in Xtract and observe how it prioritizes the necessary extractions on 8 nodes of the Theta supercomputer with a strict 1-hour wall time. The real-system approach can behave differently than the simulation;



(a) **Good:** expected yield over time



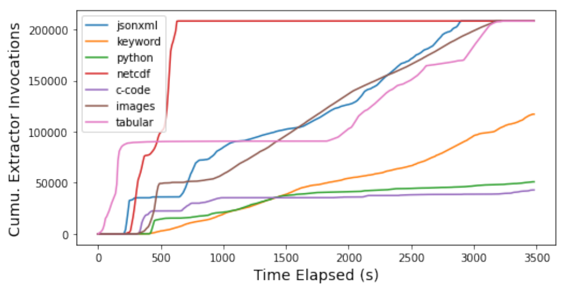
(b) **Bad:** expected completeness over time

Figure 4.13: **Simulated schedule performance over total invocations:** (a) yield-over-time schedule; (b) completeness-over-time schedule.

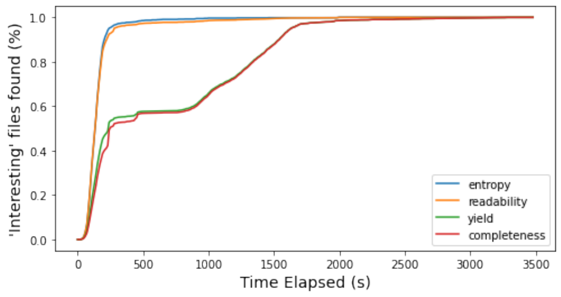
extractor execution times vary from file-to-file. The invocation graph and the interesting files found over 1 hour are shown in Figure 4.14. In this case, interesting files are still found relatively quickly (although not as fast as in the simulated version). This extraction job generates 97% of the interesting metadata documents by 2100 seconds, and 99% of interesting metadata documents by 2900 seconds. Therefore, a user leveraging this scheduler can nearly minimize their resource usage and still find nearly all interesting files.

4.5 Summary

Accurate and performant metadata extraction depends on accurate methods for mapping extractors to files; however, traditional methods are not conducive to the wide, heterogeneous variety of science file formats. I introduce several file type identification methods that use lightweight byte features from files and various machine learning models to predict



(a) Extractor invocations over time (s)



(b) Interesting files found over time (s)

Figure 4.14: **Schedule performance on supercomputer:** expected yield-over-time schedule with strict 1-hour time limit

scientific file types. These models are used to create an extraction scheduler for the Xtract metadata extraction system, enabling Xtract to prioritize application of extractors to files. Furthermore, I introduce several metrics designed to quantify the utility of metadata, and by extension, the usefulness of the extractor scheduler.

CHAPTER 5
AUTOMATION AND SCALE OF EXTRACTION WORKFLOWS

Metadata extraction systems, as discussed in Chapter 1, have not been shown to simultaneously suit the scale and decentralized nature of science data. Existing extraction systems tightly couple compute and data storage capabilities—data are currently assumed to be processed at the compute facility at which they are stored *or* moved to a separate compute facility designated for processing. However, neither approach is satisfactory in the general case: the former because computational capabilities at data repositories may be lacking or inadequate, and the latter because of the high costs of moving large quantities of data. A hybrid approach in which metadata extraction can be performed on either centralized or decentralized systems, depending on the context, can reduce costs.

Scientific data generation processes, and therefore metadata extraction workloads, are inherently bursty, and can benefit from decentralization to utilize available computing resources. Further, once an experiment is completed and the data are to be added to a repository, many terabytes of files can all require metadata extraction at once, necessitating the large scale application of extractors across potentially disparate resources. The Function-as-a-Service (FaaS) computing model is predicated on elastically scaling resources to accommodate bursty workloads, and federated FaaS enables seamless execution across distributed computing infrastructure spanning administrative domains. Therefore, I propose the following research question: *can FaaS infrastructure enable the creation of scalable, efficient, decentralized metadata extraction workflows for large, distributed scientific data?*

In this section, I describe and evaluate Xtract, a bulk metadata extraction system that orchestrates the extraction and synthesis of metadata by dispatching and executing lightweight and specialized metadata extractors on files in a target repository. Xtract is unique in that it completely decouples data locality from computation, enabling the deployment of performant metadata extraction workflows across a continuum of decentralized computing resources [111, 112]. This decoupling is made possible using funcX, a federated FaaS platform, to invoke metadata extractors on remote computers. Xtract abstracts data location and movement (when optimal), decisions as to which extractors to apply, and the orchestration of extractors across files on disparate computing resources.

This chapter extends my prior work [113, 115] by creating a system that leverages federated FaaS to construct scalable, efficient, decentralized metadata extraction workflows on scientific data. The contributions of this work are:

- Xtract, the first distributed metadata extraction system that leverages FaaS to scalably crawl and extract metadata from large, distributed collections of files.
- Performance evaluation of remote metadata extraction on research cyberinfrastructure, showing a 20% speedup over leading extraction tools.
- Design and evaluation of an algorithm to reduce extraneous file transfers and transfer time.
- Demonstration that Xtract can scale to process 2.5 million file groups with materials science extractors deployed to more than 2048 workers on a supercomputer.
- Application of Xtract to a large scientific data repository, the Materials Data Facility (MDF), and to a scientific Google Drive account.

The remainder of this chapter is as follows. §5.1 describes the Xtract design and §5.2 presents its architecture and implementation. §5.3 explores performance and scalability in scientific case studies. Finally, §5.5 summarizes the contributions.

5.1 Xtract

Xtract is a bulk metadata extraction system that provides on-demand metadata extraction from heterogeneous scientific file formats using remote and distributed computing infrastructure. Xtract performs end-to-end metadata extraction by applying a series of extractor functions to groups of files in a repository. The order of processes by which Xtract extracts metadata is as follows:

- Users interact with the **Xtract service** to initiate metadata extraction on a repository of data.
- Xtract invokes the **crawler** to traverse the files stored in a target repository, determine which files need to be grouped, and create an initial metadata record for each group.
- Xtract determines a dynamic extraction plan for file groups, including a set of extractors that will likely yield metadata. Note: the plan may be updated as metadata are obtained from allocated extractors.
- Xtract determines where extractors should be executed for each file and dispatches executor invocations to remote computing **endpoints** for execution.
- The remote endpoint receives the path to the file to be processed; if the file is not accessible locally, it initiates a download. It then applies the extractor to each file group before sending the updated metadata back to Xtract.
- At the conclusion of a group's extraction plan, the **validator** updates the metadata record to a user-specified format, and initiates the transfer of metadata to an external location.

I next describe each component of Xtract in more detail.

Xtract User Interface. Xtract offers an asynchronous interface via which users can register file grouping functions, metadata extractors, extractor containers, and compute and data endpoints; authenticate with cloud or compute providers; execute extraction and validation jobs; monitor the status of extraction jobs; and retrieve or deposit the extracted metadata. Users specify an extraction job to start the extraction process. A job includes a list of target repositories (and access credentials), paths specifying the root directories to be processed, a list of compute endpoints to be used, and a file grouping function (which may be "single file group").

Crawling. The crawler lists the contents of a remote storage system to identify what files need to be processed, and to extract minimal file system metadata (e.g., file name, size, creation date). Once a directory is crawled and all files identified, the grouping function is invoked in order to assign all files that need to be processed together to a single metadata object.

In addition to file groups, Xtract defines an additional level of grouping, called *families*. Families are used to reduce unnecessary transfer costs. For instance, if a file belongs to group A and group B, it may be more efficient to process both groups at the same location so as to not transfer the same file multiple times. I discuss the Xtract family generating algorithm called `min-transfers` in detail later in this chapter.

Extraction Orchestration. Xtract manages the metadata extraction process by applying a set of extractors to a file group. After crawling, Xtract dequeues each group and identifies an initial set of extractors to be applied, as identified by the crawler's grouping function, and selects an appropriate computing resource on which to execute the extractor. If Xtract opts to invoke the extractor on the machine on which all files in the group reside, then it serializes and transmits a '*family*' (containing a list of individual files) and extractor function(s) directly to that machine for processing. Alternatively, if any files in the group are stored only on another machine, Xtract initiates the transfer of those files from their host machine to the one conducting the extraction, and then proceeds as when the data are available locally.

Extractors. (I discuss extractor design in detail in Chapter 3). Extractors are functions that take a file group as input and generate a dictionary of extracted and synthesized metadata for that group. Xtract includes over 20 extractors (listed in §5.2) for myriad data types commonly used in science and engineering. Users may also define and add their own custom extractors. Extractors are implemented as a Python function or Bash shell script. Each extractor has an associated container (e.g., Docker) to encapsulate a runtime environment for that extractor and to provide access to libraries not otherwise available on the computing resource (e.g., Tensorflow or materials science packages). Containers also ensure that extractors can be deployed on different target systems.

Endpoints. Xtract requires a data substrate to access and move data between resources as well as a compute substrate to remotely execute extractors. I call these remote Xtract sites *endpoints*, where an endpoint contains both a data and compute *layer*. The data layer abstracts the remote storage system (e.g., file system, object store) and makes data accessible to the endpoint. Xtract's extractors can both access data stored in the data layer and write data from another endpoint. The compute layer represents the computing allocations available to process files. The compute layer is tasked with allocating compute resources (e.g., local cores, HPC nodes, or cloud instances), invoking metadata extractors on the files, and sending results back to the Xtract service.

Validation (and Transformation). The validation step ensures that the resulting metadata have all required attributes; it can also optionally transform the metadata into a schema more amenable for subsequent use. Validation enables users with different metadata requirements, for example because they work in different domains, to leverage metadata produced by the same extractors. Xtract users specify the validation/transformation method to be applied; it processes the supplied metadata and sends a valid JSON document to a

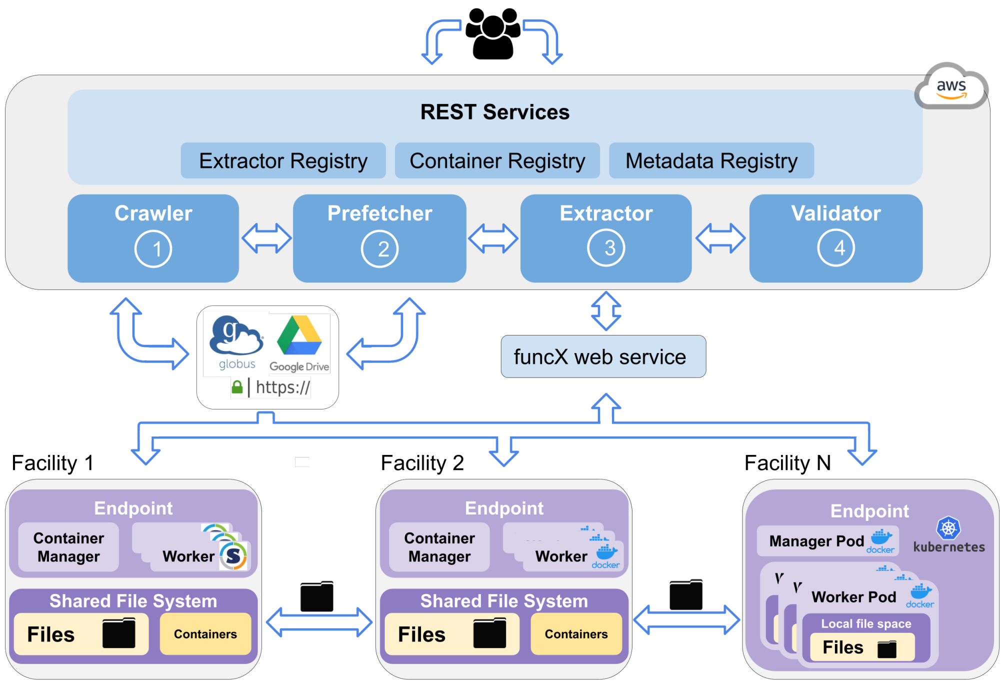


Figure 5.1: Overview of the Xtract architecture.

user's Globus endpoint.

5.2 Architecture and Implementation

Xtract is implemented as a service exposing a REST API for user interactions. It follows a microservices architecture in which each of the core components is deployed as a web service and exposes an API for coordination between services. Figure 5.1 presents an overview of the Xtract architecture.

5.2.1 System Components

The Xtract service receives extraction job requests via the REST interface and first records the job in an AWS Relational Database Service (RDS) instance. It then invokes the crawler to begin processing the target repository. Simultaneously, Xtract reads from the crawler's completed queue—implemented with AWS Simple Queue Service (SQS)—and determines an extraction plan for each file group. While much of the extraction plan focuses on determining which extractors to apply to which files, it also determines on which resources each extraction should be executed. If any file in a group has different source or destination endpoints, a task is placed onto an internal prefetcher queue to orchestrate the required transfers. Xtract then sorts all files into same-endpoint, same-extractor batches (coined Xtract batches), reads the container location and endpoint information for a file's extraction location, and then further batches multiple Xtract batches (coined funcX batches) and sends them to the funcX service. funcX serializes and dispatches each batch to its relevant endpoint. Xtract then polls funcX to retrieve task results from the endpoint. Based on the results, Xtract determines if additional steps should be added to the extraction plan. If not, it places the results onto a shared validation SQS queue.

The crawler is implemented as an elastically scalable microservice that is invoked by the Xtract service through a REST API. The input to the crawler includes a list of remote endpoints, the paths to be recursively crawled, authentication headers, file grouping rules to aggregate metadata objects, and any source-specific information such as the top-level URL for HTTPS-accessible data, Google Drive API tokens for Google Drive, or Globus Auth access tokens for Globus. The crawler service deploys a pool of crawl worker threads and a shared work queue for each metadata extraction job, and starts new EC2 instances, if needed (i.e., if current instances are overloaded). The shared work queue is initialized with the root paths specified in the extraction job. Worker threads retrieve a path from the queue, perform a list operation on it, apply the grouping function to the discovered files, and addnewly discovered directories to the work queue. Xtract supports a number of grouping functions, as granular as placing each individual file into its own group, and as broad as placing entire directories and subdirectories into a single group. In order to keep the crawler service operating with minimal overhead, and to account for repositories without local compute, grouping functions consider only metadata available from the crawler (e.g., filenames, extensions, paths, size). The crawler bundles the initial metadata into a universally readable family object, serializes it, and places it onto an SQS queue for return to the Xtract service. The crawler exposes a modular interface for crawling remote repositories with implementations for Globus, S3, and Google Drive, using their respective APIs, and remote POSIX file systems (using a Python function that is executed via an endpoint's compute layer).

The prefetcher is responsible for managing the movement of data between endpoints when required. The prefetcher reads tasks directly from a dedicated queue (populated by the Xtract service). For each file transfer job, the prefetcher first authenticates with the data layer on both the source and destination endpoints of each file, places the files into a batch, and then initiates the batch Globus Transfer of files between them. The prefetcher polls each transfer task until it is completed, and then places the task back onto Xtract's queue for further processing.

The extractor library contains information about each extractor and the endpoints on which they can execute (e.g., extractors whose containers are only available in Docker may not be run on Singularity-only systems). When users register a custom extractor, they provide an extraction function in Python or Bash, a path to a container, and a list of endpoint IDs on which the function is able to run. These function:container:endpoints tuples are registered with funcX to create FaaS functions to be used by the Xtract service. The funcX function ID, container ID, and endpoint ID are then stored in Xtract's RDS database. Listing 2 shows an example extractor function[1] that extracts metadata from a file stored locally on a compute endpoint. I provide an `xtract_sdk` Python SDK to simplify access to remote files and packing and unpacking metadata objects. I illustrate the `xtract_sdk` in Chapter 7.

1. This interface for building extractors is deprecated as of 2021. For an updated description of extractor design and creation, see the discussion in Chapter 7.

```python
def keyword_extract(event):
  import shutil
  from xtract_sdk.downloaders import GoogleDriveDownloader
  # A Python function located in the extractor's container
  import xtract_lib

  # Load transfer credentials and list of families
  creds = event["creds"]
  family_batch = event["family_batch"]

  # Apply the keyword extractor to all families
  for family in family_batch.families:
    kw_mime = family.files[0]['mimeType']
    is_pdf = True if 'pdf' in kw_mime.lower() else False
    path = family.files[0]['path']

    # Invoke the extractor library in the container
    mdata = xtract_lib.extract_keyword(path, pdf=is_pdf)

    # Package the metadata back into the 'family' object
    family.metadata = mdata

    # Remove the associated file, if necessary
    if family_batch.delete_files:
      shutil.rmtree(family.base_path)

  return {'family_batch': family_batch}
```

Listing 2: Example code of an extractor to extract keywords from documents stored in Google Drive.

The **endpoints** provide a computing and data fabric to abstract the complexities of accessing and using remote and heterogeneous hardware. Xtract leverages two existing technologies to create its endpoints—funcX [14] and Globus [13]. funcX endpoints provide a mechanism to dynamically provision computational resources and manage execution of metadata extractors within containers. Each endpoint also includes a reference to a container library such that extraction containers can be started on the machine. Depending on the target machine, the container library can either be retrieved from a remote location or is immediately accessible via a shared file system. Globus endpoints enable remote data management, including being able to list, move, and share files and folders. If Globus endpoints are deployed on two remote computers, Xtract can request that files be moved directly from one endpoint to another. Importantly, the data do not pass through Xtract or the Globus service. While endpoints also support direct download from cloud repositories such as Google Drive and AWS S3, they do not yet support the transfer of files to other non-Globus endpoints.

The **validation service** is implemented as an asynchronous microservice that can validate and transform metadata subject to a user's set of schemas: e.g., the 'passthrough' validator that converts a metadata dictionary into valid JSON, and the MDF validator that adapts the extracted metadata to one of 12 schemas. As metadata are processed, they are transferred to an endpoint of the user's choosing for post-processing (e.g., ingestion into a search index). The validation service acts on metadata objects as they are added to the result queue. These objects are dequeued, processed in accordance with the user's requirements, and then are queued for transfer to an external file system for client post-processing.

Xtract's **security model** ensures that bulk metadata extraction operations are performed on behalf of an authenticated and authorized user. Xtract uses Globus Auth [129] for authentication and authorization. Users must provide valid authentication tokens with appropriate authorization to initiate crawls, extractions, and validations. Xtract is registered as a Globus Auth resource server, allowing users to authenticate using a supported Globus Auth identity (e.g., institution, Google, ORCID) and enabling various OAuth-based authentication flows (e.g., native client) for different scenarios. Xtract has associated Globus Auth scopes via which other clients (e.g., applications and services) may obtain authorization for programmatic access. To support Google Drive repositories, I retrieve a user's Google OAuth token and use it in conjunction with appropriate Globus Auth tokens to both access data and perform extractions.

Xtract extractors are isolated in containers to ensure they cannot access data or devices outside their context. In particular, I use both Docker and Singularity containers to encapsulate extractors and enable their execution at various computing resources. Within the container, each function executes within its own local Python namespace to avoid program state changes between invocations.

The **XtractClient** facilitates REST communication between user programs and the Xtract service. I discuss the XtractClient in more detail and show example usage in Chapter 7.

5.2.2 Extractors

While I outline the breadth of extractors in Chapter 3, I summarize several more extractors leveraged as use cases in this work, and the types of files (or file components) on which they are meant to operate. I describe these extractors and present detailed performance information in a previous paper [113].

The **MaterialsIO set of extractors** [134] can process multiple common formats used in materials science. The set of extractors wraps the MaterialsIO file parsing library, which contains a number of parsers for atomistic simulations, crystal structures, electron microscopy outputs, density functional theory (DFT) calculations, and images. Since many file types generally used in materials science are processed in groups (e.g., VASP files generated from atomistic simulations), I have written a grouping function that executes at crawl-time and matches groups of files to a MaterialsIO extractor. All MaterialsIO extractors share a container runtime.

