# A Serverless Framework for Distributed Bulk Metadata Extraction

Tyler J. Skluzacek
University of Chicago

Ryan Wong
University of Chicago

Zhuozhao Li
University of Chicago

Ryan Chard
Argonne National Laboratory

Kyle Chard
Argonne National Laboratory and University of Chicago

Ian Foster
Argonne National Laboratory and University of Chicago

## ABSTRACT

We introduce Xtract, an automated and scalable system for bulk metadata extraction from large, distributed research data repositories. Xtract orchestrates the application of metadata extractors to groups of files, determining which extractors to apply to each file and, for each extractor and file, where to execute. A hybrid computing model, built on the funcX federated FaaS platform, enables Xtract to balance tradeoffs between extraction time and data transfer costs by dispatching each extraction task to the most appropriate location. Experiments on a range of clouds and supercomputers show that Xtract can efficiently process multi-million-file repositories by orchestrating the concurrent execution of container-based extractors on thousands of nodes. We highlight the flexibility of Xtract by applying it to a large, semi-curated scientific data repository and to an uncurated scientific Google Drive repository. We show that by remotely orchestrating metadata extraction across decentralized storage and compute nodes, Xtract can process large repositories in 50% of the time it takes just to transfer the same data to a machine within the same computing facility. We also show that when transferring data is necessary (e.g., no local compute is available), Xtract can scale to process files as fast as they are received, even over a multi-GB/s network.

## KEYWORDS

serverless; metadata extraction; search index; storage; files

**ACM Reference Format:**

Tyler J. Skluzacek, Ryan Wong, Zhuozhao Li, Ryan Chard, Kyle Chard, and Ian Foster. 2021. A Serverless Framework for Distributed Bulk Metadata Extraction. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '21), June 21–25, 2021, Virtual Event, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3431379.3460636

## 1 INTRODUCTION

The abundance of files accumulated within science and engineering organizations may, both individually and collectively, contain data of great value. However, poor organization and inadequate documentation frequently make these files inaccessible. Standard file systems and object stores do little more than de-duplicate files; users need methods for inferring file contents and for linking related files, both to simplify navigation of large collections and to enhance awareness of contents and semantics.

Automated methods for extracting or inferring information about file contents and relationships can support and complement the FAIR (findable, accessible, interoperable, and reproducible) data principles. Ideally, every file created within an organization would be described in a standardized manner and in so much detail that it could trivially be discovered and repurposed for interdisciplinary or cross-institutional research [32] throughout the research lifecycle [15]. But in practice, files are often created without thought for future reuse. Thus, as a prerequisite to FAIR data, organizations must first mine latent information from their files, and from this information, synthesize a desired level of semantic meaning. We call the union of the mined information and the semantic meaning *metadata*, and the process of acquiring it *metadata extraction.*

Metadata extraction grows more difficult as data sizes, types, and sources increase; storage becomes more distributed; and computational methods for learning about files change. Therefore manual metadata extraction must give way to automated methods capable of handling large, distributed data collections containing idiosyncratic file formats [9, 17] and spanning multiple storage systems [22]. However, existing metadata extraction methods either operate entirely locally (e.g., on a personal file system) or require that data be moved to a central location (e.g., to the cloud). But 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 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, we propose the following research question: *can FaaS infrastructure enable the creation of scalable, efficient, decentralized metadata extraction workflows for large, distributed scientific data?*

We present 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 deployment of performant metadata extraction workflows across a continuum of decentralized computing resources. This decoupling is made possible by the use of 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 paper extends our prior work [23, 24] by creating a system that leverages federated FaaS to construct scalable, efficient, decentralized metadata extraction workflows on scientific data. The contributions of our 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 over 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 paper is as follows. §2 describes unique requirements of scientific metadata extraction and our case study repositories. §3 describes the Xtract design and §4 presents its architecture and implementation. §5 explores performance and scalability in scientific case studies. §6 and §7 discuss related work and summarize our contributions, respectively.

## 2 BACKGROUND

Metadata extraction is a problem that extends across scales and disciplines. We first formalize metadata extraction as an optimization problem with customizable constraints, and then review several scientific use cases that require scalable extraction, while also illuminating the constraints, or challenges, of each.

