RIPPLE: Home Automation for Research Data Management

Ryan Chard[1], Kyle Chard[2], Jason Alt[3], Dilworth Y. Parkinson[4], Steve Tuecke[2], and Ian Foster[2]

[1]Computing, Environment, and Life Sciences, Argonne National Laboratory
[2]Computation Institute, University of Chicago and Argonne National Laboratory
[3]National Center for Supercomputing Applications
[4]Advanced Light Source Division, Lawrence Berkeley National Laboratory

***Abstract*—Exploding data volumes and acquisition rates, plus ever more complex research processes, place significant strain on research data management processes. It is increasingly common for data to flow through pipelines comprised of dozens of different management, organization, and analysis steps distributed across multiple institutions and storage systems. To alleviate the resulting complexity, we propose a home automation approach to managing data throughout its lifecycle, in which users specify via high-level rules the actions that should be performed on data at different times and locations. To this end, we have developed RIPPLE, a responsive storage architecture that allows users to express data management tasks via a rules notation. RIPPLE monitors storage systems for events, evaluates rules, and uses serverless computing techniques to execute actions in response to these events. We evaluate our solution by applying RIPPLE to the data lifecycles of two real-world projects, in astronomy and light source science, and show that it can automate many mundane and cumbersome data management processes.**

I. INTRODUCTION

Researchers are faced with an increasingly complex data landscape in which data are obtained from a number of different sources (e.g., instruments, computers, published data), stored in disjoint storage systems, and analyzed on an area of high performance and cloud computers. Given the increasing speed at which data are produced, combined with increasingly complex scientific processes and the requisite data management, munging, and organization activities required to make sense of data, researchers are faced with new bottleneck in the discovery process. Improving data lifecycle management practices is essential to enhancing productivity, facilitating reproducible research, and encouraging collaboration [1]. Yet current practices are typically manual and ad hoc, requiring considerable human effort and ensuring little adherence to best practices. As a result, researchers struggle to manage, analyze, and share data reliably and efficiently [2] and research results are frequently irreproducible. We posit that researchers require automated methods for managing their data such that tedious and repetitive tasks (e.g., transferring, archiving, and analyzing) are accomplished without continuous user input.

Automated approaches have revolutionized many domains such as machinery use in factories, aircraft flight, and more recently managing devices within the home. Home automation in particular shares similar features to research data management: as it is focuses on controlling and automating a range of devices within a home such as lighting, heating, security, and other appliances. The main goal of these systems is to increase convenience and decrease time spent on mundane tasks by automating repetitive processes, such as turning on lights when it gets dark, setting security alarms when leaving the house, and controlling room temperatures based on weather conditions. Home automation systems enable users to finely customize and control their environments by defining policies that dictate how household appliances should perform under different circumstances. It is these same types of capabilities that are needed for managing research data.

In this paper we present a new approach to data management called RIPPLE. RIPPLE aims to allow researchers, lab managers, and administrators to define data management practices as a set of simple if-trigger-then-action recipes. Actions, such as moving data or executing an analysis script, are triggered in response to events, such as files being created, modified, or deleted. Filesystem events are captured through various APIs, including Linux's inotify and the Globus API [3]. Given the broad range of actions that might be possible RIPPLE builds upon severless computing systems to enable on-demand processing of recipes. In particular, using Amazon Web Services Lambda as a scalable and low-latency solution for performing arbitrary, loosely coupled actions.

To guide and evaluate our approach we focus on two use cases: the data management processes associated with the Large Synoptic Survey Telescope (LSST) and a multi-institutional materials science project. We show that RIPPLE can satisfy the needs of these two projects by automating important data management tasks. Furthermore, we evaluate the scalability and performance of our prototype implementation by analyzing event collection and processing operations.

The rest of this paper is organized as follows. Section II discusses related work. We describe the LSST and materials science scenarios in Section III. We present RIPPLE in Section IV. We evaluate RIPPLE's performance in Section V and its ability to meet application requriements in Section VI. We summarize in Section VII.