The **images extractor** extracts metadata from arbitrary images (e.g., plots, maps, and photographs) that are stored in common formats (e.g., *.png*, *.jpg*, *.tif*). The image extractor dynamically builds a workflow for each image by first determining its class (e.g., plots, photographs, diagrams, and geographic maps). To generate these classifications, I first extract a number of features from the image, including color histograms, and predict its class using a pretrained support vector machine (SVM) model. If the image is a photograph, I apply the ImageNet extractor mentioned in the following. If the figure is a map, I apply object character recognition (OCR) software to determine its geographic coordinates and return location tags (e.g., “South America”, “Montgomery, Minnesota”).

The **tabular extractor** processes data in common row-column formats, such as spreadsheets and database tables, that may contain a header of column labels. Metadata can be derived from the header, rows, or columns. Aggregate column-level metadata (e.g., mean and maximum) often provide useful insights.

The **keyword extractor** identifies uniquely descriptive words in unstructured free-text documents such as READMEs, academic papers (e.g., *.pdf* and *.doc* files), and abstracts. It uses word embeddings to curate a list of the top-n keywords in a file, and an associated weight corresponding to the relative relevance of a given keyword as a proper descriptor for that document

The library also contains extractors not discussed in detail here, including **hierarchical** for NetCDF and HDF files, **null-value** to determine null-values in tabular data, **Python** and **C** for mining information from programs, **semi-structured** for data in *.json* and *.xml* formats, **BERT** to extract key entities from text, and **ImageNet** to recognize objects in images.[2]

5.2.3 Optimizations

Xtract applies three optimizations to reduce metadata extraction costs: the creation of *family* objects to reduce the number of times a file is transferred, batching to amortize network and startup costs, and offloading tasks to other compute sites to use idle resources.

2. The names and functionalities of many extractors have changed since this chapter was published in HPDC 2021 [110]. For updated extractor information, please refer to Chapter 3.

Families. During crawling, file groupings are not necessarily disjoint: one file can belong to multiple groups. This, however, creates problems when deciding where to send each file group for extraction, as a file belonging to multiple groups may need to be transferred to disparate places, thus incurring unnecessary transfer costs. To avoid these costs, I introduce a collection data type called a *family*.

A family contains one or more groups whose individual file sets intersect. Because some directories are large, automatically considering *all* files to be members of the same family is detrimental to parallelization (i.e., the worker drawing that extraction task will certainly become a straggler). Thus, I set a user-configurable maximum group size $s > 0$. Thus, I can minimize transferring the same file twice, or inversely, transfer a file and then invoke all of its groups' extractors on it.

Transfer Minimization. In order to facilitate building families with minimal overhead, I developed the *min-transfers* algorithm that leverages Karger's Randomized Min-Cut algorithm [64]. The input to the algorithm is a multigraph $G =< V, E >$ of each directory across all file systems, where each node $v \in V$ is a file and each weighted edge $e(v_i, v_j) \in E$ represent how often files f_i and f_j appear in separate subgraphs. In simpler terms, w_e represents the number of times the file may be redundantly transferred. I isolate G into its connected subgraph components $g =< V_g, E_g >\in G$, as each connected component, by definition, shares no v (and therefore no f) with other g.

For each connected component $g \in G$, I run Karger's Min-Cut to determine an approximate minimum cut, producing two subgraphs. I recursively run Min-Cut on each subcomponent until $\forall g \in G, |E_g| \leq s$. At this point, all files (v) in a still-connected subcomponent are labelled as a family and considered a single metadata extraction task object. See Algorithm 1. As each group is packaged as one or more minimum-transfer families, the crawler asynchronously enqueues them for processing by the Xtract service.

Algorithm 1: Min-Transfers

```
Inputs: G=<V, E>: file system graph

families = list()

// Step 1: Make queue of connected components
connected_components = get_connected_components(G)

// Step 2: Iteratively run Karger's min-cut in each component
for each comp in connected_components:
    if $|comp.V| \leq |S|$:
        families.append(comp)
        continue
    else:
        newcomp_1, newcomp_2 = karg_mincut(comp)
        connected_components.put(newcomp_1)
        connected_components.put(newcomp_2)

// Step 3: return a list of families
return families
```

To calculate the efficiency of my approach, I start with $O(E) = O(|V|^2)$ complexity in the worst case when all files are in a group with each other file. In the worst case, only one node is removed on each iteration, which means it can take $|V| - r - 1$ iterations to get the largest component of the graph down to maximum scalar group size r. Therefore, this algorithm operates in $O(|V|^2 * (|V| - r - 1)) \approx O(|V|^3)$ time.

Batching. Batching enables Xtract to amortize the overheads associated with transmitting thousands of function invocation requests to an endpoint. Xtract batches tasks at two levels: file families and extraction requests. First, **Xtract batching** combines families that use the same extractors into a single funcX task. This reduces the cost of transmitting many families through funcX, through the endpoint, and to the extractor, and back, across all subtasks in the task. These batches are transparent to funcX. Second, I exploit **funcX batching**

to reduce the number of funcX web service requests. Here, I create batches of tasks to be executed and submit each batch individually to funcX. funcX expands the batch into a set of individual function invocations. I also use funcX's batch polling functionality to retrieve the status and output of completed functions. Batching at both levels not only amortizes costs at the function execution level, but maximizes file throughput through the Web services.

Offloading. Xtract can offload tasks to other idle resources to maximize the total task throughput. To determine the resources on which extraction should occur, Xtract uses a rule set that varies between metadata runs and relationships of (i) how long it would take to move a group to a given remote computer and (ii) how long the extraction is expected to take, given information about the file's size and extraction time, on a given computer. These rules are implemented as user-configurable modes: offload n bytes (ONB) and random (RAND). In ONB, each computer is given a size limit (either max or min); if a computer is fully occupied with work, all files on that computer that are larger (for max) or smaller (for min) than the size limit are transferred to another, allowing Xtract to leverage idle resources. In RAND, a specified % of files are selected at random to move from a 'main' machine (e.g., cluster) to worker machines (e.g., cloud instances). Xtract invokes batch file transfers before the extractors are serialized and shipped, and only sends the extractors upon confirming that the transfers are completed successfully.

5.3 Evaluation

I examine Xtract's performance in terms of scalability, throughput, latency, and application to real-world research data repositories. I also evaluate the batching, file fetching, and offloading optimizations, and the min-transfers algorithm. To showcase the flexibility of Xtract's design, I leverage a diversity of research cyberinfrastructure.

5.3.1 Experiment Testbed

Xtract services are hosted on an m4.16xlarge AWS EC2 instance with 256 GB RAM and a 2.4 GHz Intel Xeon E5-2676 v3 (Haswell) processor, located in the us-east-1 availability zone (Northern Virginia, USA). Other instances such as the PostgreSQL RDS database and SQS queues are also located in us-east-1. I use four compute resources: Theta supercomputer, River Kubernetes cluster, Jetstream, and University of Chicago Midway cluster.

Theta is a 11.7-petaflop Cray XC40 system comprised of second-generation Intel Xeon Phi "Knight's Landing" (KNL) processors. Its 4392 nodes each have a 64-core processor with 16 GB MCDRAM, 192 GB DDR4 RAM, and are interconnected via high speed Infiniband. Data are stored on Theta's Lustre file system.

River is a Kubernetes cluster housed at the University of Chicago. The cluster has 70 nodes, each with 48 cores and 256 GB RAM. Nodes are connected with a 10 Gbps network and the cluster is accessible via two 40 Gbps links to the campus science DMZ.

Midway is a campus cluster with 572 nodes and 16 016 cores. It has both Intel Broadwell (28 core, 64 GB RAM) and Skylake (40 cores, 96 GB RAM) nodes. Tightly-coupled nodes are connected with 1000 Gbps Infiniband interconnect, loosely coupled nodes are connected with 40 Gbps GigE. I use the Broadwell partition.

Jetstream [119, 127] is an open research cloud composed of two homogeneous clusters at Indiana University and the Texas Advanced Computing Center. Each cluster has 320 Dual Intel E-2680v3 (Haswell) nodes, each with 24 cores and 128 GB RAM. Jetstream uses 40 GigE for network aggregation and has 100 Gbps connections to Internet2. Jetstream includes nine different cloud virtual machine types ranging from 1–44 vCPU. I use m1.large (10 vCPU, 10 GB RAM) instance types in the TACC cluster.

I also leverage **Petrel** [1] as a data store. Petrel is a data service hosted at ANL[3] that provides user-managed storage allocations to the research community. It is an eight-node

3. Petrel was decommissioned by ANL in early 2022.

cluster with a Ceph file system offering 3 PB of storage. Access is provided via Globus. It has no connected compute resources.

For experiments with Apache Tika [79], I deploy an air-gapped Tika server locally with n incoming processing threads, where n is the number of funcX workers being evaluated on that machine. As Tika has no built-in data fabric, I use Xtract to move files between resources, when appropriate.

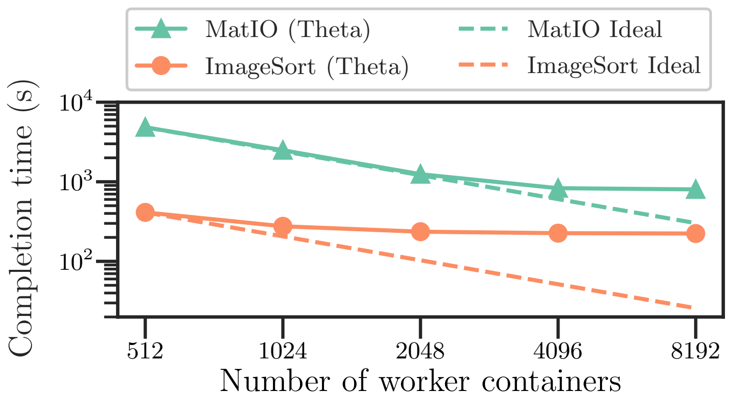
5.3.2 Scalability and Throughput

Effectively processing metadata from the data sizes present in modern science requires Xtract to scale to a large number of concurrent extraction processes. To evaluate this, I analyze the strong and weak scaling of the Xtract service using endpoints deployed on ANL's Theta supercomputer.

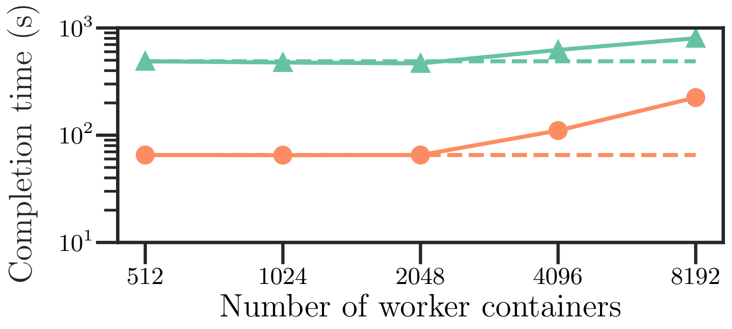
Strong scaling measures performance when the total number of extractors applied to a set of files is fixed; weak scaling measures performance when the average number of extractor invocations is fixed. As crawling time is negligible compared to the overall execution time, I focus here on only the metadata extraction process. I evaluate crawler scaling in §5.3.4.

Each experiment measures the total time required to complete the bulk metadata extraction task, from the request to the Xtract service to the result being returned. This time includes the time for the Xtract service to dequeue families, construct extraction plans, and push requests to funcX; funcX to deploy families and functions to funcX workers on each endpoint; and each funcX worker to invoke the function on the file and return the results. To evaluate Xtract'sscalability, I use just two extractors: the short-duration ImageSort extractor that classifies images as one of five types (photograph, diagram, plot, geographic map, and other) and the long-duration MaterialsIO extractor.

I apply these extractors to two representative datasets. For the image extractor, I use the 2014 Common Objects in Context training dataset of 80 000 images (14 GB) [76]. For Ma-



(a) Strong scaling



(b) Weak scaling

Figure 5.2: Strong and weak scaling of ImageSort and MaterialsIO metadata extraction tasks.

terialsIO I use a subset of the MDF dataset: 200 000 file groups (1.1 TB), chosen uniformly at random. I use an Xtract batch size of two for ImageSort and eight for MaterialsIO, and a funcX batch size of 16 (i.e., Xtract sends batches of 16 requests to funcX).

Strong Scaling. Figure 5.2(a) shows the completion time of 200 000 extractor requests with an increasing number of worker containers on Theta. For ImageSort, completion time decreases until 2048 workers are deployed, after which the short task execution time limits performance. For the longer MaterialsIO extraction, one can see that the completion time decreases until 4096 workers are employed. I conclude that Xtract is primarily limited by the rate at which funcX delivers tasks and data to an endpoint.

Weak Scaling. To evaluate weak scaling, I deploy concurrent extraction tasks where each worker, on average, receives 24 ImageSort and MaterialsIO extraction tasks. I see in Figure 5.2(b) that Xtract maintains good throughput for both the ImageSort and MaterialsIO extractors on up to 2048 workers, but that the longer-duration task (MaterialsIO) again scales better than its shorter ImageSort counterpart as the number of workers increases to 4096.

Throughput. I observe a maximum extraction throughput (successful extraction invocations per completion time) to be 357.5 for ImageSort and 249.3 for MaterialsIO, respectively, on Theta.

5.3.3 Latency

A decentralized FaaS-based architecture must engage multiple components to place functions and files where needed for extraction. To better understand the resulting costs, I measure per-component latencies when submitting a single unbatched metadata extraction task (extracting keywords from a free-text document) to an endpoint on River: see Figure 5.3. As this endpoint has no shared file system, Xtract must transfer the file in from either a Globus or Google Drive endpoint.

The time cost of the crawler service, t_{cs}, is predominantly due to Globus Auth and remote Globus directory listing requests. Other crawler service events such as grouping, calculating the min-transfer families, and packing metadata objects are relatively short (less than 20 ms) in comparison. The 539 ms required to report the task back to the Xtract service is high as it includes the cost of enqueueing and dequeueing the task from SQS.

Once the task is received by the Xtract service, the majority of the cost t_{xs} is due to

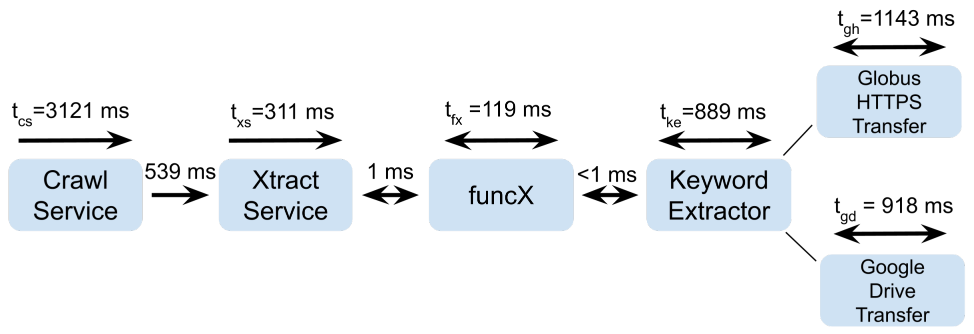


Figure 5.3: Xtract latency breakdown across all components (boxes) and the communication costs between them. Uni-directional arrows imply I measure data flow latency in just one direction (downstream), whereas bi-directional arrows imply the sum of latency in both directions (downstream and upstream).

resolving the endpoints and container associated with a given metadata extractor from the RDS database. These values are cached for subsequent requests. The cost of determining an extraction plan for a group is negligible.

The funcX invocation costs (t_{fx}) represent the time required to send the task through the funcX service to the compute layer (e.g., a funcX endpoint containing multiple funcX workers) on River. Once a given task is transmitted to funcX, it also incurs an authentication/authorization cost using Globus Auth.

Once the task reaches the endpoint, it is dispatched to an appropriate warmed Docker container on an idle Kubernetes pod. A majority of the keyword extractor cost, t_{ke}, arises from using Python libraries that process each word in the file, tokenize them, and then analyze those tokens to generate keywords. In the case where the file should be fetched, moving a file via Globus HTTPS, t_{gh}, or the Google Drive API, t_{gd}, is more costly than the extraction itself (i.e., in general, $t_{gh}, t_{gd} > t_{ex}$).

Many costs are amortized when the scale of the metadata extraction task is increased. For example, crawling a directory of 1000 files is much more efficient than performing 1000 individual requests. Similarly, the extractor can download many files in a family in parallel to increase overall throughput. Furthermore, funcX costs can be reduced by batching extraction

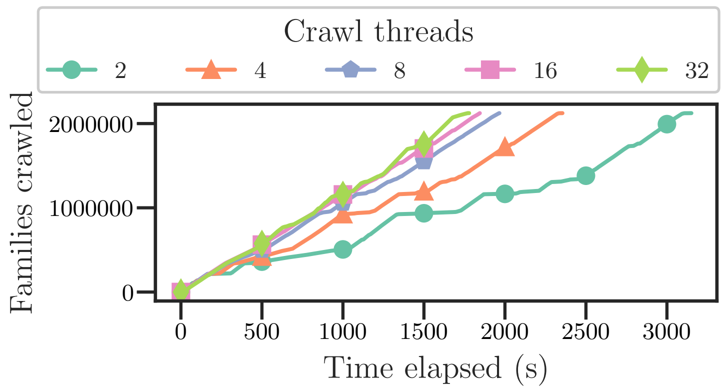


Figure 5.4: Number of files crawled over time for 2, 4, 8, 16, and 32 worker threads for 2.3M files from MDF.

tasks into a single request.

5.3.4 Crawl Parallelization

In order for Xtract to maintain a high-throughput stream of data to workers, it is important that the crawlers efficiently process directories and enqueue family objects. I evaluate the effects of parallelizing the number of workers processing directories from the crawler's queue. I perform crawler parallelization experiments on an AWS t3.medium instance (2 vCPUs and 4 GB RAM). Figure 5.4 shows performance for crawling all 2.3M files on MDF, requiring 50 minutes with just two workers, and ∼25 minutes on 16–32 workers. I observe minimal benefit after 16 concurrent workers, due to network congestion on the instance caused by large file lists simultaneously returning from Globus.

5.3.5 Batching

I evaluate the effects of batching in two ways (§5.2.3): **Xtract batching** combines tasks on the Xtract client such that tasks are serially processed by the same extractor and **funcX batching** reduces the number of requests to funcX. To this end, I try to find an optimal batching pattern by submitting 100 000 MaterialsIO tasks to the endpoint and varying both

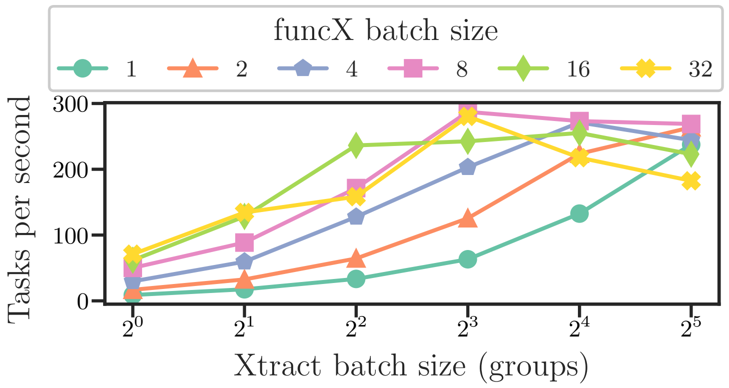


Figure 5.5: Extraction tasks processed per second when varying the Xtract batch size and the funcX batch size.

Xtract and funcX batch sizes from 2–32. I have 224 Midway workers processing these tasks. The results of this experiment are shown in Figure 5.5. I observe that overall throughput is maximized by extracting 8 extraction tasks per batch and sending 8–16 of these batches at a time to funcX.