### 2.1 Terminology

We 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, we define the terms file, metadata, group, and storage system. We assume that the data of interest are organized as a set of files, where the *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, $g.m$, and we acknowledge 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 the 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.

### 2.2 Bulk Metadata Extraction

We 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. Let $R$ (for repository) be such a collection of files and *next* be a function that when applied to a group $g$ and set of extractors $E$ returns the extractor that should be applied next to the group: i.e., $e = next(E, g)$. Bulk metadata extraction then proceeds as follows: $\forall g \in R$, repeatedly first evaluate $e = next(E, g)$ and then apply $e(g)$, until $next(E, g) = \emptyset$. (We define extraction in terms of groups rather than files for simplicity; in practice, an extractor can update $f.m$, $g.m$, neither, or both)

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 we 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 context, we may then want to select the extractors to apply and the locations to run those extractors so as to maximize some measure of utility of the extracted metadata (a complex issue [13]) subject to limits on incurred costs. Here, we 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 [29] and thus we explore various heuristics in the following.

## 2.3 Use Cases

We examine three real-world research repository examples that benefit from automated metadata extraction, and discuss the relative utility expectations and cost from each. Table 1 summarizes characteristics of these repositories.

The **Materials Data Facility** (MDF) [5] is a centralized hub for publishing, sharing, and discovering materials science data. MDF stores over 19 million files (61 TB of data) from different research groups, covering many disciplines of materials science, containing a diverse range of file types. However, the expansive range of materials data held by MDF can make it difficult for users to find data relevant to their work, so utility is rooted in the quality of metadata elements to make data findable and accessible. MDF data are primarily stored at Argonne National Laboratory (ANL) and the National Center for Supercomputing Applications (NCSA), and are accessible via Globus. Close proximity to large-scale computing resources makes transfer costs relatively low for extractors applied at the designated computing resources.

The **Carbon Dioxide Information Analysis Center** (CDIAC) compiled an emissions dataset from the 1800s through 2017. The dataset contains more than 330 GB in 500 000 files, with over 10 000 unique file extensions. The archive contains little descriptive metadata and includes a number of irrelevant files, such as as debug-cycle error logs and Windows desktop shortcuts. We can increase the utility of these data by making them more easily discoverable by the wider research community.

**Individual researchers and research groups** may store data in many locations, including laptops, cloud storage (S3, Google Drive), and computing facilities. Data are accessible via different protocols, such as HTTP and Globus. The utility of these data can be enhanced by making it possible for researchers to search them, track versions, and link data and publications. Choices of computing locations can be influenced by computing allocations that change over time and by the use of storage systems (e.g., Google Drive) that are not mounted on computing resources, necessitating data transfer prior to extraction. We analyze here the semi-scientific repository of a graduate student's Google Drive account.

**Table 1: Characteristics of our example data repositories.**

| Repository | Size (TB) | Files | Unique Extensions |
|---|---|---|---|
| MDF | 61 | 19 968 947 | 11 560 |
| CDIAC | 0.33 | 500 001 | 152 |
| Individuals | 0.005 | 4K | 71 |

## 3 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.

We 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. We discuss our family generating algorithm in §4.3.1.

**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.** Extractors are functions that take a file group as input and generate a dictionary of extracted and synthesized metadata for that group. Xtract includes 12 extractors (listed in §4) for

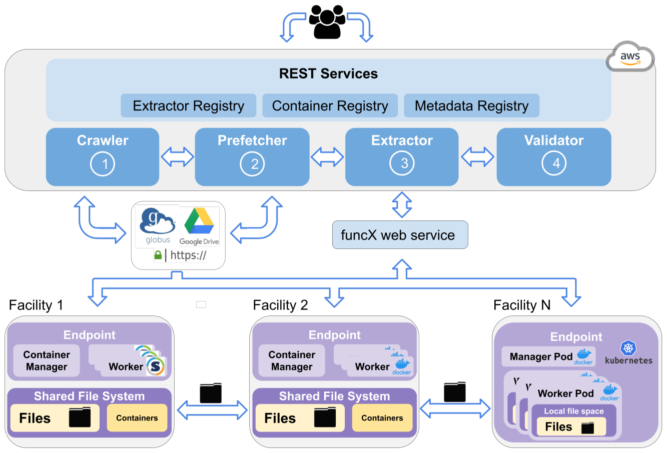

Figure 1: Overview of the Xtract architecture.