5.3.6 Offloading and System Comparison

Xtract's decentralized metadata extraction allows metadata extraction tasks to be offloaded to remote resources with a compatible compute layer that the user is authorized to use, for example, to use additional idle cloud or HPC allocations. Opportunistically offloading tasks enables Xtract to minimize the makespan of extraction tasks. To this end, Xtract enables users to define, via its RAND scheduling policy, a percentage of the data to be sent to alternative resources. To evaluate the effectiveness of this strategy, I measure the performance of offloading tasks to the Jetstream cloud. In particular, I evaluate the makespan of extracting 100 000 files using a 56-worker endpoint on the Midway cluster when offloading 0%, 10%, and 20% of the files to 10 idle funcX workers on a Jetstream instance. Furthermore, I compare Xtract to a similar offloading setup using the same files, but instead running Apache Tika at the endpoints for metadata extraction. I configure Tika to automatically detect each file's type and execute the 'best' parser from its default library. I present the results of the three offloading scenarios for both extraction tools in Table 5.1.

I observe that there is an equilibrium point between transferring too few and too many files. In this case, if Xtract transfers too few files (0%), then too many tasks remain queued waiting for resources on Midway. These tasks may queue longer than the time to transfer and extract them on Jetstream. When offloading too many files (20%), Jetstream's 10 funcX workers or Tika processes become saturated, and the Midway workers are underutilized. In the best case, I transfer just 10% of all files and see total extraction occur 8% faster than when processing everything *in situ.* Additionally, Xtract executes its extractions 20% faster than Tika, on average, but using a different (and less domain-specific) set of parsers.

Table 5.1: Completion time for various RAND policy offloading rules from 56 concurrent workers on Midway to 10 concurrent workers on Jetstream: Xtract and Apache Tika.

System	**Percentage Transferred (%)**	**Transfer Time (s)**	**Completion Time (s)**
Xtract	0%	0	1696
	10%	374	1560
	20%	655	1662
Apache Tika	0%	0	2032
	10%	384	1868
	20%	649	1935

A key use case for Xtract is to process files residing on a storage system without an associated computer, in which case data must be transferred to permit extraction. I show in Figure 5.6 results from such a configuration. Specifically, Xtract moves (prefetches) 200 000 MDF files from Petrel to Midway using 10 concurrent Globus transfer jobs, and extracts metadata on 4–32 Midway nodes, each with 28 workers. I observe that the time required to crawl the data is small compared to the prefetch and extraction costs; that file prefetch (transfer) incurs the majority of the time; and that on 32 nodes, Xtract processes the data nearly as quickly as it arrives.

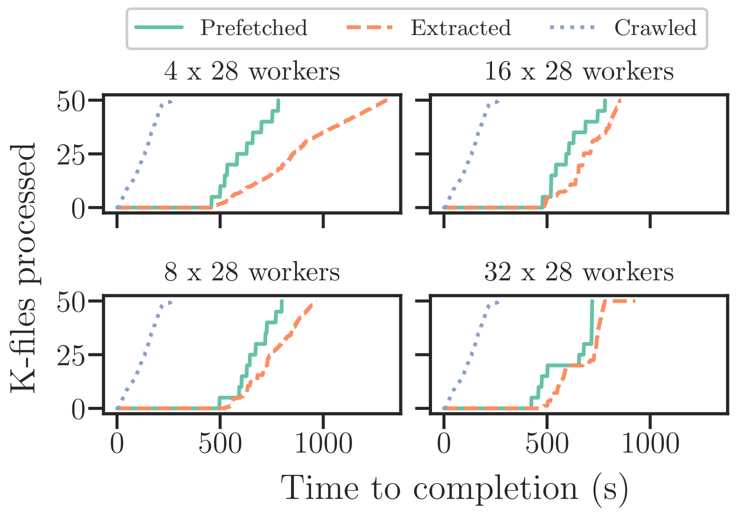


Figure 5.6: Bulk metadata extraction times for an MDF subset processed on 4, 8, 16, and 32 remote Midway nodes (each running 28 workers).

5.3.7 Min-Transfers Grouping

Xtract's FaaS compute layer effectively decouples the data storage location from the extractor execution location. However, in such an environment it is possible that data may be moved unnecessarily (e.g., when the same file is included in multiple families each moved to different compute endpoints). Xtract's lightweight min-transfers algorithm aims to minimize overall transfer time and data transferred by batching groups that have intersecting sets of files into family objects. The min-transfers algorithm is automatically applied to each directory as part of the crawler. To be effective, the min-transfers algorithm would significantly reduce transfer time in exchange for comparatively small overheads in the crawler. I seek here to explore the benefit of applying the min-transfers algorithm against the regular approach of simply transferring each file group separately, regardless of the overlap between groups.

Figure 5.7 shows performance with and without min-transfers when crawling 100 000 (161 GB) randomly selected files on both Midway2 and Petrel and then transferring those files to four Jetstream instances for extraction. I observe that in the regular approach, 3246 of the randomly selected families contain multiple files, leading to 20 258 files (32 GB)

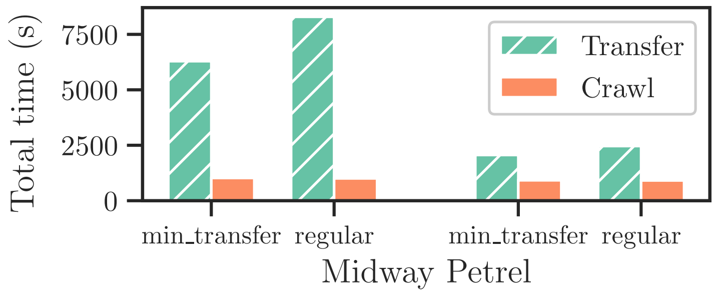


Figure 5.7: Min-transfers algorithm influence on crawl and transfer times when moving data to Jetstream from the Midway2 and Petrel file systems.

that are transferred redundantly. The figure shows that min-transfers adds little overhead to crawling. The regular crawls on Midway2 and Petrel took 913 and 1005 seconds, respectively. The slowdown caused by min-transfers is only 19 and 7 seconds, respectively: a penalty of less than 1%. The transfer time to Jetstream from Midway2 decreased by 24% (from 8291 to 6290 seconds, at an effective transfer rate of 26 MB/s), and from Petrel by 16% (from 2464 to 2060 seconds at an effective transfer rate of 79 MB/s). I conclude that the min-transfers algorithm helps reduce both transfer time and redundant bytes transferred.

5.3.8 Case Studies

To examine whether Xtract is capable of bulk metadata extraction of real, heterogeneous data stores using heterogeneous computing resources, I outline my experience applying Xtract to MDF and a graduate student's Google Drive repository.

Case 1: MDF. I first examine Xtract's performance on the 61 TB, 2.5 million group MDF repository. I conduct this test using a Theta endpoint with 4096 workers evenly spread across 64 nodes. Xtract crawls the entire repository in 26.3 minutes using 16 parallel crawlers. The Xtract service begins extracting data within 3 seconds of the crawler being initiated as file groups are returned asynchronously.

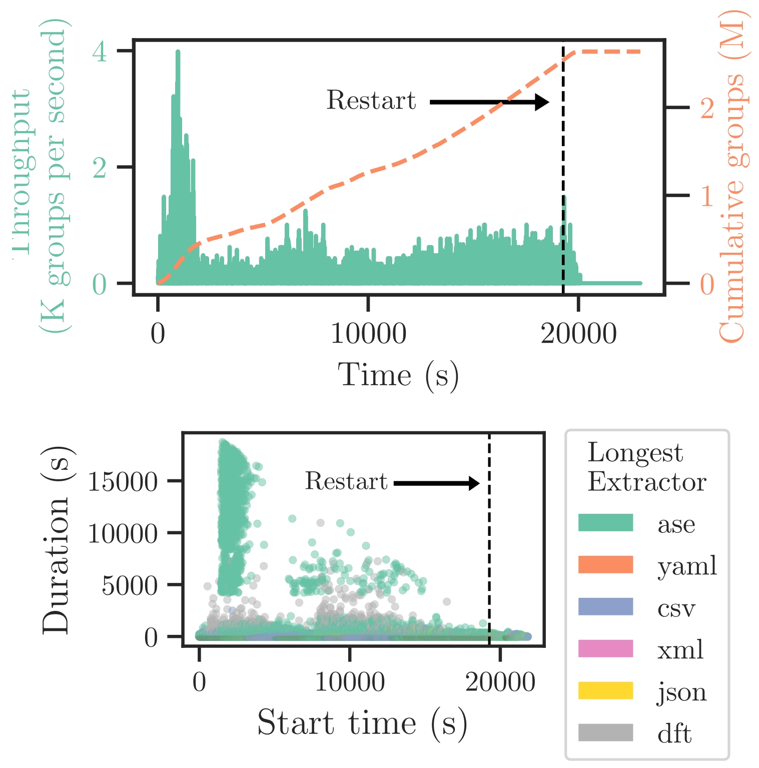


Figure 5.8: Metadata extraction on all of MDF. Top: Throughput in K-groups per second (blue line) and cumulative groups processed (red line) over time. Bottom: Per-family extraction duration vs. start time, colored by the extractor that took the longest. In both figures, the black-dashed line at 6 hrs show when extraction was terminated and restarted from checkpoint

Full extraction from MDF data took 26 200 core hours and 6.4 walltime hours. Figure 5.8 shows both throughput (groups processed per second) and cumulative groups processed over time. The higher throughput in the first hour is due to the order of task submission, as many long-duration tasks saturate multiple funcX workers. A graph of extraction start time by duration for each processed family is shown in Figure 5.8. It follows that that many families whose overall extraction time is dominated by the compute-intensive ASE extractor begin executing within the first hour, with many such families taking multiple hours to finish.

These results also highlight the reliability of Xtract's processing. Theta's scheduling policies allow users to allocate under 256 nodes for only six hours at a time, less than the total extraction time. Xtract checkpoints and resubmits the remaining tasks during a second allocation some time later. For this experiment, Xtract checkpointed progress via a 'checkpoint-flag' in the extractor that, when present, flushes each processed group's metadata to disk on completion. When funcX returns a heartbeat to the Xtract service stating that a family's task id is lost (i.e., the allocation ended), then the entire family is resubmitted, and in the presence of the 'checkpoint-flag', the metadata are reloaded. Figure 5.8 shows that Xtract was able to restart the job with minimal overhead at 19 274 seconds. In sum, the total metadata spanned 2.5 million files (14 GB).

Extracting metadata without transferring data is particularly valuable in the case of large many-file repositories. For example, despite the fact that Theta and Petrel are located in the same machine room, transferring all 64 TB of MDF to Theta would take 13.3 hours: double the time required to perform extraction on Theta.

Case 2: Scientific Google Drive Repository. To explore the effectiveness of Xtract when applied to a smaller, uncurated repository not mounted to a computing system (e.g., when an extraction must move data), I consider the Google Drive repository of a graduate student. This Google Drive repository contains 4443 files: 2976 text files, 333 tabular files, 564 images, 184 presentations, 1 hierarchical file and 6 compressed files. For 379 files, Xtract was unable to derive an associated type, so Xtract initially treats them as free-text files. Due to the absence of a 'presentation' extractor, Xtract also treats presentations as free-text files. As compute is not available on Google Drive, I configure Xtract to use 30 Kubernetes pods on River.

Table 5.2 presents statistics on the extraction process, including the average extraction and transfer time for each extractor type. There are more extractor invocations (4980) than total files (4443), as multi-typed files are processed by multiple extractors.

Xtract completed the extraction process in $\sim$35 minutes or 23 total Kubernetes pod-hours. As each extraction plan for a file may contain up to five extractors, and because

Table 5.2: Graduate student Google Drive extraction statistics.

Extractor	**Total Invocations**	**Avg. Extract Time (s)**	**Avg. Transfer Time (s)**	**Avg. File Size (MB)**
Keyword	3539	2.76	1.38	0.559
Tabular	333	0.21	0.31	0.024
Null-Value	333	0.84	0.30	0.024
Images	774	1.06	0.80	4.0
Hierarchical	1	2.2	5.9	14.0

Kubernetes pods do not mount a shared disk, a significant portion of this time was spent transferring data and starting new extractors, incurring a cold-start cost of ∼70 seconds per container. While being able to build and execute a rich metadata extraction plan for an average student's repository in a handful of minutes is certainly valuable, one can again see the benefit of being able to offload extractions to another location.

5.4 Related Work

In this section I outline related work in remote execution systems—catalysts for the scaling and decentralization supported by Xtract.

As discussed in Chapter 1, alternative extraction systems use either entirely local or entirely centralized computation to perform extractions. I evaluate metadata extraction workflows, or extraction plans, atop the funcX federated FaaS platform [14] that is designed to support distributed execution across computing resources. In the following, I discuss related work in FaaS as well as outline competing edge and FaaS systems.

Modern high-speed networks and simplifying abstractions such as FaaS make it easy to perform computation in different locations. Xtract leverages remote function execution for extractor execution to deliver a flexible metadata extraction service capable of bulk metadata extraction in distributed systems. AWS Lambda [101] is an event-driven, serverless platform for function execution that has seen wide-spread adoption in industry to lower infrastructure costs, and it has multiple commercial competitors [47, 82]. AWS Snowball [102] is an industrial platform for edge computing and data migration. While I have shown in past work [16] that Lambda is the lowest-latency FaaS system, it is not open source and impossible to run on HPC (i.e., one must bring their own compute resources for metadata extraction). In that same article, funcX effortlessly runs on diverse cyberinfrastructure (including HPC), and has comparable latency to Lambda. Additionally, there are several open source FaaS alternatives [40, 36], but opt to use funcX for its ease of setup, seamless remote execution across compute facilities, and its ability to share authentication protocols with the Globus ecosystem.

5.5 Summary

Traditional metadata extraction methods either act entirely on locally available files or move data to a central system (e.g., cloud). In contrast, Xtract implements a hybrid model in which metadata extractors are executed on remote and heterogeneous computing endpoints. Xtract leverages the funcX federated FaaS platform to dispatch extractors for remote execution and the Globus research data management platform for moving data between endpoints. I have demonstrated that Xtract can scale well with materials science extractors concurrently executing across 2048 funcX workers on an endpoint, crawl millions of files, and support batching for better performance. As a measure of Xtract's efficacy, I showed that Xtract can crawl and extract metadata from the 61 TB MDF repository in just over six hours.

CHAPTER 6
EVALUATION OF METADATA ON REPOSITORY NAVIGABILITY

Previous chapters focused on the design and evaluation of the Xtract automated metadata extraction system. While I have shown that it has many desirable qualities—fast, intelligent, flexible, and quality-conscious—I am yet to address what is arguably the most important question: *do these metadata actually correspond to research value?* Exploring this question is particularly important, as no prior work examines whether, and the extent to which, automatically extracted metadata enhance users' collective ability to navigate science repositories.

Formally, I evaluate the following question in this chapter: *given a repository, a user's navigation needs, and an arbitrary interface to an automatically-populated data catalog, how confidently, correctly, and quickly can users navigate data?* To explore this, I conducted an intensive, two-part mixed-methods study of six frequent users of two U.S. Department of Energy (DOE) national laboratory research repositories.

In Part 1, I used a combination of surveys and informal conversations to enumerate participants' repository navigation requirements—both of the metadata and research tasks.

In Part 2, I observed participants as they performed a number of navigation tasks on a given interface (search portal or Jupyter notebook) connected to an automatically generated data catalog (i.e., a search index).

The remainder of this chapter is organized as follows: I first describe related work in §6.1. I present a two-part methodology in §6.2. I detail the results of Part 1 and Part 2 in §6.3. I conclude with lessons for the design of metadata extraction systems like Xtract in §6.4.

6.1 Related Work

This work builds on prior efforts related to file storage navigation.

Understanding user file storage behaviors is vital to understanding the breadth of their repository navigation needs. Groups have extensively studied user behavior on personal computers, where users have, on average, 5000–8000 files stored [46, 55] organized hierarchically into directories to reflect their usage for various activities [62]. I look to broaden the scope of these efforts to include large science file systems that can contain millions of files grouped into folders.

When attempting to analyze the catalysts of navigation, many have studied the influence of high-quality metadata on users' search capabilities. Some have attempted to enumerate the entire space of possible questions that can be asked of HDF files—a common science data format—and explored the potential numeric- and string-based queries that users could write to retrieve files from a large astronomy data set [141]. They find that it is important to extract and index certain metadata attributes to enable low-latency queries. In this study, I look to examine users' ability to explore their data, evaluating user performance and perceived utility when conducting individualized navigation tasks. I also customize metadata extraction to include useful searches beyond just numbers and strings, such as graphical representations of data, and explore two additional science domains.

Others have leveraged user studies to show the value of metadata in file retrieval contexts. In one study [18], Open Directory Project [90] metadata improve user-reported web search quality. In a study of the Spyglass file navigation interface [73], metadata extracted from NetApp's storage systems constitute a rich, searchable index filled with metadata that are extracted to facilitate users' file management tasks, as learned from a survey of the storage system's users. I look to similarly prioritize metadata based on user input from informational interviews, but additionally plan to validate users' performance and utility when navigating their data via the extracted metadata.

6.2 Methodology

To evaluate the influence of automatically extracted metadata on the navigation of science data, I conducted a live, two-part user study on 6 users of two research repositories. Part 1 consisted of informational interviews where I recruited participants, learned about their interactions with the science repository, and identified their navigation requirements. In Part 2 I leveraged the navigation requirements identified in Part 1 to (i) construct a specialized metadata extractor for each project, and (ii) observe the extent to which the automatically extracted metadata enable participants to perform navigation tasks. I conducted the user study in two parts due to the significant time required to construct extractors and execute metadata extraction jobs. I describe both study parts in detail in the following.

6.2.1 Part 1: Recruitment and Informational Interviews

To recruit eligible projects, I first consulted with the labs' computing facility leadership to obtain a list of projects, PIs, and storage allocations (in TB) associated with a large lab storage facility. In an introductory email to project PIs, I presented the project title, a description of the two-part study, and explained that participating projects could keep any search indexes, extractors, and interfaces associated with their participation. This led to having conversations with 12 users across 7 unique science projects. In the initial conversations, I sought to exclude any projects not meeting the following parameters:

- **Opportunity for navigation improvement.** I sought projects where improving users' navigation capabilities could provide significant research value. I excluded projects that did not have a single navigation task that (1) took users more than 5 minutes to complete, or (2) users were unable to complete.
- **Automated extraction requirements.** I prioritized projects with large volumes of data in varied data formats. I focused on those for which little existing metadata was

Repository	Num. Datasets	Total Size (GB)	Extensions
Battery	834	455	CSV, JSON
Spectroscopy	1.3M	123K	HDF, IMM, BIN

Table 6.1: Repository composition by project.

available and thus there was a need to automate metadata extraction.

- **Broadly impactful.** I focused on projects for which the ability to navigate research data was vitally important to many users ($n \geq 20$). These users could have been any combination of active users who, at least one time per month, either read or wrote files in the repository.

- **Shareable artifacts.** To perform the experiments, I required that projects be willing to securely share data and software (to be integrated into extractors), and that I could share and publish anonymized accounts of their participation in the study.

With these criteria, I narrowed my scope to two anonymous projects—spectroscopy and battery modeling—whose contents I illustrate in Table 6.1. I wanted to study the most critical stakeholders—active, long-term users of the repository. To accomplish this, I required that users have 12 months experience with the repository and have interacted with the repository on a roughly-monthly basis. I identified 3 participants meeting these criteria in each project. Personas meeting these criteria included domain scientists who specialized in science content and instrumentation, computational scientists who understood the science content *and* relevant computational methods, and computer scientists who constructed necessary research computing infrastructure.

In the Part 1 interviews, I met with each participant individually via Zoom for 60–90 minutes and loosely followed a structured survey to better understand their data navigation needs and potential metadata that could help address these needs. I specifically asked participants to draft up to six representative "boilerplate" scenarios that could be used to craft navigation exercises for Part 2. For instance, a simple boilerplate could have resembled "I want to count the number of files where the observed {temperature OR speed OR voltage} is between {minimum value} and {maximum value}." I used axial coding to identify overlaps between participants' navigation needs. Additionally, I encouraged participants to share with me any relevant software tools that I could encapsulate into Xtract extractors.

6.2.2 Part 2: Preparation and Interactive Navigation Exercises

In Part 2, participants performed 7–9 data navigation tasks meant to simulate the actual navigation needs outlined in Part 1. In preparation for Part 2, I constructed metadata extractors from the software tools shared in Part 1, constructed interfaces so participants could interact with the automatically extracted metadata, and prepared a list of relevant tasks to be completed by participants. In measuring navigability, I explicitly sought to measure performance in the following ways:

- Participant confidence that results are correct.
- Participant correctness.
- Time taken to reach result.
- Time taken relative to alternative methods.
- Perceived participant utility of extracted metadata.
- Other unstructured participant feedback.

The remainder of this section outlines the methodology for extractor construction, the metadata extraction job, interface construction, and navigation task creation.

Metadata Extraction. I first created extractors for both battery modeling and spectroscopy that, when paired with Xtract's built-in extractors, extracted metadata that may have been useful to researchers when performing their desired navigation tasks. I included the necessary attributes by integrating participants' existing software scripts with their recommended open source libraries. After I registered the extractor with Xtract, my research team conducted an extraction job for each project, and automatically ingested metadata into an Elasticsearch index. To enforce strict data confidentiality, I performed all extractions on an HPC cluster located at the same laboratory where a project's data were stored.

Interface Construction. In this study I investigated whether, regardless of the search interface, participants could leverage automatically extracted metadata to navigate research repositories effectively. In order to evaluate the influence of metadata, rather than the interface, on data navigation, I created two feature-light interfaces for participants so as to avoid bias in performance or perception (i.e., confidence) caused by one or the other.

The first of these feature-light interfaces was a JupyterHub notebook that exposed to users an API to *count*, *dump* (to pandas dataframe), *visualize*, and *query* with basic Elasticsearch [69] AND/OR operators. The second was a search portal that presented to participants a classic search interface, including up to five search facets and a free-text search box.

Both interfaces linked directly to an Elasticsearch index. To avoid evaluating interface features, I did not include common search modalities such as ranking, file recommendation, or auto-complete.

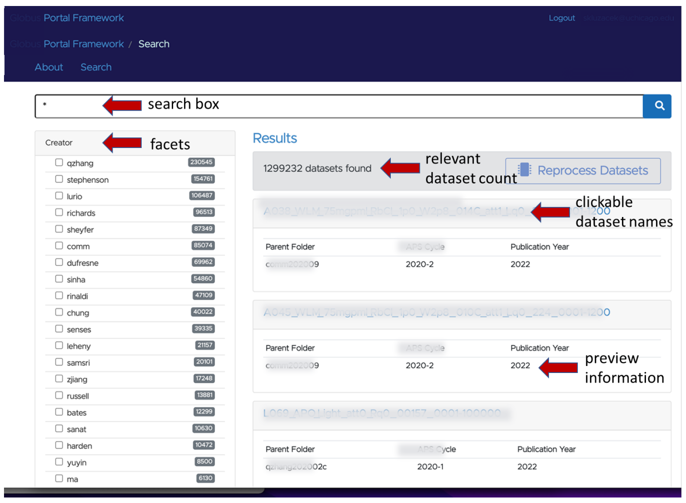
Task Creation. In an attempt to map a simulated, interactive environment to real-world research situations, I designed questions that linked to the data navigation needs of participants and their peers outlined in Part 1. To accomplish this, I took the boilerplate navigation tasks from Part 1 and parameterized them with specific values observed in the extracted metadata. For instance, if a participant had stated that they wanted to search on a given element between minimum and maximum temperature values, I could have written the following task: "*The date is May 13th, 2021. You want to look for all past experiments where gold is used between temperatures 40 and 49 degrees Celsius. How many data sets is this?*" Additionally, I attempted to vary the navigation type (discover, find) and the overall difficulty between tasks; I created tasks along a hardness scale: 2 easy, 2 medium, 1–2 hard, 1 null, and 1–2 "own" tasks, described as follows:

- **Easy tasks** required users to navigate using 1–2 obvious metadata elements.
- **Medium tasks** required 2–3 metadata attributes and some operation performed over them (e.g., convert time units from the task to the available metadata).
- **Hard tasks** required 3+ metadata attributes and multiple views of data (e.g., flipping between a count API, a dataframe, and a graph) or using domain-specific insights to solve (e.g., requiring participants to comment on voltage curves).
- **Own tasks** were *any* tasks that participants thought would aid their research and were not previously completed in the study (i.e., participants brought their own tasks). Having these tasks ensured that performance was not due to a favorable question set. This enabled participants to derive and execute any tasks not previously identified in Part 1.
- **Null tasks** did not contain the required metadata to solve, ensuring that users did not exhibit positive perceptions in the case of missing metadata. These tasks were correctly solved only by stating that the task was unsolvable.

Interactive Session. Each participant joined me in a 60-minute shared-screen Zoom call, where I instructed them to open two web browsers. On the first, I had them load a local web site that displayed navigation tasks. On the second, I had them load the specific interface used for navigation (i.e., the search portal or Jupyter notebook). Participants were asked to read the documentation (est. 3 min.) and familiarize themselves with the interface (est. 1 min.) before continuing. For each task, participants read the task, and before searching, I asked them to describe what metadata they would need to solve the problem. While I did not explicitly tell them if the required metadata were correct, I performed this exercise to ensure that the question was read and understood in its entirety before they engaged with the interface. Next, I instructed participants to freely solve the task via the interface, and to verbally indicate when they had completed the task. Upon this verbal indication, I recorded their answer to later determine correctness. In cases where participants authorized recording, I later calculated the time spent familiarizing, filtering, and validating their answers (n=4); otherwise, I manually live-transcribed such behaviors (n=2). I told participants that they had a hard time limit of 6 minutes to complete each task, but no user hit this maximum.

At the conclusion of each task, I surveyed participants in order to understand their confidence in solving the problem, and the perceived value of both the navigation task and the metadata. Participants were asked to enumerate responses on a 1–5 Likert scale. At no point during the study were participants told whether they correctly or incorrectly answered a question, except in the case of the null question (to avoid negative perception due to intentionally-excluded metadata, as participants were not told in advance that there would be such questions). I concealed this information to simulate real-world research: there is no system in place to alert participants of research repositories that they have successfully discovered or found all relevant files.

Finally, I surveyed participants at the end of the study on their navigation preferences, the usefulness of the metadata in general, the usefulness of the metadata in the context of these particular questions, and the relative benefit of the metadata compared to their existing methods of conducting research tasks. Throughout the study, I encouraged participants to provide qualitative feedback on the metadata, questions, and ideas for future use (i.e., after


[redacted] Portal Framework Logout [redacted]

[redacted] Portal Framework / Search

About Search

*

Creator

qzhang	230545
stephenson	154761
lurio	106487
richards	96513
sheyfer	87349
comm	85074
dufresne	69962
sinha	54860
rinaldi	47109
chung	40022
senses	39335
leheny	21157
samsri	20101
zjiang	17248
russell	13881
bates	12299
sanat	10630
harden	10472
yuyin	8500
ma	6130

Results

1299232 datasets found Reprocess Datasets

[redacted]

Parent Folder	[redacted]	Publication Year
c[redacted]09	2020-2	2022

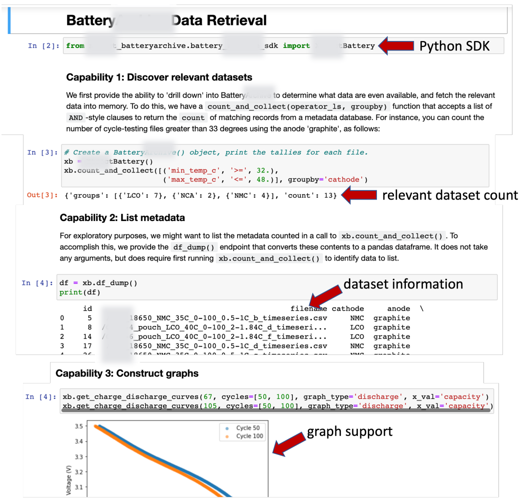
[redacted]

Parent Folder	[redacted]	Publication Year
c[redacted]09	2020-2	2022

[redacted]

Parent Folder	[redacted] Cycle	Publication Year
q[redacted]02c	2020-1	2022

(a) Spectroscopy: Search Portal


Battery[redacted] Data Retrieval

```
In [2]: from [redacted]_batteryarchive.battery[redacted]_sdk import [redacted]Battery
```

Capability 1: Discover relevant datasets

We first provide the ability to 'drill down' into Battery[redacted] to determine what data are even available, and fetch the relevant data into memory. To do this, we have a `count_and_collect(operator_ls, groupby)` function that accepts a list of `AND` -style clauses to return the `count` of matching records from a metadata database. For instance, you can count the number of cycle-testing files greater than 33 degrees using the anode 'graphite', as follows:

```
In [3]: # Create a Battery[redacted]() object, print the tallies for each file.
        xb = [redacted]Battery()
        xb.count_and_collect([('min_temp_c', '>=', 32.),
                              ('max_temp_c', '<=', 48.)], groupby='cathode')
Out[3]: {'groups': [{'LCO': 7}, {'NCA': 2}, {'NMC': 4}], 'count': 13}
```

Capability 2: List metadata

For exploratory purposes, we might want to list the metadata counted in a call to `xb.count_and_collect()` . To accomplish this, we provide the `df_dump()` endpoint that converts these contents to a pandas dataframe. It does not take any arguments, but does require first running `xb.count_and_collect()` to identify data to list.

```
In [4]: df = xb.df_dump()
        print(df)

     id                                       filename cathode    anode  \
0     5  [redacted]18650_NMC_35C_0-100_0.5-1C_b_timeseries.csv     NMC  graphite
1     8  /[redacted]4_pouch_LCO_40C_0-100_2-1.84C_d_timeseri...     LCO  graphite
2    14  /[redacted]6_pouch_LCO_40C_0-100_2-1.84C_f_timeseri...     LCO  graphite
3    17  [redacted]18650_NMC_35C_0-100_0.5-1C_d_timeseries.csv     NMC  graphite
```

Capability 3: Construct graphs

```
In [4]: xb.get_charge_discharge_curves(67, cycles=[50, 100], graph_type='discharge', x_val='capacity')
        xb.get_charge_discharge_curves(105, cycles=[50, 100], graph_type='discharge', x_val='capacity')
```

(b) Battery: Python SDK on JupyterHub

Figure 6.1: Interfaces used in this study. Note: parts of the interfaces are obscured for participant anonymity.

the study).

6.2.3 *Ethics*

The University of Chicago's Institutional Review Board (IRB) reviewed all protocols. Since the studies were outside of the standard data protocol mechanisms of national laboratories, I received direct written consent from the laboratories' computing facility directors to (1) contact their project PIs and participants, and (2) process the data on their HPC systems (i.e., without transferring their data outside of the laboratory). All participants were above the age of 22 and lived in the USA at the time of their participation. Each participant signed a consent form and provided ongoing verbal consent for each part of the study. To protect participant privacy, all personally identifiable data was anonymized to protect the identity of laboratories, projects, participants, and sensitive attributes of the data. Access to data was explicitly (and temporarily) granted by PIs to me and the Xtract service via Globus Auth. Interfaces containing sensitive metadata were also protected by Globus Auth and were brought offline at the completion of each session. Participants were not financially compensated for their participation in this study, but were given access to any requested artifacts of the study (search indexes, metadata, and extractors).

6.3 Evaluation

In this section, I describe the results for both Part 1 and Part 2.

6.3.1 Part 1

In structured interviews with participants, I attempted to characterize each participant's data navigation needs on their given repository. I began by discussing participants' roles in the project and quantified the frequency of their interactions with the data. As shown in Table 6.3, users had varied backgrounds, with each project having 1 domain-, 1 computational-, and 1 computer scientist participant. Participants, on average, read 7 times and wrote 4 times per month, and each participant had at least one year of experience using the repository.

I next attempted to uncover specific data navigation tasks required by participants to conduct research activities. I first asked the questions "*How do you see* {*yourself — others*} *interacting with these data.*" I found that participants tended to perform the common navigation tasks [12] of finding files known to be in their repository (i.e., retrieval) and discovering whether certain classes of data exist. In addition to their current capabilities, I also asked participants to list any metadata that would improve navigation; I summarize these metadata with context in Table 6.2. Interestingly, I found that participants wanted metadata to bolster existing data navigation abilities in a few ways. I noticed common themes between both projects; participants desired metadata that enabled them to navigate more (and larger) files, filter on additional attributes, graphically explore data, and even compare previously incongruous data formats.

In the remainder of this section, I describe in further detail the Part 1 conversations had

Table 6.2: Areas of improvement where metadata can aid user navigation, as depicted by study participants in Part 1.

Improvement	Participants	Description	Participant Statement
Add attributes	Bat-1,2,3 Spec-1,2,3	Provide additional information by which users can query and filter.	"I think there's a field in the data file that explains what type detector it was actually collected in. So basically, I just need to update our script to account for that...and rerun over all the [datasets]" (Spec-1)
Increase volume	Bat-2,3 Spec-1,2,3	Enable search over files that are too large or abundant for existing extraction methods to process.	"Yeah nobody has gone back and processed the older data yet with [spectroscopy data extraction tool] yet. I already know it's going to take a very long time." (Spec-2)
Support quality assurance	Bat-2,3 Spec-1,2,3	Extract metadata that enable users to evaluate experimental outputs.	"If I see something that the users should focus on, then I can quickly point to them and say 'hey this data looks interesting; by the way, this data seems like there's some potential issues that you should try to resolve." (Spec-3)
Facilitate discovery	Bat-1,2,3 Spec-3	Extract and structure metadata that enable users to flexibly 'browse' dataset items.	"We try to go around to all the resources I can to find data that are similar enough for us to be able to treat with the same analytical technique" (Bat-1)
Enable graphing	Bat-2,3 Spec-3	Extract 2D and 3D graph metadata, and expose to users visualization and comparison functionalities.	"Visualizing charge and discharge curves will add value... Users should be able to pull out quantities for specific parts of a voltage curve." (Bat-2)
Extract keywords	Bat-3 Spec-3	Extract metadata that enable users to flexibly search over file keywords, regardless of format.	"We want users to be able to use a blank search box where even those unfamiliar with batteries can interact with the archive." (Bat-3)
Convert formats	Bat-1,2	Create and extract metadata from data formats more-conducive to comparison or visualization.	"There are different sample rates for different datasets. Solution could be—I will 'keep' original test data and create other normalized or 'downsampled' timeseries." (Bat-2)

Table 6.3: The six user study participants.

		Years	**Monthly**	
ID	**Scientist type**	**with repo**	**Reads**	**Writes**
spec1	Computer	3	3-5	8-10
spec2	Computational	1	1	4-10
spec3	Domain	2.5	30	0
bat1	Computational	2	2	0
bat2	Computer	2	1	2
bat3	Domain	2	2	1-4

with participants from both the spectroscopy and battery modeling projects.

Spectroscopy

The spectroscopy repository contained over a decade's worth of correlation spectroscopy data generated by multiple science groups using a handful of detector instruments. All experiments were overseen by the group leader and domain scientist, Spec-3, who frequently meets with users of the facility (mainly graduate students and research scientists), discusses the experimental protocols, and aids them in both using the detectors and verifying results. Each science run produces a number of large grayscale images; HDF files containing both experimental conditions (e.g., temperature) and observed detector behaviors; and configuration and error files generated by the instrument. Participant Spec-1 is a computer scientist who constructed and implemented an automated workflow system to move data from a scientific instrument to persistent storage, execute analysis and metadata extraction scripts, and then push the metadata to a search index. Spec-2 is a computational scientist with a Ph.D. in a similar scientific domain who developed the aforementioned extraction scripts. Spec-3 is a domain scientist (and facility leader) who helps users of the facility configure, execute, and interpret the results of experiments by directly interacting with the search index.

The spectroscopy participants pointed to a number of unique data navigation needs. Spec-1 and Spec-2 had largely overlapping sets of needs. Both mentioned the importance of testing their automated workflow on various formats of data: “So my use cases are generally entirely driven by selecting subsets of data to push through our analysis pipeline. I’m not concerned with finding things at the right temperature; I’m concerned with finding things in a selected format.” The formats in question largely related to the make and model of the detector, as the two main detector types vary in structure, size, and processing requirements. Because the detector and its configuration change between trimesters and creators (users of the detector), it was also important that Spec-1 and Spec-2 could filter on attributes of date and creator. Spec-2 mentioned using the existence of data in persistent storage to measure the quality of their workflow: “if data make it to the [storage], I know there’s no catastrophic failure.” Spec-3 primarily wanted to find whether an experiment was (1) likely to be successful, and (2) calibrated correctly: “We run some standard samples that are basically calibrating the [scientific instrument] to make sure that it is functioning normally. Like you take your car for a test drive, right? So I run through some samples and I say ‘okay, this looks good—cow looks like a cow, goat looks like a goat.’ and then we...hand off the experiment to users so they can start collecting data.” To check the success probability, Spec-3 specifically looked for past data matching similar experimental constraints: “I need to know that this is a polymer. I need to know this is a tungsten metal or a silver metal, and I need to know the temperature at which it has been collected.”

Participants provided a long list of unaddressed navigation tasks that they would like to perform, if able. Foremost, participants expressed the desire for a breadth of users to be able to search on ‘any’ attribute tags, including those that may be inscribed within the filename. Everyone mentioned new metadata attributes that they did not (or could not) extract at that time, including instrument type (Spec-1), dates before 2018 (Spec-2), and keywords (Spec-3). Additionally, participants wanted to extract metadata from more datasets, particularly those preceding their automated ingestion capabilities. To illustrate the scale of this problem, I note that the spectroscopy project had 1.3 million total data sets, but had only indexed 3,000 (0.02%) at that point. Spec-3 specifically mentioned that their prior modes of navigation were compute constrained, so that metadata must thus serve as a trustworthy proxy for actual data elements: “I don’t have the luxury of keeping 10 years worth of data on my computer.” Finally, Spec-3 outlined the need to create smaller thumbnail representations of each raw image.

Participants suggested various software tools that my research team could use in constructing extractors. Spec-1 and Spec-2 pointed to their internal toolkit for extracting metadata and discussed how I could augment it to mine new attributes. Spec-3 provided context on filename nomenclature patterns so that I could mine metadata from there as well. Finally, Spec-2 shared codes for calculating thumbnail representations of images.

Battery Modeling

The battery modeling repository contained battery testing records from five separate institutions. These records listed values about a battery over time, such as the temperature of the cell, the voltage it produces, or the rate of charge that is coming out of the system. The data were predominantly in a tabular (row-column) time-series format, where consecutive rows correspond to measurements at consecutive time steps. Additionally, there were tabular summaries of the testing data over time. In terms of raw data, there were approximately 400 files spanning 500GB, but according to Bat-3, both the file count and size were expected to soon exceed multiple terabytes. Bat-1 is a computational scientist and a frequent user of the repository, Bat-2 is a computational scientist who builds and manages nearly all computational tooling, and Bat-3 is an energy scientist who performs many roles, but importantly solicits data contributions from institutions, ingests these data into their archive, and uses the data to conduct battery experiments.