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 compute substrate to remotely execute extractors. We 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 the metadata extractors on the files, and sending results back to the Xtract service.

**Validation (and Transformation).** The validation step ensures that 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 supplied metadata and sends a valid JSON document to a user's Globus endpoint.

## 4 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 1 presents an overview of the Xtract architecture.

### 4.1 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 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 of the 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 resources, 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 discovered files, and add newly-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

**Listing 1: Example code of an extractor to extract keywords from documents stored in Google Drive**

```
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}
```

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 address 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 1 shows an example extractor function that extracts metadata from a file stored locally on a compute endpoint. We provide an xtract_sdk Python SDK to simplify access to remote files and packing and unpacking metadata objects.

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 [7] and Globus [6]. 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 from one endpoint to another. While endpoints also support direct download from cloud repositories such as Google Drive and AWS S3, they do not yet support 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 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 [28] 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 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 authorizations for programmatic access. To support Google Drive repositories we 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, we 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. Listing 2 illustrates how a user with two crawlable computing resources (one capable of running a funcX endpiont) can authenticate, register and execute functions, and track the progress of a two-endpoint metadata extraction. In this listing, the `store_path=None` indicates that endpoint `globus_ep_2` lacks a compute layer; Xtract will then automatically move the files to another endpoint with a valid funcX endpoint id.

## 4.2 Extractors

We briefly summarize several extractors, their use cases for this work, and the types of files (or file components) on which they are meant to operate. We describe these extractors and present detailed performance information in a previous paper [23].

The **MaterialsIO set of extractors** [3] 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), we have written a grouping function that executes at

**Listing 2: Example code of someone registering an Xtract extractor, invoking an extraction job, and monitoring progress.**

```python
from xtract_sdk import XtractClient
import keyword_extract

# Xtract Client: authenticate via Globus/Google Auth
xmc = XtractClient(cache_credentials=True,
                  resources=['GLOBUS'])

# Register an extractor
ext_id = xmc.register_extractor(func=keyword_xtract,
                                container_id='<UUID>')

# Globus endpoints with and without computing layer
globus_ep_1 = {'ep_id': <UUID>,
               'read_path': /science/data,
               'store_path': /tmp/xtract,
               'available_gb': 32}
globus_ep_2 = {'ep_id': <UUID>,
               'read_path': /other_science/papers,
               'store_path': None}

# FuncX endpoints
fx_ep = {'ep_id': <UUID>,
         'container_path': /path/to/containers}

# Submit, then check crawl and extraction status
task_id = xmc.submit(headers=auth_headers,
                     grouper='extension',
                     resources=[{'globus': globus_ep_1,
                                 'funcx': fx_ep},
                                {'globus': globus_ep_2,
                                 'funcx': None}])
print(xmc.get_crawl_status(task_id))
print(xmc.get_extract_status(task_id))
```

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). In order to generate these classifications, we 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, we apply the ImageNet extractor mentioned in the following. If the figure is a map, we 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 beyond those analyzed in this work. These include **hierarchical** for NetCDF and HDF files, **null-value** to determine null-values in tabular data, **Python** and **C** for isolating comment and function names 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.

## 4.3 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.

*4.3.1 Families: Transfer Minimization.* 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. In order to avoid these costs, we 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 we set a user-configurable maximum group size $s > 0$. Thus, we can minimize transferring the same file twice, or inversely, transfer a file and then invoke all of its groups' extractors on it.

In order to facilitate building families with minimal overhead, we developed the *min-transfers* algorithm that leverages Karger's Randomized Min-Cut algorithm [11]. 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. We 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 $g \in G$, we run Karger's Min-Cut to determine an approximate minimum cut, producing two subgraphs. We 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 minimum-transfer families, the crawler asynchronously enqueues it for processing by the Xtract service.

To calculate the efficiency of our approach, we start with the $O(E) = O(|V|^2)$ 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.