Given the variety of roles, participants in this project had highly individualized navigation behaviors. Bat-1 primarily wanted to navigate these data as a catalog for machine learning training sets, and periodically scanned the repository for new data: "we try to go around to all the resources I can to find data that are similar enough for us to be able to treat them with the same analytical technique...See if they've got anything I don't already have." Bat-2 was concerned with extracting metadata from each file that was deemed to be interesting for users. Bat-2 mentioned specific challenges processing heterogeneous data sets from multiple groups that barred this process from being automated: "there is too much variability in data sets and file formats to trust others to do this. Even similar [battery] testers have formats that are slightly different." Finally, Bat-3, as a battery expert, wanted to curate an archive that people were comfortable querying as the data were documented (and FAIR) relative to battery data standards outlined by leading battery publishers [120, 118].

Participants mentioned several unmet needs in performing navigation tasks. As in the spectroscopy project, all participants outlined metadata attributes on which they wanted to filter, but their indexes at the time did not support: Bat-1 wished to "isolate all cycle data matching certain aging requirements," Bat-2 wanted graphical representations of multiple curves, and Bat-3 wanted to enable flexible, free-text search over all files. Like the spectroscopy project, participants cited issues of scale: for example, Bat-3 described that they could not process "very large" (>1GB) files on their current compute infrastructure. Multiple participants called for making the repository more accessible to other users. For instance, Bat-3 wanted users to be able to "compare and validate data by comparing graphs of their underlying curves." Finally, participants desired metadata that were simply different representations of the files. Bat-2 wanted a system that could automatically use Bernoulli downsampling to create versions of data sets with standardized measurement intervals for analysis and make these available to the visualization tools.

Participants recommended tools that could be used to construct an extractor for additional metadata. Bat-1 had created a parsing toolkit for cycling data that they suggested I could use to mine other metadata attributes. Bat-2 and Bat-3 proposed the use of the

BEEP toolkit [56] to find graphical metadata (e.g., voltage curves) about each data set.

6.3.2 Part 2

I next discuss participants' interactions with automatically extracted metadata. First, I describe the metadata extraction process from extractor construction to the performance of extraction jobs. Second, I detail the behaviors and perceptions of users when interactively conducting simulated data navigation tasks with a given interface.

Preparation

In order to create research tasks and a navigation interface for participants, I first created two new extractors—one for each project. For the spectroscopy project, the extractor included two key components: (1) a filename parser that isolated attributes embedded by users into the filename, and (2) enhanced versions of their existing scripts that mined new, relevant information from HDF files. I next extracted metadata from all 1.3 million data sets generated between 2018 and 2021. Extracting all records took just 5 hours using 4 compute nodes of their lab's supercomputer. I loaded the extracted metadata into an Elasticsearch index, and configured the connected portal to display facets for commonly mentioned attributes in Part 1: creator, date, temperature, and detector type. The search portal—whose source code was provided by the spectroscopy participants—enabled filtering on facets or providing free-text queries into the search box. Once participants conducted a query, they were presented with a scrolling view of clickable result summaries (showing the filename, creator, and trimester created). Participants could click a result summary to see *all* metadata associated with a data set.

For the battery modeling project, I created an extractor capable of handling both visual (graph) representations of data as well as the classical numeric and string-based attributes. I started with Bat-1's personal scripts that mapped battery testing data from multiple sources into a unified, parsable format. Next, I used the BEEP toolkit to create two separate Bernoulli-downsampled representations of the voltage and discharge axes. I saved these outputs as Python pickle objects that could then be accessible to the graphing components of the interface. From the downsampled representations, I also computed all sorts of metadata attributes, including ranges of other attributes, various rates of change at the start and end of the experiments, and both calendar and cycle representations of the experiment length. Extracting metadata from all 1200 files (400 original; 800 downsampled representations) took just 1 hour using 1 node of the lab's supercomputer. I pushed the metadata to an Elasticsearch index, and mounted the battery data files to the underlying machine serving their JupyterHub navigation interface. The navigation interface, unlike in the spectroscopy project, was created entirely from scratch. The notebook contained both documentation listing all metadata attributes and operators, and runnable example cells for the *query*, *count*, *dump*, and *graph* functionalities.

Interactive exercises

I asked participants to perform 7 to 9 navigation tasks that simulated the desired navigation tasks outlined in Part 1. I then questioned them about their experience, both after each task and at the end of the survey. I outline these responses, as well as participant performance metrics, in the following.

User confidence in results. After each task, I asked participants to rate on a 1–5 Likert scale (where 1 is *strong disagreement* and 5 is *strong agreement*) whether they were confident that they had successfully completed the task. As illustrated in Figure 6.2a, participants were 100% confident (Likert score of 4 or 5) in their ability to successfully accomplish easy, medium, and hard tasks. For their "own" derived tasks, participants expressed confidence in completing 70% of tasks. In the controlled 'null' metadata scenario, participants expressed

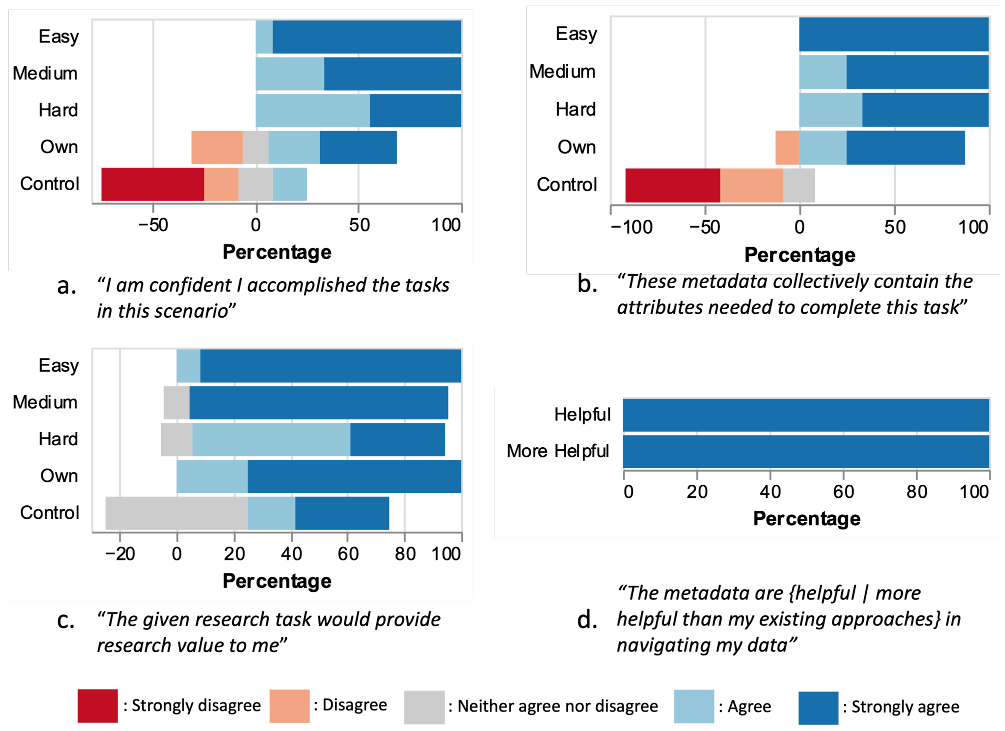


Figure 6.2: Confidence and value measurements (1–5 Likert Scale) across all participants and question difficulties.

confidence in completing just 10% of tasks.

Answer correctness. After the study, I validated participant results against the list of correct results. Table 6.4 shows the per-task correctness for each participant. No participant got more than 1 wrong, and on average, correctly answered 89% of questions (s.d. 5.5%). Interestingly, all participants capably identified the control scenario where I omitted the metadata necessary to resolve the task.

Time taken to result. Participants quickly solved the tasks when compared to their best alternative methods. Such methods included manually downloading and parsing files for necessary information, using an existing tool as is, using an edited version of an existing tool, or designing and using an entirely new tool. I illustrate each participant's task com-

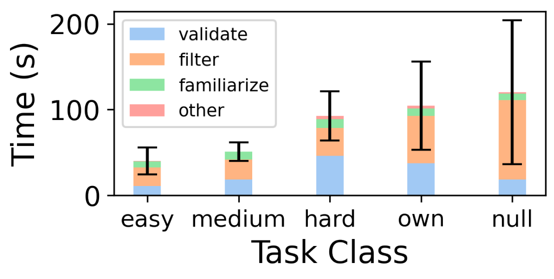
ID	T1	T2	T3	T4	N5	T6	T7	T8	T9
Spec-1	✓	✓	✓	✓	✓	✓	✓	✓	✓
Spec-2	✓	✓	✓	✓	✓	✓	✓	✓	✗
Spec-3	✓	✓	✓	✓	✓	✗	✓	✓	-
Bat-1	✓	✓	✓	✓	✓	✗	✓	-	-
Bat-2	✓	✓	✗	✓	✓	✓	✓	-	-
Bat-3	✓	✓	✓	✓	✓	✓	✗	-	-

Table 6.4: Display of task completion correctness across all participants and questions. Question **N5** represents the null-metadata control question. Accuracy: 89%; Standard Deviation: 5.5%.

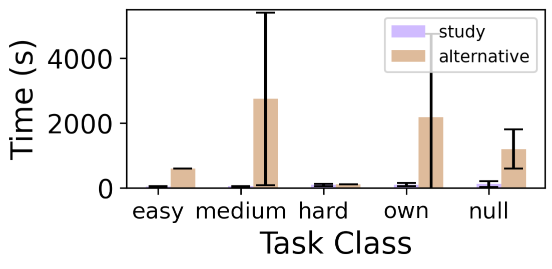
pletion times in Figure 6.3. As shown in Figures 6.3a and 6.3c, participants from both projects finished all problem classes in under 2 minutes, on average. I observed that participants spent a majority of their time directly interacting with the metadata between filtering (crafting or editing queries over metadata attributes) and validating (parsing through the metadata of query results to determine correctness and/or next steps). When able, I added each participant's self-projected time to finish the task via their "best" alternative method in Figures 6.3b and 6.3d. In 6.3b, the alternative method time for hard tasks is only 20% slower than in the study; this is due to two reasons: (1) there is only one hard task solvable by one battery participant (Bat-2), and (2) Bat-2 mentioned they had a tool to solve that particular task. Overall, I see a 1.2–50× navigation time reduction across all task classes.

Perceived User utility. As a baseline for the other measurements, I first needed to know whether participants believed solving the simulated research tasks correlated to real-world research value. As shown in Figure 6.2b, participants across 96% of non-null questions believed that the ability to solve the navigation task would, if done in the context of their research, add value.

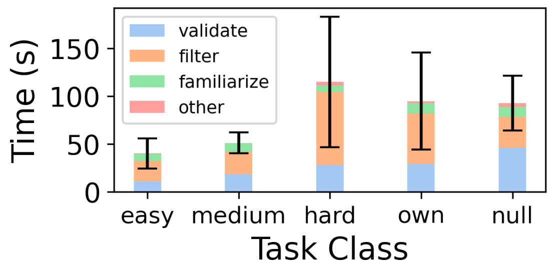
I next evaluated whether participants gained utility from the metadata themselves. As illustrated in Figure 6.2c, participants believed that the metadata contained attributes necessary to solve tasks in all easy, medium, and hard tasks, and nearly all cases on their own



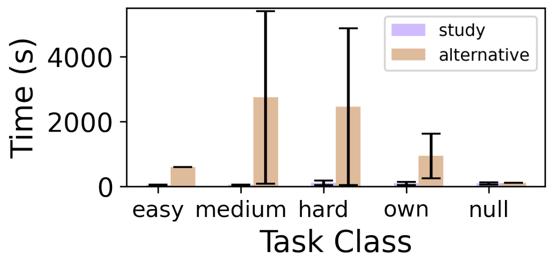
(a) Battery: Study-Only



(b) Battery: Study + Alternative



(c) Spectroscopy: Study-Only



(d) Spectroscopy: Study + Alternative

Figure 6.3: (Left) Time taken to perform navigation tasks. I additionally show total time spent validating, filtering, familiarizing, and other. (Right) Comparison of time to complete task in study, and self-reported time using best alternative.

tasks.

I found that the automatically extracted metadata, as given, enabled participants to perform more navigation tasks than their existing approaches. As shown in Table 6.5, I observed that 40% of tasks could simply not be completed via participants' alternative interfaces. In cases where users *could* alternatively perform the task, I asked (1) how they would do it, and (2) how long completing the task would take. I illustrate the encoded versions of such alternative methods in Figure 6.4. A plurality of problems would be solved by some form of manual scan through files (i.e., parsing through all data files individually). Other tasks could be solved by editing an existing tool (e.g., to provide scalability or new metadata attributes), using an existing tool in its current state, or creating an entirely new tool. (In the last case, the user stated, "I would build my own metadata extraction system, extractors, and Notebook!")

At the conclusion of the survey, I asked participants to state whether the metadata were

ID	**T1**	**T2**	**T3**	**T4**	**N5**	**T6**	**T7**	**T8**	**T9**
Spec-1	✓	✓	×	✓	×	✓	✓	✓	✓
Spec-2	×	×	✓	✓	✓	✓	✓	✓	×
Spec-3	×	×	✓	×	×	×	✓	✓	-
Bat-1	✓	✓	×	×	✓	×	✓	-	-
Bat-2	✓	×	×	✓	×	×	✓	-	-
Bat-3	✓	×	×	✓	✓	×	✓	-	-

Table 6.5: Display of whether or not each participant could perform each task using alternative methods.

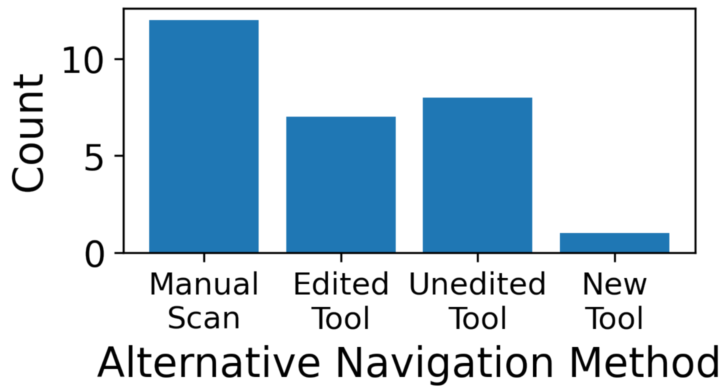


Figure 6.4: Alternative Methods Counts. How do participants claim they would solve the navigation tasks outside the context of this study? Respondents mentioned methods where they manually download and parse individual files (Manual Scan), use an existing script or tool (Unedited Tool), use an edited version of a script or tool (Edited Tool), or develop an entirely new tool (New Tool).

helpful in navigating their data, and further, whether the metadata were *more helpful* than their alternative approaches in navigating data. As shown in Figure 6.2d, all participants strongly agreed on both counts.

Additional feedback. I collected unstructured feedback throughout Part 2 from both participants and my own observations. I identified both unexpected behaviors and areas for improvement. In terms of unexpected behaviors, I found that some participants could more-quickly answer some questions due to prior knowledge of the repositories. When asked to count all data sets containing "silicon" Spec-3 immediately filtered on creator based on their prior knowledge of the data's origin: "I kind of know too much detail; I remember there were some issues with their experiment. I know the people who studied silicon." When asked to graphically compare datasets from two separate science groups, Bat-3 correctly commented "I'll go through the steps, but these experiments were conducted under completely different conditions. So I can't see how they wouldn't be different."

Participants noticed minor discrepancies, due either to the data or imperfections in the automated tooling, in the software toolkit and extractor. In the spectroscopy metadata, participants found that some non-study users had accidentally flipped fields in their scripts to create HDF files. Bat-3 commented, "there are some things that are [dates] that aren't [dates]. But people put it in the [date] part of the metadata." Bat-1, while exploring their own navigation task, commented about the graphical representations of downsampled voltage data, "it seems that there are some screw-up points. Perhaps the data are out of order?" Bat-3 also noticed my use of 'minimum' and 'maximum' temperature was not as scientifically searchable as possible: "it's probably fine, but if you only take the single highest or lowest outlier, it could just be detecting a blip or mistake from the tester." Bat-3 then recommended taking an average of multiple maxima and minima. I will consider each of these points in future study of these projects.

6.3.3 Limitations

National laboratory storage represents a specific view of science data, and the challenges seen in battery modeling and spectroscopy may not represent research repositories in other disciplines. Additionally, having participants navigate tasks subject to my own crafted scenarios may not perfectly simulate the full scope of participants' research tasks. Participants' self-reported perceptions may not indicate behavior that would manifest outside of these particular tasks. The research scenarios may be biased towards the experiences of participants, rather than to those of 'outsider' users who may be less aware of file contents. Despite my best efforts to avoid measuring the influence of user interfaces on file navigation, some results may change based on participants' perceptions of, and interactions with, their provided interface. Furthermore, my definition of navigation does not directly measure participants' ability to organize their files, which should be studied in future work.

6.4 Discussion and Conclusion

In this study, I explored the influence of automatically extracted metadata on users' ability to navigate science repositories. I found that automatically extracted metadata, regardless of the interface used, has positive net effects for users; they can confidently, correctly, and quickly navigate repositories and can solve problems that are currently either cumbersome or otherwise impossible. A secondary outcome from this work is demonstrating the importance of automated extraction systems for science data.

A key finding from this study is that the automatically extracted metadata enable users to not only perform potential tasks more quickly than via alternative methods, but also perform a wider range of such tasks. In particular, the scale of the repository and the individual files therein made manual annotation infeasible. The battery modeling project previously had been unable to process large files and the spectroscopy project had long desired to index the remaining 99% of files in their repository, but simply had not been able to do so despite the demonstrated benefits of indexing. Furthermore, given the rapidity of research and the constant pressure to 'produce', there is a direct productivity benefit to faster navigation.

Overall, participants were satisfied with the quality of the metadata from both general metadata extractors and from those created specifically for their use case. This is especially important as automation, when done poorly, may not encompass the nuanced perspectives of human users. Participants across science domains and research backgrounds stated that not only were the metadata helpful in navigating their data, but were a stark improvement over their alternative navigation approaches. This in itself is a strong testimony for the benefits of automatically extracted science metadata.

I envision several ways to build upon both the study of metadata utility and system performance. First, this study examined regular users of repositories; in reality, collection curators often wish to make their collections navigable by new users, either for collaboration or for the overall benefit to the discipline [124]. To measure whether existing metadata elicit utility to new users, one could simply take the metadata from this study and determine whether new users experience the same net benefit. Additionally, in future work I will continue to measure user experience across additional sciences and repository compositions.

Finally, I plan to improve the metadata in multiple ways based on user feedback from this study. I will fix minor errors noticed by users, particularly around data interpolation and graphical representation of data. Furthermore, I will explore methods for including zero-count metadata to more clearly represent via the interface that a given search result does not exist, which could be propagated to a given interface as a facet. These small changes would address the majority of the issues experienced by users in the study.

CHAPTER 7
XTRACT: THE METADATA EXTRACTION SYSTEM FOR SCIENCE

In this thesis, I have described the Xtract metadata extraction system, how I have optimized it to suit science data, and an evaluation as to how users interact with the metadata it produces. This chapter focuses on the implementation details of Xtract, or simply how users can *use* it to better navigate their science data. In §7.1, I discuss Xtract's user interface requirements, which position it to suit a diverse community of science users. Afterwards, the tone of the chapter changes the focus from "theory" to "practice" as I discuss exactly how users can configure endpoints at their compute facilities (§7.2), create and submit their own extractors to Xtract (§7.3), and use a simple Python interface to submit and monitor jobs (§7.4). Finally, in §7.5, I present concluding remarks and future work.