*4.3.2 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

**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
```

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, we exploit **funcX batching** to reduce the number of funcX web service requests. Here, we 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. We also use funcX's batch polling functionality to retrieve the status and output of completed functions. Batching at both levels enables us to not only amortize costs at the function execution level, but maximize file throughput through the Web services.

*4.3.3 Offloading.* Xtract can offload tasks to other idle resources in order to maximize 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 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 extractors are serialized and shipped, and only sends the extractors upon confirming that transfers completed successfully.

## 5 EVALUATION

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

### 5.1 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. We 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 Intell 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. We use the Broadwell partition.

**Jetstream** [25, 27] 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. We use m1.large (10 vCPU, 10 GB RAM) instance types in the TACC cluster.

We also leverage **Petrel** [4] as a data store. Petrel is a data service hosted at ANL that provides user-managed storage allocations to the research community. It is an eight-node 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 [17], we 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, we use Xtract to move files between resources, when appropriate.

### 5.2 Scalability and Throughput

Effectively processing metadata from the data sizes present in modern science requires that Xtract scale to a large number of concurrent extraction processes. To evaluate this, we 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 overall execution time, we focus here on only the metadata extraction process. We evaluate crawler scaling in §5.4. 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,

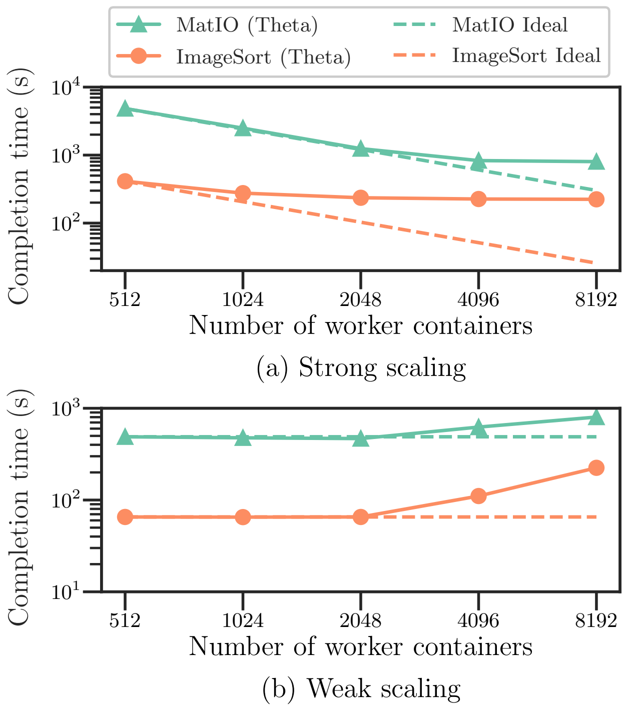

**Figure 2: Strong and weak scaling of ImageSort and MaterialsIO metadata extraction tasks.**

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's scalability we 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.

We apply these extractors to two representative datasets. For the image extractor, we use the 2014 Common Objects in Context training dataset of 80 000 images (14 GB) [14]. For MaterialsIO we use a subset of the MDF dataset: 200 000 file groups (1.1 TB), chosen uniformly at random. We 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).

*5.2.1 Strong Scaling.* Figure 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, we see that the completion time decreases until 4096 workers are employed. We conclude that Xtract is primarily limited by the rate at which funcX delivers tasks and data to an endpoint.

*5.2.2 Weak Scaling.* To evaluate weak scaling we employ concurrent extraction tasks where each worker, on average, receives 24 ImageSort and MaterialsIO extraction tasks. We see in Figure 2(b) that Xtract maintains good throughput for both the ImageSort and

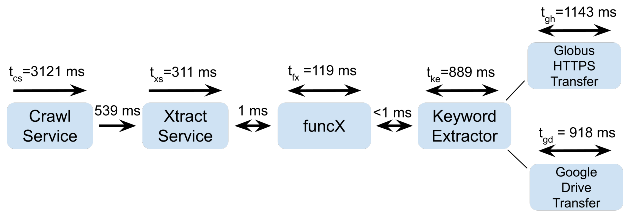

**Figure 3: Xtract latency breakdown across all components (boxes) and the communication costs between them. Unidirectional arrows imply we measure data flow latency in just one direction (downstream), whereas bi-directional arrows imply the sum of latency in both directions (downstream and upstream).**

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.

*5.2.3 Throughput.* We 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 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, we 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 3. As this endpoint has no shared file system, we 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 resolving the endpoint 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,

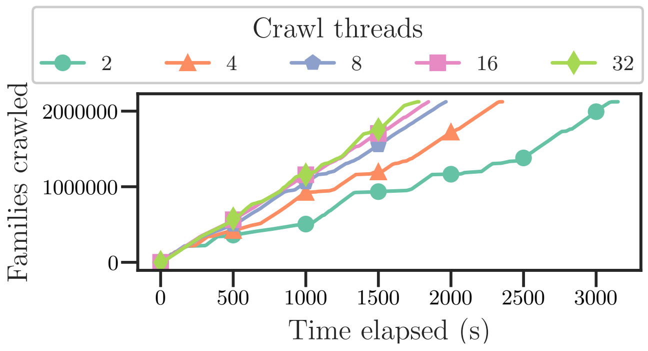

**Figure 4: Number of files crawled over time for 2, 4, 8, 16, and 32 worker threads for 2.3M files from MDF.**

$t_{gh}$, 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. Further, funcX costs can be reduced by batching extraction tasks into a single request.

## 5.4 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. We evaluate the effects of parallelizing the number of workers processing directories from the crawler's queue. We perform crawler parallelization experiments on an AWS t3.medium instance (2 vCPUs and 4 GB RAM). Figure 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. We observe minimal benefit after 16 concurrent workers, due to network congestion on the instance caused by large file lists simultaneously returning from Globus.

## 5.5 Batching

We evaluate the effects of batching in two ways (§4.3.2): **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 we try to find an optimal batching pattern by submitting 100 000 MaterialsIO tasks to the endpoint and varying both Xtract and funcX batch sizes from 2–32. We have 224 Midway workers processing these tasks. The results of this experiment are shown in Figure 5. We examine that overall throughput is maximized by extracting 8 extraction tasks per batch and sending 8–16 of these batches at a time to funcX.

## 5.6 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 make use of additional idle cloud or HPC allocations. Opportunistically offloading tasks enables Xtract to minimize the makespan

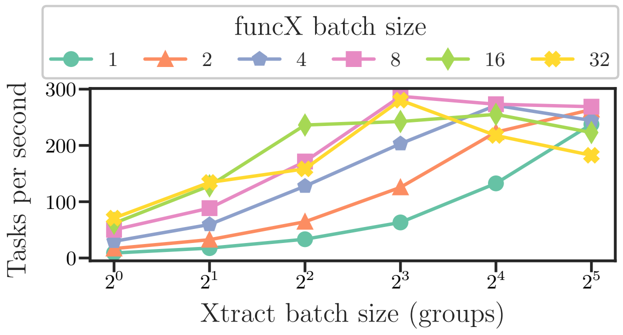

**Figure 5: Extraction tasks processed per second when varying the Xtract batch size and the funcX batch size.**

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 we measure the performance of offloading tasks to the Jetstream cloud. In particular, we 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. Further, we compare Xtract to a similar offloading setup using the same files, but instead running Apache Tika at the endpoints for metadata extraction. We configure Tika to automatically detect file type and execute the 'best' parser from its default library. We present the results of the three offloading scenarios for both extraction tools in Table 2.

We observe that there is an equilibrium point between transferring too few and too many files. In this case, if we transfer 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, we 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 2: 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. We show in Figure 6 results from

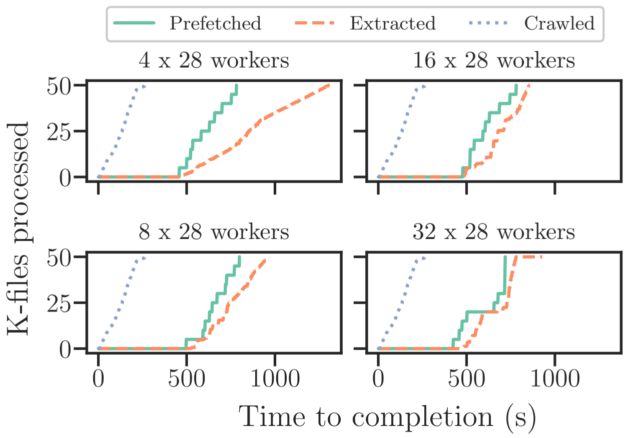

**Figure 6: Bulk metadata extraction times for an MDF subset processed on 4, 8, 16, and 32 remote Midway nodes (each running 28 workers).**

such a configuration. Specifically, we move (prefetch) 200 000 MDF files from Petrel to Midway using 10 concurrent Globus transfer jobs, and extract metadata on 4–32 Midway nodes, each with 28 workers. We see 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.

## 5.7 Min-Transfers Grouping

Xtract's FaaS compute layer effectively decouples data storage location from 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. Ideally to be effective, the min-transfers algorithm would significantly reduce transfer time in exchange for comparatively small overheads in the crawler. We 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 overlap between groups.