7.1 User Requirements

In this section, I discuss how Xtract's user interface enables a breadth of science users, from system administrators to domain scientists to students, to leverage its powerful metadata extraction capabilities. In the following, I outline each requirement and provide a high level description as to how that requirement manifests within Xtract:

1. **High Level Programming Model.** With low technical barriers to entry, Xtract could potentially grow to become an extraction system used in myriad research organizations. I designed Xtract to be broadly usable by computational scientists who can capably leverage simple software libraries, but who may not possess deep knowledge of, or experience with, underlying computing systems. Therefore, Xtract needs a high-level programming model to abstract resource scaling infrastructure and any job-specific logic like crawling, extracting, or metadata offloading. To this end, I have

constructed a Python software development kit (SDK) that wraps vital Xtract functionalities in the ubiquitous Python syntax. SDKs facilitate accelerated adoption of APIs and systems [49], and the Python scripting language has the largest user base of all programming languages, among computational data scientists.[1] Furthermore, users should be able to, with minimal effort, design and build extractors using scripts or libraries from their research workflows. To this end, Xtract provides a `base_extractor` interface and an example container configuration such that users should be able to quickly and easily convert familiar research codes into functional metadata extractors.

2. **Simple System Configuration.** While the long-term goal of Xtract is to have system administrators manage Xtract endpoints, users must currently configure their own endpoints. Having simple configuration mechanisms is vital, as many data scientists are not necessarily familiar with low-level machine utilities. Command-line interfaces (CLI), historically, have successfully facilitated easy remote endpoint configuration in other projects, such as funcX and Globus. I borrow the funcX and Globus configuration models by exposing a pip-installable `xtract-cli` that provides utilities for users to configure and register Xtract endpoints, fetch containers, and test the liveness of various capabilities (i.e., funcX, Globus, and Singularity). Additionally, I provide Xtract documentation with detailed instructions for configuring the compute and data layers, including pointers to relevant documentation on Globus, funcX, Conda, and Singularity.

3. **Customizable**. Science data and users' metadata requirements vary significantly between scientists and projects. To address the individuality of users' needs, Xtract contains utilities to support the creation, submission, and workflow integration of custom metadata extractors. To this end, I have constructed an internal `xtract_sdk.agent`

1. According to a 2018 survey of 24,000 data scientists by Kaggle [83], 83% of data scientists regularly use Python

(i.e., not directly used by users) that can turn any Python script into an extractor, so long as that script adheres to non-restrictive naming and type requirements. Users package their extractor into a container, and submit (and later fetch) their extractors by using the `xtract-cli`. Xtract does not yet automatically integrate custom extractors in the file type identification models, so like other extraction systems, new extractors are invoked based on users' libmagic matching rules.[2]

4. **Secure.** Security is of utmost importance, as Xtract can be run on sensitive data at access-conscious compute facilities. Xtract's security model closely mirrors that of the underlying funcX federated FaaS infrastructure [14]. Xtract uses Globus Auth [129] for authentication, authorization, and protection of its API. The Xtract service is registered as a *resource server*, allowing users to authenticate using a supported Globus Auth identity (e.g., institution, Google, ORCID) and enabling various OAuth-based authentication flows (e.g., native client) for different scenarios. Xtract uses the associated funcX Globus Auth scopes (e.g., "urn:globus:auth:scope:funcx:register_function") via which other clients (e.g., applications and services) may obtain authorization for programmatic access. Xtract endpoints are themselves Globus Auth native clients, each dependent on the funcX scopes, which are used to securely connect to the funcX and Xtract services. Xtract endpoints require the administrator to authenticate prior to registration to acquire access tokens used for constructing API requests. Extractor execution is isolated in containers to ensure that functions cannot access data or devices outside their context. To enable fine-grained tracking, I store execution request histories in the Xtract service and funcX stores function invocation logs at endpoints.

2. Automatically integrating new extractors into FTI models is not too far off; the minimum viable functionality for this would be to simply retrain the FTI model using additional features automatically-generated from the new extractor. Advanced functionality, as I outline in Chapter 8, could instead leverage active learning models that suit ever-changing data input streams.

7.2 Xtract endpoints

Xtract endpoints enable compute facilities to be both findable and usable by the Xtract service. To provide extraction capabilities, each Xtract endpoint must have a *data fabric* that enables Xtract to find, crawl, access, and transfer (when necessary) files. Additionally, at least one Xtract endpoint per extraction job should have a *compute fabric* that enables the Xtract service to remotely invoke extractors on the files in the data fabric.

7.2.1 Compute Fabric Setup

The compute fabric, discussed in detail in Chapter 5, contains a funcX endpoint and a series of software containers encompassing the extractor library. Users perform the following steps to configure their compute fabric: install funcX, create a funcX endpoint, and configure it to leverage their compute resource (i.e., supercomputer, laptop, or cluster). To test that the funcX endpoint is functional in Xtract, the `xtract-cli` lets users (1) fetch the desired containers,[3] and (2) test whether Xtract can invoke extractors on the funcX endpoints, or if changes must first occur around funcX endpoint liveness, container dependencies, or file system permissions. Users need not register the funcX endpoint with Xtract; they provide the funcX ID on the client side at runtime.

At the time of this writing, Xtract supports fetching three classes of container: `all` fetches every container accessible by a user, `tika` fetches only the container encapsulating the Apache Tika web service, and `custom` enables users to manually fetch all custom containers registered to their user. Assuming a properly configured compute fabric, the commands for fetching and testing containers are the following:

1. Fetch containers (in this case, *all* containers) by using the Xtract CLI:[4]

3. at the time of writing, Xtract only supports the transfer of custom Singularity containers; Docker and Shifter can be used by manually pulling them to the endpoint.

4. if this is your first time using the Xtract CLI, you will be prompted to login to Globus Auth.

```
$ xtract-cli fetch containers --all
```

2. Test that the compute fabric is functional by providing the funcX endpoint ID to the CLI. This will either return that the compute fabric is working as intended, or print debugging steps to the console:

   ```
   $ xtract-cli test compute --funcx <fx_eid>
   ```

At this point, the compute fabric is configured and Xtract can invoke extractors at this compute resource. If container updates are required for any extractor, they can be subsequently fetched by again calling `xtract-cli fetch ...`, which will pull all new or updated containers.

7.2.2 Data Fabric Setup

The data fabric consists of a Globus endpoint configured atop a user-accessible file system or storage service (e.g., Google Drive or AWS S3 Bucket). While the `xtract-cli` documentation provides details for installing and configuring a Globus Connect Personal endpoint from the command line, many institutional computing clusters contain preconfigured, multi-tenant endpoints. Once the Globus endpoint is configured, users can validate endpoint liveness and access rules by executing the following command in the `xtract-cli` utility:

```
$ xtract-cli test data --globus <globus_eid> --stage_dir <path_on_machine>
```

This will either return that the endpoint is functional (and that the staging directory is writable via Globus Transfer), or provide the necessary debugging steps in case either test fails. At this point, if both the compute and data fabric are functional, users can perform metadata extraction workflows on their machine by using Xtract via the `xtract-sdk` library.

7.3 Custom Xtract Extractors

As shown in the navigability user study results in Chapter 7, users have diverse metadata requirements requiring specialized extractors built specifically for their data collections. To this end, I have designed a schema and tooling for users to create custom extractors in Python. To reiterate the discussion of Chapter 3, an extractor is two parts: a (i) function that runs inside a (ii) container. To promote ease of use, all extractor functions by default inherit a common `BaseExtractor` class, illustrated in Listing 3, that receives all information about the extraction. The `BaseExtractor` creates an `XtractAgent` responsible for inputting extraction information and FamilyBatches (whose structure is discussed in Chapter 5), executing the extractor, handling extractor errors, and writing metadata and task information (e.g., execution time) to disk. To perform an extraction, the `XtractAgent` must receive the following event information from the Xtract service:

- `xtract_dir`: path to the Xtract folder (on the Xtract endpoint) containing configuration information and the container library (by default, this is created at `/<home>/.xtract` when `xtract-cli` is installed).
- `sys_path_add`: list of directories to be added to the system path (for complex dependencies such as Apache Tika or Tensorflow).
- `module_path`: path to the custom Python extractor script (i.e., scientists' custom scripts that parse particular attributes from files).
- `recursion_depth`: maximum Python call stack depth allowed by any extractor (default: 5000). This helps to protect against stack overflows, and thus, worker failures.
- `metadata_write_path`: the path at which to write the extracted metadata objects

Users extend the `BaseExtractor` class in their own script outlined in the `module_path` event argument. All extractor-specific logic is put into Python scripts (or Python wrappers for non-Python programs) that are automatically imported by the `XtractAgent`. These scripts need only adhere to two principles—to have an `execute_extractor` function that inputs a list of file system paths and outputs a dictionary (of metadata) and to package its own errors into the metadata dictionary. Additionally, users create containers by simply augmenting the sample container provided in the Xtract documentation with any additional libraries and files. At this point, the custom extractor should be wrapped into a single container to be loaded into the Xtract ecosystem. Users load their container image file into the Xtract ecosystem via the following command:

```
$ xtract-cli push container --path <path_to_image>
```

Once the container is pushed, an extractor ID is printed to the console. This ID can subsequently be used in the `xtract_sdk` to integrate the extractor into the metadata extraction workflow.

```python
def base_extractor(event):
    from xtract_sdk.agent.xtract import XtractAgent

    # Load endpoint configuration. Init the XtractAgent.
    xtra = XtractAgent(xtract_dir=event['xtract_dir'],
                       sys_path_add=event['sys_path_add'],
                       module_path=event['module_path'],
                       recursion_depth=event['recursion_limit'],
                       metadata_write_path=event['metadata_write_path'])

    # Execute the extractor on the family_batch.
    xtra.execute_extractions(family_batch=event['fam_batch'], input_type=event['type'])

    # All metadata are held in XtractAgent's memory. Flush to disk!
    paths = xtra.flush_metadata_to_files(writer=event['writer'])
    stats = xtra.get_completion_stats()
    stats['mdata_paths'] = paths

    return stats
```

Listing 3: BaseExtractor and underlying XtractAgent

7.4 The Xtract SDK

To enable users to easily and programmatically use Xtract, I designed the Python `xtract_sdk`, which handles all user interactions with the Xtract service. The `xtract_sdk` contains utilities for authentication, running and monitoring crawls, running and monitoring extractions, and performing actions on the generated metadata. I illustrate example `xtract_sdk` usage in Listing 4, and the remainder of this section explains each functionality in more detail:

1. **XtractClient.** The XtractClient handles authentication, job submission, and task monitoring. The first time users instantiate an XtractClient (or when one's authentication tokens expire), they are prompted with a link to a login flow. Users can add custom Globus Auth scopes to the `auth_scopes`; this is especially useful in the context of large computing facilities that have strict authorization requirements for storage and compute resources. The XtractClient is created once and can be persistently used until the authentication tokens expire. In the remainder of this chapter, I refer to this client as `xtr` for brevity.

2. **XtractEndpoint.** XtractEndpoint objects programmatically represent all endpoints (as discussed in 7.2) to be used as part of a metadata extraction job. Users must denote, at minimum, one endpoint to be used for an extraction job. Users denote the data fabric by signifying a type (e.g., "*GLOBUS*") and its accompanying ID, the directories to be processed by the extraction job, optional funcX endpoint IDs, and an optional directory to write metadata.

3. **Crawling** A user starts their end-to-end metadata extraction job by crawling by each directory denoted in the XtractEndpoint objects. Users pass XtractEndpoint objects into a call to `xtr.crawl()`, and the system returns a `crawl_id` used to track crawling (and later, extraction) progress. Users can monitor crawling status by calling `xtr.get_crawl_status()`.

4. **Container Registration.** Once users crawl files, they may want to "lock in" the available containers to be used for their extraction process. Users do so by calling `xtr.register_containers()`. In future iterations of this work, I plan to add a server-side container service that would replace manual container registration.

5. **Extracting.** Once a crawl job has started (and the crawl_id is issued), users can asynchronously perform an extraction job over all files discovered during the crawl. Despite being the most compute-intensive task, there is relatively little configuration to be had at this point—a user must simply launch the extraction job with `xtr.xtract()` and monitor with `xtr.get_crawl_status()`. Users should only launch one extraction job per crawl.

6. **Offloading Metadata.** Once metadata are extracted from a file, Xtract saves these metadata as JSON documents at the directory outlined in the `XtractEndpoint`. In order to move the metadata to another machine, users can either transfer them manually via their own protocols *or* use an automatically configured Globus Transfer. Users can automatically perform the latter by calling `xtr.offload_metadata()` to push the data to a destination Globus endpoint.

```python
from xtract_sdk.client import XtractClient
from xtract_sdk.endpoint import XtractEndpoint

xtr = XtractClient(auth_scopes=[], dev=False, force_login=True)
xep1 = XtractEndpoint(repo_type="GLOBUS",
                      globus_ep_id='1234567890-abcdefghi',
                      dirs=['/home/user/file/path1', '/home/user/file/path2'],
                      grouper='file_is_group',
                      funcx_ep_id='0987654321-zyxwvutsrq',
                      metadata_directory='/home/user/mdata_here')

xtr.crawl(endpoints=[xep1])
xtr.get_crawl_status(crawl_ids=None)
 # will automatically determine which endpoints to get the status of if none are given
```

```

xtr.register_containers(endpoint=xep1,
                        container_path='/home/user/containers')

xtr.xtract()
xtr.get_xtract_status()

xtr.offload_metadata(dest_ep_id='a0b1c2-d3e4f5-g6h7i8',
                     dest_path='/Documents/mdata/',
                     timeout=600,
                     delete_source=False)
```

Listing 4: Example xtract_sdk script to create an XtractClient and XtractEndpoint, perform crawl and extraction jobs, and offload metadata to a destination resource.

7.5 Future Work

Xtract's user interface enables even casual programmers to leverage advanced metadata extraction capabilities. I plan multiple ambitious changes to enable widespread adoption by a breadth of science communities, from large computing centers to individuals with a laptop and minimal (or no) programming experience. I discuss both in detail in the following.

I plan to add capabilities for system administrators and users of large, multi-tenant research clusters, including HPC. First, I plan to provide isolated multi-tenancy support to Xtract. In the current model, all extraction jobs are added to the same queue; therefore, one user's extraction job can directly affect the performance of another's. To mitigate this, I can separate each extraction plan orchestrator into its own elastically scalable resource, such as a Kubernetes pod [6]. In this scenario, users could use Xtract with stricter quality of service (QoS) constraints. Second, I plan to develop and expose additional scheduling capabilities to users. As I discuss in Chapter 8, I plan to explore hierarchical scheduling algorithms for more effective processing of files, and will expose scheduling options to users. Finally, I will add service-level capabilities for users to automatically train and integrate Xtract's file type identification (FTI) models on new data and extractors. In the current model, this is done by manually training models and adding their paths to Xtract's config file, but the method for obtaining ground-truth data (i.e., executing each extractor on each file in the training and test sets) is too manual.

I will next make Xtract usable by those without significant programming experience by creating an accessible ecosystem of tools for constructing an interface.[5] Such functionality could manifest as a direct manipulation interface [54] in which users could construct extractors (and subsequently execute extraction jobs) by switching between a drag-and-drop style interface for representing data flow between extractor sub-components and a scripting panel (such that updates to one updates the other). The visual elements could be automatically translated into extractor code, thereby removing (or greatly reducing) the programming knowledge required by users.

5. this effort spawns from an attendee question from a 2022 presentation of mine—a computational materials scientist asked *"How much Python programming is necessary to use Xtract? Do you have any plans to make it available in other languages? Or no programming languages at all?"*

CHAPTER 8
FUTURE WORK

In this chapter, I discuss future work as well as some open-ended ideas that emerged from the design, implementation, and evaluation of Xtract. Specifically, I discuss potential research frontiers for the extractor library, system evaluation, science applications, and user experience.

8.1 Extractors

Metadata extraction systems like Xtract depend on rich, evolving libraries of extractors that suit users' needs. While the ad hoc addition of more extractors will broaden Xtract's coverage of science data, additional optimizations at the extractor level are needed to support the needs of both the system and its users. I propose three optimizations: merging dependencies between similar extractors, enabling execution of extractions on specialized hardware (i.e., accelerators), and chaining extractors.

Methods are needed to reduce the container footprint—the overall number and size of containers at each compute facility—within Xtract. As the extractor library continues to grow, each new extractor is accompanied by a container wrapping its dependencies. In the current model, Xtract has one container per extractor, which unfortunately leads to redundant features between containers. This leads to the *container explosion problem* where the number and cumulative size of containers pushes the bounds of the available (or desired) space for storing them. Furthermore, even in the context of similar extractors, the system must cold-start containers when switching between extractors. Existing work [103] addresses the container explosion problem by intelligently constructing the minimum number of containers given dependency requirements.

Cutting-edge methods for mining metadata require specialized hardware such as GPUs and FPGAs to be sufficiently performant. For instance, contemporary models for computer vision and natural language processing leverage specialized hardware for processing. For instance, video encoding processes leverage FPGAs to conduct near real-time analysis [71]. To enable Xtract's awareness of these hardware, Xtract's prefetcher must be augmented with real-time awareness as to the capacity of available hardware components at each compute facility. I plan to build upon existing work in hardware abstractions for user applications [128].

Chaining extractors to run on the same open file object would promote efficient resource usage and faster end-to-end extraction times. Currently, each extractor opens each file just one time, but across m extractors run on a file, this could mean opening the file m times. Furthermore, the statelessness of each extractor means that any useful information is forgotten between extractions—for instance, the location of free-text data identified by a tabular extractor. I look to extend the current extractor scheduler to identify compatible extractor linkages, and to enable direct data flow between them (e.g., using Parsl [2]).

8.2 System

I chose serverless and federated FaaS design principles for Xtract due to its deployment flexibility on diverse cyberinfrastructure and its demonstrated scalability [14]. However, federated FaaS can be sub-optimal due to the inherent communication latency. As a next step, I plan to examine the performance benefits, if any, of pushing the extractor scheduling to the facility containing a file. For instance, the system could leverage many computing facilities' existing Spark [117, 105], Ray [85], Dask [97], or Parsl parallelization capabilities to perform extractions within the confines of a cluster. Furthermore, such a setup could still benefit from a centralized service that can offload files between resources as discussed in Chapter 5.

Next, I plan to explore methods for file type identification model retraining in the context of new data streams. Currently, all models are statically trained on a subset of the data and are used until they are explicitly retrained. Online machine learning techniques [37] allow a given model to update in the context of “concept drift”. Intuitively, science repositories are likely prone to concept drift as different groups write their own style of data to a repository at a given time, and models trained on prior iterations of data may not accurately reflect the requirements of the new data. Exploring online methods for machine learning would enable Xtract to support streaming repository updates in addition to large batch updates. Additionally, as a metadata extraction system grows to encompass many facilities, federated learning approaches become beneficial so as to avoid moving large quantities of file features to a centralized resource for training [99].

8.3 Science Applications

This thesis focuses on one use case of metadata extraction: the creation of a data catalog from bulk files at rest. In future work, I plan to explore online metadata extraction as part of a science workflow. Such functionality would enable real-time digital curation [23], quality assurance [30], and decision-making in research automation [67, 116]. To this end, I plan to engage relevant science use cases with these needs, and to prepare Xtract for use in relevant science workflow systems [26, 15].

Additionally, I plan to directly support machine learning applications with Xtract. Currently, researchers must manually determine each file’s compatibility for model training, and subsequently develop parsers to convert files into features [142]. Xtract’s automated schema determination could quickly identify files on a file system that are compatible with various classes of learning models. Therefore, Xtract could enable researchers to easily explore their training data universe without necessarily opening a file.

8.4 User experience

To enable Xtract's adoption across research communities, I plan to identify pain points via web-enabled chatbots, a user study on extractor creation and use, and a user study on the propensity of "outsiders" to interact with a repository.