Figure 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. We observe that in the regular approach, 3246 of our randomly-selected families contain multiple files, leading to 20 258 files (32 GB) 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

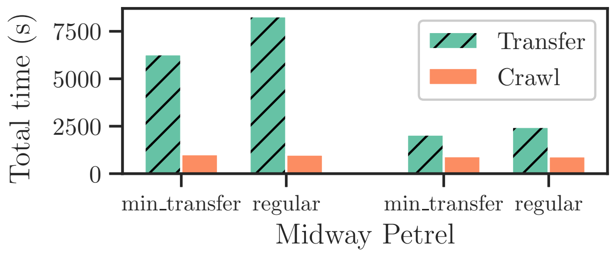

**Figure 7: Min-transfers algorithm influence on crawl and transfer times when moving data to Jetstream from the Midway2 and Petrel file systems.**

from Petrel by 16% (from 2464 to 2060 seconds at an effective transfer rate of 79 MB/s). We conclude that the min-transfers algorithm helps reduce both transfer time and redundant bytes transferred.

## 5.8 Case Studies

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

*5.8.1 MDF.* We first examine Xtract's performance on the 61 TB, 2.5 million group repository. We conduct this test using a Theta endpoint with 4096 workers. We crawl 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.

Full extraction from MDF data took 26 200 core hours and 6.4 walltime hours. Figure 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 8. Here we see 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 the Xtract process. Theta's scheduling policies allowed us to allocate nodes for only six hours at a time, less than total extraction time. Xtract could checkpoint and resubmit remaining tasks on a second allocation some time later. For this experiment we 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 re-loaded. We see in Figure 8 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,

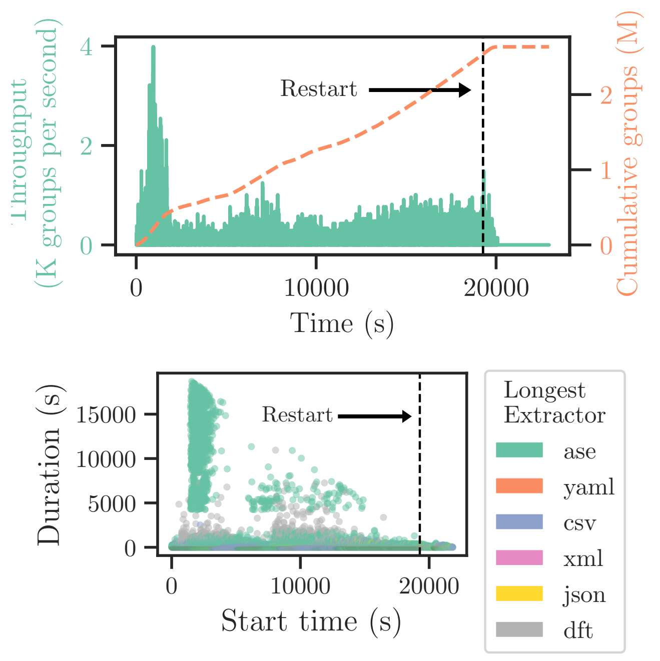

**Figure 8: Metadata extraction on all of MDF. Above: Throughput in K-groups per second (blue line) and cumulative groups processed (red line) over time. Below: Per-family extraction duration vs. start time, colored by the extractor that took the longest. In both figures, the black-dashed line at 6 hrs show when extraction was terminated and restarted from checkpoint**

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.

*5.8.2 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 decentralized extractions *without* transferring data prove impossible), we 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, we were unable to derive an associated type, so we initially treat them as free text files. Due to the absence of a 'presentation' extractor, we also treat presentations as free text files. As compute is not available on Google Drive, we configure Xtract to use 30 Kubernetes pods on River.

Table 3 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 some files are processed by multiple extractors: for example, when a text file contains both free text and tabular content.

We completed the extraction process in ~35 minutes or 23 total Kubernetes pod-hours. As each extraction plan for a file may contain up to five extractors, and because 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, we again see the benefit of being able to offload extraction to another location.

**Table 3: 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 |

## 6 RELATED WORK

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 such analyses can proceed without domain knowledge. For example, file-level deduplication [18], commonly applied in storage systems to reduce storage requirements [10], 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 [12, 20, 31, 33], is time-consuming, even if required expertise is available. In the general case, automatic methods are needed.

To alleviate these challenges researchers have developed methods to extract standard metadata from nonstandard file types and formats [26]. Further, a number of systems have been developed to automatically extract and organize metadata from files. Here we review several such systems and compare them to Xtract.

ScienceSearch [21] 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.

Apache Tika [17] is an open-source metadata extraction toolkit and library with an extensible parser interface for developing custom parsers. 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 [19] 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 [16] data management system 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. Xtract could be used by Clowder to enable distributed execution of bulk metadata extraction on distributed repositories.

Constellation [30] 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 [8].

The systems just discussed all use either entirely local or entirely centralized computation to perform extractions. Modern high-speed networks and simplifying abstractions such as FaaS make it easy to perform computation in different locations. AWS Lambda [1] is an event-driven, serverless platform for function execution that has seen wide-spread adoption in business to lower infrastructure costs. funcX [7] is a federated FaaS system designed to support distributed function execution across computing resources. AWS Snowball [2] is an industrial platform for edge computing and data migration. We leverage the ideas of remote function execution for our extractor execution to deliver a flexible metadata extraction service capable of bulk metadata extraction in distributed systems.

## 7 CONCLUSION

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 system to dispatch extractors for remote execution and the Globus research data management platform for moving data between endpoints. We have demonstrated that Xtract can scale well to 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, we showed that we can crawl the 61 TB MDF repository in just over six hours.

In future work we will extend Xtract to dynamically offload extraction tasks intelligently to heterogeneous resources based on optimization criteria such as transfer cost, resource availability, and processing speed. We will also expand the extractor library to encompass a broader range of files. To facilitate efficient file storage use, we will explore methods for identifying duplicated or nearly-duplicated data. We will also evaluate the utility of extracted metadata, so that we can explore utility-cost tradeoffs.

## ACKNOWLEDGMENTS

We gratefully acknowledge the computing resources provided and operated by the Research Computing Center at the University of Chicago; and the Joint Laboratory for System Evaluation (JLSE) and the Advanced Leadership Computing Facility (ALCF) at Argonne National Laboratory. This work was performed under financial award 70NANB19H005 from the U.S. Dept. of Commerce, National Institute of Standards and Technology as part of the Center for Hierarchical Materials Design (CHiMaD). This work is supported in part by the National Science Foundation under Grant No. 2004894.

## REFERENCES

[1] AWS Lambda. https://aws.amazon.com/lambda/. Visited Jan 20, 2021.
[2] AWS Snowball. https://aws.amazon.com/snowball. Visited Jan 20, 2021.
[3] MaterialsIO. https://github.com/materials-data-facility/MaterialsIO.
[4] W. E Allcock et al. 2019. Petrel: A Programmatically Accessible Research Data Service. In *Practice and Experience in Advanced Research Computing*. Article 49.
[5] B Blaiszik et al. 2016. The Materials Data Facility: Data services to advance materials science research. *JOM* 68, 8 (2016), 2045–2052.
[6] K Chard et al. 2014. Efficient and Secure Transfer, Synchronization, and Sharing of Big Data. *IEEE Cloud Computing* 1, 3 (2014), 46–55.
[7] R Chard et al. 2020. funcX: A federated function serving fabric for science. In *29th Intl Symp on High-Performance Parallel and Distributed Computing*. 65–76.