Web-enabled chatbots [108] can help users validate the contents of the metadata extraction system. In Chapter 3, I stated that two tenets of effective extractor design are correctness and relevance, where both are validated by human experts. Such a chatbot would likely sit within the interface, as described in Chapter 6. Early versions of a chatbot could simply ask a series of questions that evaluate the effectiveness of existing approaches. More advanced versions could guide user behavior and collect additional context for users' metadata desires.

Even more pain points can be identified (and later resolved) via user study. While the work in this thesis focused on examining the value of metadata objects, I will next examine users' interactions with the extraction system. Constructing extractors, instrumenting endpoints, and configuring extraction jobs are development-heavy tasks rife with potential for pain points. I plan to evaluate the psychological elements of users' interactions with a metadata extraction system that can hamper or enhance their ability to navigate their repositories. For instance, can a search interface create research value by prioritizing potentially "surprising" results that spur innovation? Can I identify common metadata traits that cause users to give up during a search (e.g., *too many irrelevant attributes*)? Finally, the work in this thesis focused on "power" users of repositories—those who have read and written data contents frequently. The next step is to examine whether metadata attributes promote navigability for a user class many publicly-accessible repositories covet: *new users.*

CHAPTER 9
CONCLUSION

Motivated by the long-term goal to enable effortless navigability over broad science data collections, I have explored and evaluated optimizations for automated metadata extraction systems. In this thesis, I packaged my findings in the form of Xtract, a nascent metadata extraction system designed specifically to address the challenges of science data. I outline findings from five separate-but-related research expeditions in the following.

In Chapter 3, I explore low-level methods for constructing extractors that input one or more files and output their descriptive metadata. I find that there are no widely accepted guides for (i) selecting which extractors to include in one's extractor library, or (ii) principles for designing effective extractors. In selecting extractors for a library, each extractor should derive and/or synthesize metadata from the union of all communities' needs, within reason, for a particular file schema. Furthermore, extractors should be designed to simultaneously fulfill the needs of people (i.e., the metadata should be correct and relevant) *and* the machine (i.e., the extractor logic should be amenable to scaling on diverse research computing systems). I show that the Xtract extractor library achieves $>$ 80% coverage—the fraction of files yielding non-negligible metadata from at least one extractor—across four separate real science repositories.

In Chapter 4, I investigate how an extraction system can automatically prioritize extractors that are most likely to return "quality" metadata documents, subject to interchangeable criteria. The first step to extractor prioritization is to accurately assess the relevant extractors for a given file. To assign extractors, I build upon prior work in file type identification (FTI), but subsequently treat the problem as a multilabel, multi-output classification task to accommodate files that can be reasonably processed by more than one extractor. I evaluate these approaches against the FTI tool used by Linux and leading extraction systems: libmagic. I show that while libmagic accurately assigns 65% of science file types, a balanced random forests model trained on the first 512 bytes of each file accurately assigns 88%. Next, given a probability vector for each file, one can compute a "priority" score (I call it *alpha*) meant to prioritize extractors that either return high-yield *or* complete metadata. I find that a schedule optimizing for the expected metadata yield in the CDIAC repository effectively prioritizes 97% of file-extractor mappings that generate high-yield, near-full, uniquely findable, and semantically searchable metadata documents within the first 25% of alpha-ordered invocations.

In Chapter 5, I construct and evaluate an architecture to scalably apply extractors to files, even when files are decentralized across compute facilities. I outline the design decisions for each component of an automated extraction system—Xtract—from the crawler through the extraction plan orchestrator, and investigate the latency of the system's components, both individually and collectively. I show that one can use Xtract to scale to over 2048 HPC workers invoking extractors simultaneously, and that one could process the entire Materials Data Facility in just over 6 hours. I show that a number of optimizations can effectively enhance system performance: batching, offloading, and the min-transfers algorithm for preventing duplicate file transfers. I show that offloading to idle compute resources can improve end-to-end extraction times for both Xtract and Apache Tika. While not necessarily an apples-to-apples comparison due to differences in computing models and extractor libraries, I find that Xtract executes extractions up to 20% faster than Tika across 56 workers on a 5-node research computing cluster.

Finally, in Chapter 6, I investigate whether automatically extracted metadata enable navigability over real science data collections. Specifically, I explore whether these metadata let users quickly, correctly, and confidently solve data navigation tasks that simulate their day-to-day research efforts. I find that Xtract-generated metadata enable users to perform tasks 1.2×–50× faster than self-reported alternative methods, correctly solve 89% of tasks, and report confidence in all non-null tasks. Overall, all users strongly agree that the automatically extracted metadata enable them to better navigate their data than via alternative methods.

I envision that the work presented in this thesis, when combined with additional work in related areas, will lead to the construction of an optimal and cutting-edge metadata extraction system for science. As shown in this thesis, science data swamps are made significantly more navigable when users can interact with automatically extracted metadata from their contents. While my efforts in building Xtract have shown significant progress in many of the areas to be explored for effective and ubiquitous metadata extraction, there is still much work to do.

APPENDIX A
FORMALIZATION: METADATA EXTRACTION

Metadata extraction is a broad term used in different contexts across disciplines. In this section, I attempt to provide a formal definition of *bulk metadata extraction* such that it can be consistently explored both in this thesis and beyond.

A.1 Terminology

I define automated metadata extraction as the application of computing tools to data to both *extract* and *synthesize* descriptive or summary information. For example, in the case of an image represented by a TIFF file, metadata extraction operations could include extracting information contained in tags (e.g., objective used, exposure), determining the size of the file, computing the average color of the image, and applying a machine learning model to extract "entities" (for some definition of entity).

For clarity in exposition, I define the terms file, metadata, group, and storage system. I assume that the data of interest are organized as a set of files, where a *file* is the basic unit of data storage. A file, f, has two components: $f.b$, the (potentially empty) sequence of bytes that represent the file's contents; and $f.m$, the (potentially empty) set of associated *metadata*. A *group* identifies zero or more files that have some logical relationship: for example, all files associated with an experiment, or all files created on a particular day. A group g has two components: its files, $g.f$, and a (potentially empty) set of group-specific metadata, $g.m$. Note that $g.m$ and $f.m$ can contain overlapping elements. Group membership is non-exclusive: a file may be contained in more than one group.

Each file resides in a *storage system*: for example, a file system, object store, or database. Each file is located on a single storage system, but files that form a group may span multiple storage systems. For example, a group corresponding to a microscopy experiment might comprise two files: a microscopy image and a spreadsheet containing descriptive information, located on a storage cluster and on Google Drive, respectively. A storage system s may also have associated metadata, denoted $s.m$.

An extractor is a function e that when applied to a group g, with its associated files $g.f$ and metadata $g.m$, may update the group metadata $g.m$ and/or the metadata associated with one or more of the files in the group.

A.2 Bulk Metadata Extraction

I define *bulk metadata extraction* to be the task of applying a set of extractors to many files: for example, all files located on a particular storage system. Given the potential scale of both extractors in an extractor library and the number of files in a repository, only extractors applicable to each given file should be executed on it. Additionally, extractors can be prioritized (i.e., assigned an order) based on the probability that they will return valid metadata. Let R (for repository) be such a collection of files and *next* be a function that when applied to a group g and a set of extractors E returns the extractor that should be applied next to the group, that is, $e = \mathit{next}(E, g)$. Bulk metadata extraction then proceeds as follows: $\forall g \in R$, repeatedly first evaluate $e = \mathit{next}(E, g)$ and then apply $e(g)$, until $\mathit{next}(E, g) = \emptyset$. (I define extraction in terms of groups rather than files for simplicity; in practice, an extractor can update $f.m$, $g.m$, neither, or both).

I next consider the computing resources, and the accompanying constraints, of metadata extraction jobs. Let C be the set of all computing resources available for metadata extraction. Running an extractor e on a group g on a particular $c \in C$ incurs various costs, of which I consider two here: the time required to transfer g to c, $p_{tr}(c, g)$, and the time required to run $e(g)$ on c, $p_{ex}(c, e, g)$. Depending on the context, I may then want to select the extractors to apply and the locations to run those extractors to maximize some measure of utility of the extracted metadata (a complex issue [68]) subject to limits on incurred costs. Here, I assume a fixed set of extractors and focus simply on finding a mapping of extractors to compute resources that minimizes the total incurred costs:

$$\min_{a \in A} \sum_{g \in G} \sum_{e \in E(g)} p_{tr}(c, g) + p_{ex}(c, e, g)$$

where $E(g)$ is the extractors to be applied to a group g, and A is the set of all possible allocations of extractor invocations to available compute resources.

This scheduling problem is NP-complete [130], so I use heuristic-based approaches to evaluate extraction efficacy.

REFERENCES

[1] William E. Allcock, Benjamin S. Allen, Rachana Ananthakrishnan, Ben Blaiszik, Kyle Chard, Ryan Chard, Ian Foster, Lukasz Lacinski, Michael E. Papka, and Rick Wagner. 2019. Petrel: A Programmatically Accessible Research Data Service. In *Practice and Experience in Advanced Research Computing*. ACM, Chicago, IL, 1–7.

[2] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. 2019. Parsl: Pervasive Parallel Programming in Python. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '19)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/3307681.3325400

[3] Hrishav Bakul Barua and Kartick Chandra Mondal. 2019. Approximate computing: A survey of recent trends—bringing greenness to computing and communication. *Journal of The Institution of Engineers (India): Series B* 100, 6 (2019), 619–626.

[4] Paul Beckman, Tyler J Skluzacek, Kyle Chard, and Ian Foster. 2017. Skluma: A statistical learning pipeline for taming unkempt data repositories. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*. ACM, Chicago, IL, 1–4.

[5] Francine Berman, Rob Rutenbar, Brent Hailpern, Henrik Christensen, Susan Davidson, Deborah Estrin, Michael Franklin, Margaret Martonosi, Padma Raghavan, Victoria Stodden, et al. 2018. Realizing the potential of data science. *Commun. ACM* 61, 4 (2018), 67–72.

[6] David Bernstein. 2014. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing* 1, 3 (2014), 81–84.

[7] Kireet Bhat, Jason T Lam, and Farhana Zulkernine. 2018. Content-based file type identification. In *2018 10th International Conference on Electrical and Computer Engineering (ICECE)*. IEEE, IEEE, Dhaka, Bangladesh, 277–280.

[8] Ben Blaiszik, Kyle Chard, Jim Pruyne, Rachana Ananthakrishnan, Steven Tuecke, and Ian Foster. 2016. The Materials Data Facility: Data services to advance materials science research. *Jom* 68, 8 (2016), 2045–2052.

[9] Thomas R Bruce and Diane I Hillmann. 2004. The continuum of metadata quality: defining, expressing, exploiting. In *Metadata in Practice*. ALA editions, Chicago, IL.

[10] Michael J Cafarella, Alon Halevy, Daisy Zhe Wang, Eugene Wu, and Yang Zhang. 2008. Webtables: exploring the power of tables on the web. *Proceedings of the VLDB Endowment* 1, 1 (2008), 538–549.

[11] Gillian Cameron, David Cameron, Gavin Megaw, RR Bond, Maurice Mulvenna, Siobhan O'Neill, Cherie Armour, and Michael McTear. 2018. Back to the future: lessons from knowledge engineering methodologies for chatbot design and development. In *British HCI Conference 2018*. BCS Learning & Development Ltd, BCS Learning & Development Ltd, Belfast, Ireland, 1–5.

[12] David Canter, Rod Rivers, and Graham Storrs. 1985. Characterizing user navigation through complex data structures. *Behaviour & Information Technology* 4, 2 (1985), 93–102.

[13] K. Chard, S. Tuecke, and I. Foster. 2014. Efficient and Secure Transfer, Synchronization, and Sharing of Big Data. *IEEE Cloud Computing* 1, 3 (2014), 46–55.

[14] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. 2020. funcX: A federated function serving fabric for science. In *29th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, Stockholm, Sweden (Online), 65–76.

[15] Ryan Chard, Kyle Chard, and Ian Foster. 2019. Globus Automate: A distributed research automation platform. *eResearch Annual Review* 7, 1 (2019), 1–19.

[16] Ryan Chard, Tyler Skluzacek, Zhuozhao Li, Yadu Babuji, Anna Woodard, Ben Blaiszik, Steven Tuecke, Ian Foster, and Kyle Chard. 2019. Serverless supercomputing: High performance function as a service for science, In arXiv. *arXiv preprint arXiv:1908.04907* 1, 1, 1–13.

[17] Weiwei Cheng and Eyke Hüllermeier. 2009. Combining instance-based learning and logistic regression for multilabel classification. *Machine Learning* 76, 2 (2009), 211–225.

[18] Paul Alexandru Chirita, Wolfgang Nejdl, Raluca Paiu, and Christian Kohlschütter. 2005. Using ODP metadata to personalize search. In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, Salvador, Brazil, 178–185.

[19] Muntabir Hasan Choudhury, Himarsha R Jayanetti, Jian Wu, William A Ingram, and Edward A Fox. 2021. Automatic Metadata Extraction Incorporating Visual Features from Scanned Electronic Theses and Dissertations. *Joint Conference on Digital Libraries '21* 1 (2021), 230–233.

[20] Simon Clark, Francesca L Bleken, Simon Stier, Eibar Flores, Casper Welzel Andersen, Marek Marcinek, Anna Szczesna-Chrzan, Miran Gaberscek, M Rosa Palacin, Martin Uhrin, et al. 2021. Toward a Unified Description of Battery Data. *Advanced Energy Materials* 9, 48 (2021), 2102702.

[21] Cloudera. 2014. RecordBreaker. https://github.com/cloudera/RecordBr-eaker/tree/master/src. Visited Feb. 28, 2017.

[22] Weidong Cui, Marcus Peinado, Karl Chen, Helen J Wang, and Luis Irun-Briz. 2008. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, ACM, Alexandria, VA, 391–402.

[23] Costis Dallas. 2016. Digital curation beyond the “wild frontier”: A pragmatic approach. *Archival Science* 16, 4 (2016), 421–457.

[24] Anish Das Sarma, Xin Dong, and Alon Halevy. 2008. Bootstrapping pay-as-you-go data integration systems. In *ACM SIGMOD International Conference on Management of Data*. ACM, Vancouver, Canada, 861–874.

[25] André CPLF de Carvalho and Alex A Freitas. 2009. A tutorial on multi-label classification techniques. *Foundations of computational intelligence* 5 (2009), 177–195.

[26] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira Da Silva, Miron Livny, et al. 2015. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems* 46 (2015), 17–35.

[27] Eric W Deutsch, Roger Kramer, Joseph Ames, Andrew Bauman, David S Campbell, Kyle Chard, Kristi Clark, Mike D’Arcy, Ivo D Dinov, Rory Donovan, et al. 2018. BDQC: a general-purpose analytics tool for domain-blind validation of Big Data. *bioRxiv* 2018, 1 (2018), 258822.

[28] Till Döhmen, Hannes Mühleisen, and Peter Boncz. 2017. Multi-Hypothesis CSV Parsing. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*. ACM, ACM, Chicago, IL, 16.

[29] Claudia Draxl and Matthias Scheffler. 2018. NOMAD: The FAIR concept for big data-driven materials science. *Mrs Bulletin* 43, 9 (2018), 676–682.

[30] Ahmed Elkaseer, Tobias Mueller, Amal Charles, and Steffen Scholz. 2018. Digital detection and correction of errors in as-built parts: a step towards automated quality control of additive manufacturing. In *Proceedings of the World Congress on Micro and Nano Manufacturing*. MDPI, Portorož, Slovenia, 18–20.

[31] Widad Elouataoui, Imane El Alaoui, and Youssef Gahi. 2021. Metadata Quality in the Era of Big Data and Unstructured Content. In *The International Conference on Information, Communication & Cybersecurity*. Springer, Khouribga, Morocco, 110–121.

[32] ESS-Dive. 2018. CDIAC. `https://cdiac.ess-dive.lbl.gov/`

[33] Mina Farid, Alexandra Roatis, Ihab F Ilyas, Hella-Franziska Hoffmann, and Xu Chu. 2016. CLAMS: bringing quality to data lakes. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, San Francisco, CA, 2089–2092.

[34] Kathleen Fisher and David Walker. 2011. The PADS project: An overview. In *14th International Conference on Database Theory*. ACM, ACM, Uppsala, Sweden, 11–17.

[35] Kathleen Fisher, David Walker, Kenny Q Zhu, and Peter White. 2008. From dirt to shovels: fully automatic tool generation from ad hoc data. In *ACM SIGPLAN Notices*, Vol. 43. ACM, ACM, San Francisco, CA, 421–434.

[36] Fn. 2020. Fn project. `https://fnproject.io`. Accessed April 20, 2020.

[37] Óscar Fontenla-Romero, Bertha Guijarro-Berdiñas, David Martinez-Rego, Beatriz Pérez-Sánchez, and Diego Peteiro-Barral. 2013. Online machine learning. In *Efficiency and Scalability Methods for Computational Intellect*. IGI Global, Hershey, PA, 27–54.

[38] Inria Foundation. 2022. Scikit-Learn ExtraTreesClassifier Documentation. `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html`

[39] Python Software Foundation. 2022. AST - abstract syntax trees. `https://docs.python.org/3/library/ast.html`

[40] The Apache Software Foundation. 2022. Apache OpenWhisk. `http://openwhisk.apache.org/`. Accessed Jan 1, 2022.

[41] The Linux Foundation. 2009. Libmagic(3) - linux man page. `https://linux.die.net/man/3/libmagic`

[42] Michael Franklin, Alon Halevy, and David Maier. 2005. From databases to dataspaces: A new abstraction for information management. *ACM SIGMOD Record* 34, 4 (2005), 27–33.

[43] Johannes Fürnkranz, Eyke Hüllermeier, Eneldo Loza Mencía, and Klaus Brinker. 2008. Multilabel classification via calibrated label ranking. *Machine learning* 73, 2 (2008), 133–153.

[44] Yihan Gao, Silu Huang, and Aditya Parameswaran. 2016. *Navigating the Data Lake: Unsupervised Structure Extraction for Text-formatted Data.* Technical Report. Technical report, U. Illinois (UIUC), `http://publish.illinois.edu/structextract/files/2016/12/struct_extract.pdf`.

[45] Daniel Gomes, Miguel Costa, David Cruz, João Miranda, and Simão Fontes. 2013. Creating a billion-scale searchable web archive. In *Proceedings of the 22nd International Conference on World Wide Web*. ACM, Rio de Janeiro, Brazil, 1059–1066.

[46] Daniel J Gonçalves and Joaquim A Jorge. 2003. An empirical study of personal document spaces. In *International Workshop on Design, Specification, and Verification of Interactive Systems*. Springer, Springer, Funchal, Madeira Island, Portugal, 46–60.

[47] Google. 2022. Google Cloud Functions. `https://cloud.google.com/functions/`. Accessed Jan 1, 2022.

[48] Siddharth Gopal, Yiming Yang, Konstantin Salomatin, and Jaime Carbonell. 2011. Statistical learning for file-type identification. In *2011 10th international conference on machine learning and applications and workshops*, Vol. 1. IEEE, IEEE, Honolulu, HI, 68–73.

[49] Saul Greenberg. 2009. Promoting creative design through toolkits. In *2009 Latin American Web Congress*. IEEE, IEEE, Merida, Yucatan, Mexico, 92–93.

[50] FAIR Metrics Group. 2021. FAIR metrics. `http://fairmetrics.org`.

[51] Kenneth Haase. 2004. Context for semantic metadata. In *Proceedings of the 12th annual ACM international conference on Multimedia*. ACM, New York, NY, 204–211.

[52] Qinlu He, Zhanhuai Li, and Xiao Zhang. 2010. Data deduplication techniques. In *International Conference on Future Information Technology and Management Engineering*, Vol. 1. IEEE, IEEE, Changzhou, China, 430–433.