[8] A Das Sarma et al. 2008. Bootstrapping pay-as-you-go data integration systems. In *ACM SIGMOD Intl Conference on Management of Data*. 861–874.
[9] M Franklin et al. 2005. From databases to dataspaces: A new abstraction for information management. *ACM SIGMOD Record* 34, 4 (2005), 27–33.
[10] Q He et al. 2010. Data deduplication techniques. In *Intl Conference on Future Information Technology and Management Engineering*, Vol. 1. IEEE, 430–433.
[11] D Karger and C Stein. 1996. A new approach to the minimum cut problem. *J. ACM* 43, 4 (1996), 601–640.
[12] G King. 2007. *An introduction to the Dataverse network as an infrastructure for data sharing.* Sage Publications.
[13] P Király. 2019. *Measuring Metadata Quality.* Ph.D. Dissertation. https://doi.org/10.13140/RG.2.2.33177.77920
[14] T.-Y Lin et al. 2014. Microsoft coco: Common objects in context. In *European Conference on Computer Vision*. Springer, 740–755.
[15] R Madduri et al. 2019. Reproducible big data science: A case study in continuous FAIRness. *PLOS ONE* 14, 4 (04 2019), 1–22.
[16] L Marini et al. 2018. Clowder: Open source data management for long tail data. In *Practice and Experience on Advanced Research Computing*. 1–8.
[17] C Mattmann and J Zitting. 2011. *Tika in Action.* Manning Publications Co., USA.
[18] D Meyer and W Bolosky. 2012. A study of practical deduplication. *ACM Transactions on Storage* 7, 4 (2012), 1–20.
[19] C Quix et al. 2016. GEMMS: A generic and extensible metadata management system for data lakes. In *CAiSE Forum*. 129–136.
[20] A Rajasekar et al. 2010. iRODS primer: Integrated rule-oriented data system. *Synthesis Lectures on Inf. Concepts, Retrieval, and Services* 2, 1 (2010), 1–143.
[21] G Rodrigo et al. 2018. ScienceSearch: Enabling search through automatic metadata generation. In *14th Intl Conference on e-Science*. IEEE, 93–104.
[22] C Seltzer and I Jablokov. Distributed cloud storage. US Patent App. 14/788,618.
[23] T. J Skluzacek et al. 2019. Serverless workflows for indexing large scientific data. In *Proceedings of the 5th International Workshop on Serverless Computing*. 43–48.
[24] T. J Skluzacek et al. 2018. Skluma: An extensible metadata extraction pipeline for disorganized data. In *2018 IEEE 14th International Conference on e-Science (e-Science)*. IEEE, 256–266.
[25] C Stewart et al. 2015. Jetstream: A self-provisioned, scalable science and engineering cloud environment. In *XSEDE Conference*. 1–8.
[26] I Terrizzano et al. 2015. Data wrangling: The challenging journey from the wild to the lake. In *Conference on Innovative Data Systems Research*.
[27] J Towns et al. 2014. XSEDE: Accelerating scientific discovery. *Computing in Science & Engineering* 16, 5 (2014), 62–74.
[28] S Tuecke et al. 2016. Globus Auth: A research identity and access management platform. In *12th Intl Conference on e-Science*. IEEE, 203–212.
[29] J Ullman. 1975. NP-complete scheduling problems. *Journal of Computer and System sciences* 10, 3 (1975), 384–393.
[30] S. S Vazhkudai et al. 2016. Constellation: A science graph network for scalable data and knowledge discovery in extreme-scale scientific collaborations. In *IEEE Intl Conference on Big Data*. 3052–3061. https://doi.org/10.1109/BigData.2016.7840959
[31] D Welter et al. 2014. The NHGRI GWAS Catalog, a curated resource of SNP-trait associations. *Nucleic Acids Research* 42, D1 (2014), D1001–D1006.
[32] M Wilkinson et al. 2016. The FAIR guiding principles for scientific data management and stewardship. *Scientific Data* 3, 1 (2016), 1–9.
[33] J Wozniak et al. 2015. Big data remote access interfaces for light source science. In *2nd Intl Symposium on Big Data Computing*. IEEE, 51–60.