[53] Jeffrey Heer, Joseph M Hellerstein, and Sean Kandel. 2015. Predictive Interaction for Data Transformation. In *CIDR*. CIDR, Asilomar, California, 1–7.

[54] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-sketch: Output-directed programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. ACM, New Orleans, LA, 281–292.

[55] Sarah Henderson and Ananth Srinivasan. 2009. An empirical analysis of personal digital document structures. In *Symposium on Human Interface*. Springer, Springer, San Diego, CA, 394–403.

[56] Patrick Herring, Chirranjeevi Balaji Gopal, Muratahan Aykol, Joseph H Montoya, Abraham Anapolsky, Peter M Attia, William Gent, Jens S Hummelshøj, Linda Hung, Ha-Kyung Kwon, et al. 2020. BEEP: A python library for battery evaluation and early prediction. *SoftwareX* 11 (2020), 100506.

[57] Tin Kam Ho. 1998. The random subspace method for constructing decision forests. *IEEE transactions on pattern analysis and machine intelligence* 20, 8 (1998), 832–844.

[58] Helena Holmström and Ola Henfridsson. 2006. Improving packaged software through online community knowledge. *Scandinavian Journal of Information Systems* 18, 1 (2006), 2.

[59] Zhi Hong, Logan Ward, Kyle Chard, Ben Blaiszik, and Ian Foster. 2021. Challenges and Advances in Information Extraction from Scientific Literature: a Review. *JOM* 73, 11 (2021), 3383–3400.

[60] Baden Hughes. 2004. Metadata quality evaluation: Experience from the open language archives community. In *International Conference on Asian Digital Libraries*. Springer, Springer, Shanghai, China, 320–329.

[61] Jane Hung, Allen Goodman, Deepali Ravel, Stefanie CP Lopes, Gabriel W Rangel, Odailton A Nery, Benoit Malleret, Francois Nosten, Marcus VG Lacerda, Marcelo U Ferreira, et al. 2020. Keras R-CNN: library for cell detection in biological images using deep neural networks. *BMC bioinformatics* 21, 1 (2020), 1–7.

[62] William Jones, Ammy Jiranida Phuwanartnurak, Rajdeep Gill, and Harry Bruce. 2005. Don't take my folders away! organizing personal information to get things done. In *CHI'05 extended abstracts on Human factors in computing systems*. ACM, Portland, OR, 1505–1508.

[63] Stephen Kaisler, Frank Armour, J Alberto Espinosa, and William Money. 2013. Big data: Issues and challenges moving forward. In *2013 46th Hawaii international conference on system sciences*. IEEE, IEEE, Maui, HI, 995–1004.

[64] David Karger and Clifford Stein. 1996. A new approach to the minimum cut problem. *J. ACM* 43, 4 (1996), 601–640.

[65] Ban Mohammed Khammas. 2020. Ransomware detection using random forest technique. *ICT Express* 6, 4 (2020), 325–331.

[66] Gary King. 2007. *An introduction to the Dataverse network as an infrastructure for data sharing*. Sage Publications, Thousand Oaks, CA.

[67] Ross D King, Jem Rowland, Stephen G Oliver, Michael Young, Wayne Aubrey, Emma Byrne, Maria Liakata, Magdalena Markham, Pinar Pir, Larisa N Soldatova, et al. 2009. The automation of science. *Science* 324, 5923 (2009), 85–89.

[68] Péter Király. 2019. *Measuring Metadata Quality*. Ph.D. Dissertation. University of Göttingen. `https://doi.org/10.13140/RG.2.2.33177.77920`

[69] Oleksii Kononenko, Olga Baysal, Reid Holmes, and Michael W Godfrey. 2014. Mining modern repositories with elasticsearch. In *Proceedings of the 11th working conference on mining software repositories*. ACM, Hyderabad, India, 328–331.

[70] Morten Kristensen. 2022. Vermin. `https://github.com/netromdk/vermin`. Visited Jan 19, 2022.

[71] Ari Kulmala, Erno Salminen, and Timo D Hämäläinen. 2007. Evaluating large system-on-chip on multi-FPGA platform. In *International Workshop on Embedded Computer Systems*. Springer, Springer, Samos, Greece, 179–189.

[72] Marilyn F Lamb, Richard A Feely, and Lloyd Moore. 1995. *Total carbon dioxide, hydrographic, and nitrate measurements in the southwest Pacific during austral autumn,*

1990: Results from NOAA/PMEL CGC-90 cruise. Technical Report. Oak Ridge National Lab., TN (United States). Carbon Dioxide Information Analysis Center.

[73] Andrew W Leung, Minglong Shao, Timothy Bisson, Shankar Pasupathy, and Ethan L Miller. 2009. Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems.. In *FAST*, Vol. 9. USENIX Association, San Diego, CA, 153–166.

[74] Li Li, Yuxi Fan, Mike Tse, and Kuo-Yi Lin. 2020. A review of applications in federated learning. *Computers & Industrial Engineering* 149 (2020), 106854.

[75] Wei-Jen Li, Ke Wang, Salvatore J Stolfo, and Benjamin Herzog. 2005. Fileprints: Identifying file types by n-gram analysis. In *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop.* IEEE, IEEE, West Point, NY, 64–71.

[76] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft coco: Common objects in context. In *European Conference on Computer Vision.* Springer, Springer, Zurich, Switzerland, 740–755.

[77] Merkourios Margaritopoulos, Thomas Margaritopoulos, Ioannis Mavridis, and Athanasios Manitsaris. 2012. Quantifying and measuring metadata completeness. *Journal of the American Society for Information Science and Technology* 63, 4 (2012), 724–737.

[78] Luigi Marini, Indira Gutierrez-Polo, Rob Kooper, Sandeep Puthanveetil Satheesan, Maxwell Burnette, Jong Lee, Todd Nicholson, Yan Zhao, and Kenton McHenry. 2018. Clowder: Open source data management for long tail data. In *Practice and Experience on Advanced Research Computing.* ACM, Pittsburgh, PA, 1–8.

[79] Chris Mattmann and Jukka Zitting. 2011. *Tika in Action.* Manning Publications Co., USA.

[80] Mason McDaniel and Mohammad Hossain Heydari. 2003. Content based file type detection algorithms. In *36th Annual Hawaii International Conference on System Sciences.* IEEE, IEEE, Big Island, HI, 10–pp.

[81] Dutch Meyer and William Bolosky. 2012. A study of practical deduplication. *ACM Transactions on Storage* 7, 4 (2012), 1–20.

[82] Microsoft. 2020. Microsoft Azure Functions Documentation. `https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale`. Accessed April 20, 2020.

[83] Mackenzie Mitchell. 2019. Programming Languages For Data Scientists. `https://towardsdatascience.com/programming-languages-for-data-scientists-afde2eaf5cc5`

[84] Johann Mitlöhner, Sebastian Neumaier, Jürgen Umbrich, and Axel Polleres. 2016.

Characteristics of open data CSV files. In *2016 2nd International Conference on Open and Big Data (OBD)*. IEEE, IEEE, Vienna, Austria, 72–79.

[85] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. ACM, Carlsbad, CA, 561–577.

[86] Anthony J Myles, Robert N Feudale, Yang Liu, Nathaniel A Woody, and Steven D Brown. 2004. An introduction to decision tree modeling. *Journal of Chemometrics: A Journal of the Chemometrics Society* 18, 6 (2004), 275–285.

[87] Goda Naujokaityte. 2021. Number of scientists worldwide reaches 8.8m, as global research spending grows faster than the economy. `https://sciencebusiness.net/news`

[88] Apache Nutch. 2014. Apache Nutch.

[89] Xavier Ochoa and Erik Duval. 2009. Automatic evaluation of metadata quality in digital repositories. *International journal on digital libraries* 10, 2-3 (2009), 67–91.

[90] Stanislaw Osiński and Dawid Weiss. 2004. Conceptual clustering using lingo algorithm: Evaluation on open directory project data. In *Intelligent Information Processing and Web Mining*. Springer, Zakopane, Poland, 369–377.

[91] Mahesh Pal. 2005. Random forest classifier for remote sensing classification. *International journal of remote sensing* 26, 1 (2005), 217–222.

[92] Rainer Poisel and Simon Tjoa. 2013. A comprehensive literature review of file carving. In *2013 International conference on availability, reliability and security*. IEEE, Regensburg, Germany, 475–484.

[93] Milo Polte, Jay Lofstead, John Bent, Garth Gibson, Scott A Klasky, Qing Liu, Manish Parashar, Norbert Podhorszki, Karsten Schwan, Meghan Wingate, et al. 2009. ... And eat it too: High read performance in write-optimized HPC I/O middleware file formats. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*. ACM, Portland, Oregon, 21–25.

[94] Christoph Quix, Rihan Hai, and Ivan Vatov. 2016. GEMMS: A generic and extensible metadata management system for data lakes. In *CAiSE Forum*. Springer, Ljubljana, Slovenia, 129–136.

[95] Arcot Rajasekar, Reagan Moore, Chien-yi Hou, Christopher A Lee, Richard Marciano, Antoine de Torcy, Michael Wan, Wayne Schroeder, Sheau-Yen Chen, Lucas Gilbert, et al. 2010. iRODS primer: Integrated rule-oriented data system. *Synthesis Lectures on Inf. Concepts, Retrieval, and Services* 2, 1 (2010), 1–143.

[96] Dennis M Ritchie and Ken Thompson. 1978. The UNIX time-sharing system. *Bell System Technical Journal* 57, 6 (1978), 1905–1929.

[97] Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, Vol. 130. Citeseer, Citeseer, Austin, TX, 136.

[98] Gonzalo Rodrigo, Matt Henderson, Gunther Weber, Colin Ophus, Katie Antypas, and Lavanya Ramakrishnan. 2018. ScienceSearch: Enabling search through automatic metadata generation. In *14th International Conference on e-Science*. IEEE, Amsterdam, Holland, 93–104.

[99] Minseok Ryu, Youngdae Kim, Kibaek Kim, and Ravi K Madduri. 2022. APPFL: Open-Source Software Framework for Privacy-Preserving Federated Learning. *arXiv preprint arXiv:2202.03672* 1 (2022), 1–10.

[100] Chris Seltzer and Igor Jablokov. 2015. Distributed cloud storage. US Patent App. 14/788,618.

[101] Amazon Web Services. 2021. AWS Lambda. `https://aws.amazon.com/lambda/`. Visited Jan 1, 2022.

[102] Amazon Web Services. 2022. AWS Snowball. `https://aws.amazon.com/snowball`. Visited Jan 1, 2022.

[103] Tim Shaffer, Nicholas Hazekamp, Jakob Blomer, and Douglas Thain. 2020. Solving the Container Explosion Problem for Distributed High Throughput Computing. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, IEEE, New Orleans, LA, 388–398.

[104] I Shah and Amit Sheth. 1999. INFOHARNESS: managing distributed, heterogeneous information. *IEEE Internet Computing* 3, 6 (1999), 18–28.

[105] James G Shanahan and Laing Dai. 2015. Large scale distributed data science using apache spark. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. ACM, Sydney, Australia, 2323–2324.

[106] Claude Elwood Shannon. 2001. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review* 5, 1 (2001), 3–55.

[107] Aakanksha Sharaff and Harshil Gupta. 2019. Extra-tree classifier with metaheuristics approach for email classification. In *Advances in computer communication and computational sciences*. Springer, Singapore, 189–197.

[108] Bayan Abu Shawar and Eric Atwell. 2007. Chatbots: are they really useful?. In *LDV Forum*, Vol. 22. Gesellschaft für Linguistische Datenverarbeitung, Zurich, Switzerland, 29–49.

[109] Leon Shklar, Amit Sheth, Vipul Kashyap, and Kshitij Shah. 1995. InfoHarness: Use of automatically generated metadata for search and retrieval of heterogeneous information. In *International Conference on Advanced Information Systems Engineering*. Springer, Springer, Jyväskylä, Finland, 217–230.

[110] Tyler Skluzacek, Ryan Wong, Zhuozhao Li, Ryan Chard, Kyle Chard, Ian Foster, and Kyle Chard. 2021. A Serverless Framework for Distributed Bulk Metadata Extraction. In *30th International Symp on High-Performance Parallel and Distributed Computing*. ACM, Stockholm, Sweden (Online), 12.

[111] Tyler J Skluzacek. 2019. Dredging a data lake: decentralized metadata extraction. In *Proceedings of the 20th International Middleware Conference Doctoral Symposium*. ACM, Davis, CA, 51–53.

[112] Tyler J Skluzacek, Kyle Chard, and Ian Foster. 2016. Klimatic: a virtual data lake for harvesting and distribution of geospatial data. In *2016 1st Joint International Workshop on Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS)*. IEEE, IEEE, Salt Lake City, UT, 31–36.

[113] Tyler J Skluzacek, Ryan Chard, Ryan Wong, Zhuozhao Li, Yadu N Babuji, Logan Ward, Ben Blaiszik, Kyle Chard, and Ian Foster. 2019. Serverless workflows for indexing large scientific data. In *Proceedings of the 5th International Workshop on Serverless Computing*. ACM, Davis, CA, 43–48.

[114] Tyler J Skluzacek, Matthew Chen, Erica Hsu, Kyle Chard, and Ian Foster. 2022. Models and Metrics for Mining Meaningful Metadata, In International Conference on Computational Science. *International Conference on Computational Science* 1, 1–14.

[115] Tyler J Skluzacek, Rohan Kumar, Ryan Chard, Galen Harrison, Paul Beckman, Kyle Chard, and Ian T Foster. 2018. Skluma: An extensible metadata extraction pipeline for disorganized data. In *2018 IEEE 14th International Conference on e-Science (e-Science)*. IEEE, Amsterdam, Holland, 256–266.

[116] Tyler J Skluzacek, Suhail Rehman, and Ian Foster. 2017. Safe Double Blind Studies as a Service. In *2017 IEEE 13th International Conference on e-Science (e-Science)*. IEEE, IEEE, Auckland, NZ, 504–509.

[117] Apache Spark. 2018. Apache spark. *Retrieved January* 17, 2018 (2018), 1.

[118] Alexandra K Stephan. 2021. Standardized battery reporting guidelines. *Joule* 5, 1 (2021), 1–2.

[119] Craig Stewart, Timothy Cockerill, Ian Foster, David Hancock, Nirav Merchant, Edwin Skidmore, Daniel Stanzione, James Taylor, Steven Tuecke, George Turner, et al. 2015. Jetstream: A self-provisioned, scalable science and engineering cloud environment. In *XSEDE Conference*. ACM, St. Louis, MO, 1–8.

[120] Yang-Kook Sun. 2021. An experimental checklist for reporting battery performances. , 2187–2189 pages.

[121] S Momina Tabish, M Zubair Shafiq, and Muddassar Farooq. 2009. Malware detection using statistical analysis of byte-level file content. In *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics*. ACM, Paris, France, 23–31.

[122] John Talburt. 1986. The Flesch index: An easily programmable readability analysis algorithm. In *Int'l Conference on Systems Documentation*. ACM, Pisa, Italy, 114–122.

[123] Roselyne B. Tchoua, Kyle Chard, Debra J. Audus, Logan T. Ward, Joshua Lequieu, Juan J. De Pablo, and Ian T. Foster. 2017. Towards a Hybrid Human-Computer Scientific Information Extraction Pipeline. In *2017 IEEE 13th International Conference on e-Science (e-Science)*. IEEE, Auckland, NZ, 109–118. https://doi.org/10.1109/eScience.2017.23

[124] Carol Tenopir, Lisa Christian, Suzie Allard, and Josh Borycz. 2018. Research data sharing: Practices and attitudes of geophysicists. *Earth and Space Science* 5, 12 (2018), 891–902.

[125] Ignacio Terrizzano, Peter Schwarz, Mary Roth, and John Colino. 2015. Data wrangling: The challenging journey from the wild to the lake. In *Conference on Innovative Data Systems Research*. IEEE, Asilomar, CA, 1–9.

[126] Parastou Tourani, Bram Adams, and Alexander Serebrenik. 2017. Code of conduct in open source projects. In *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, IEEE, Klagenfurt, Austria, 24–33.

[127] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory Peterson, et al. 2014. XSEDE: Accelerating scientific discovery. *Computing in Science & Engineering* 16, 5 (2014), 62–74.

[128] Kuen Hung Tsoi and Wayne Luk. 2010. Axel: A heterogeneous cluster with FPGAs and GPUs. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, Monterey, CA, 115–124.

[129] Steven Tuecke, Rachana Ananthakrishnan, Kyle Chard, Mattias Lidman, Brendan McCollam, Stephen Rosen, and Ian Foster. 2016. Globus Auth: A research identity and access management platform. In *12th International Conference on e-Science*. IEEE, IEEE, Baltimore, MD, 203–212.

[130] Jeffrey Ullman. 1975. NP-complete scheduling problems. *Journal of Computer and System sciences* 10, 3 (1975), 384–393.

[131] S. S. Vazhkudai, J. Harney, R. Gunasekaran, D. Stansberry, S. Lim, T. Barron, A. Nash, and A. Ramanathan. 2016. Constellation: A science graph network for scal-

able data and knowledge discovery in extreme-scale scientific collaborations. In *IEEE Internationial Conference on Big Data*. IEEE, Washington D.C., USA, 3052–3061. `https://doi.org/10.1109/BigData.2016.7840959`

[132] Lucy Lu Wang, Kyle Lo, et al. 2020. Cord-19: The covid-19 open research dataset. *arXiv:2004.10706* 1, 1 (2020), 1–12. `https://doi.org/10.48550/ARXIV.2004.10706`

[133] Yi Wang, Yi Li, et al. 2015. Efficient test for nonlinear dependence of two continuous variables. *BMC Bioinformatics* 16, 1 (2015), 1–8.

[134] Logan Ward. 2019. MaterialsIO. `https://github.com/materials-data-facility/MaterialsIO`.

[135] Danielle Welter, Jacqueline MacArthur, Joannella Morales, Tony Burdett, Peggy Hall, Heather Junkins, Alan Klemm, Paul Flicek, Teri Manolio, Lucia Hindorff, et al. 2014. The NHGRI GWAS Catalog, a curated resource of SNP-trait associations. *Nucleic Acids Research* 42, D1 (2014), D1001–D1006.

[136] Mark Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip Bourne, et al. 2016. The FAIR guiding principles for scientific data management and stewardship. *Scientific Data* 3, 1 (2016), 1–9.

[137] Mark D Wilkinson, Susanna-Assunta Sansone, Erik Schultes, Peter Doorn, Luiz Olavo Bonino da Silva Santos, and Michel Dumontier. 2018. A design framework and exemplar metrics for FAIRness. *Scientific data* 5, 1 (2018), 1–4.

[138] Ryan Wong, Tyler J Skluzacek, and Kyle Chard. 2021. Delving Into the Abyss: A Distributed Decompression System for Indexing Compressed Repositories. *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC21)* 1 (Nov 2021), 1–3.

[139] Justin Wozniak, Kyle Chard, Ben Blaiszik, Ray Osborn, Michael Wilde, and Ian Foster. 2015. Big data remote access interfaces for light source science. In *2nd International Symposium on Big Data Computing*. IEEE, IEEE, Limassol, Cyprus, 51–60.

[140] Margaret Wright et al. 2010. The Opportunities and Challenges of Exascale Computing. *Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee* 1 (2010), 0–71.

[141] Wei Zhang, Suren Byna, Chenxu Niu, and Yong Chen. 2019. Exploring metadata search essentials for scientific data management. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, IEEE, Hyderabad, India, 83–92.

[142] Lina Zhou, Shimei Pan, Jianwu Wang, and Athanasios V Vasilakos. 2017. Machine learning on big data: Opportunities and challenges. *Neurocomputing* 237 (2017), 350–361.