II. RELATED WORK

Previous rule-based approaches to data management [4] are primarily designed for expert administration of large data

stores. Our approach is differentiated by its simple recipe notation, decoupling of rules from data management and storage technologies, and use of serverless computing to evaluate rules. Below we discuss several rules engines and discuss how they relate to our work.

The integrated Rule-Oriented Data System (iRODS) [5] uses a powerful rules engine to manage the data lifecycle of the files and stores that it governs. iRODS is a closed system: data are imported into an iRODS data grid that is managed entirely by iRODS and administrators use rules to configure the management policies to be followed within that data grid. In contrast, RIPPLE is an open system: any authorized user may associate rules with any storage system. Both approaches have their place in the data landscape. iRODS has been used successfully in large projects. Whereas RIPPLE aims to benefit the dynamic, heterogeneous, multi-project environments that typify many modern research labs.

IOBox [6] is designed to extract, transform, and load data into a catalog. It is able to crawl and monitor a filesystem, detect file changes (e.g., creation, modification, deletion), and apply pattern matching rules against file names to determine what actions (ETL) should be taken. RIPPLE extends this model by allowing scalable and distributed event detection, and supporting an arbitrary range of actions.

The Robinhood Policy Engine [7] is designed to manage large HPC filesystems. Although it can support any POSIX filesystem, it implements advanced features for Lustre. Robinhood maintains a database of file metadata. It allows bulk actions to be scheduled for execution against sets of files. For example, migrating or purging stale data. Robinhood provides routines to manage and monitor filesystems efficiently, such as those used to find files, determine usage, and produce reports. It is not the goal of RIPPLE to provide such utilities. Instead, RIPPLE is designed to empower users to implement simple, yet effective data management strategies.

SPADE [8] supports automated transfer and transformation of data. Users configure a SPADE *dropbox*. If a file is written to the dropbox, SPADE creates (or detects) an accompanying *semaphore file* to signal that the file is complete and that a transfer should begin. SPADE can also enable data archival or execution of analysis scripts in response to data arrival. The SPOT framework [9] is a workflow management solution developed specifically for the Advanced Light Source (ALS). SPOT leverages SPADE to automate the analysis, transfer, storage, and sharing, of ALS users' data using HPC resources. The framework includes a Web interface to provide real-time feedback. In contrast to SPOT's pre-defined flows which handle very large data volumes and numbers of data sets, RIPPLE empowers non-technical users to define custom recipes which can be combined into adaptive flows.

III. SCENARIOS

We base the design and implementation of RIPPLE on the data management requirements of two large research projects: the LSST and X-ray science at the ALS. Each provides a unique set of data management requirements and demonstrates the flexibility of our solution. Here we briefly describe these projects and their requirements.

A. Large Synoptic Survey Telescope

The LSST is a wide-field telescope currently under construction in Chile. It uses a new kind of telescope to capture panoramic, 3.2-gigapixel, snapshots of the visible sky every 30 seconds. It is expected to produce more than 30 terabytes of data every night. Over the LSST's 10 year lifetime, it will map billions of stars and galaxies. These data will be used to explore the structure of the Milky Way and assist in the exploration of dark energy and dark matter.

The computational and data management requirements of the LSST are substantial [10]. The project requires near real-time detection of "interesting" events, such as supernovae and asteroid detection. In addition, all image data will be analyzed and made available to the public. To meet the storage needs of this project two data centers (called *custodial stores*), located in Chile and Illinois, are being readied to reliably store the data generated by the telescope.

The custodial stores have replication, recovery, and analytical actions needs that must be reliably enforced and could be automated. For example, data must be immediately transferred from the telescope and archived in both stores to provide fault tolerance; data must be cataloged and made discoverable once it enters a store such that scientists can later use the data for analysis; and corrupted or deleted data must be recovered from another store.

B. X-ray science at the ALS

The ALS is a DOE-funded synchrotron light source housed at Lawrence Berkeley National Laboratory. It is one of the brightest sources of ultraviolet and soft x-ray light in the world. Given its unique characteristics, scientists from many fields use the ALS to conduct a wide array of experiments including spectroscopy, microscopy, and diffraction. The ALS is comprised of almost 40 beamlines that serve approximately 2,000 researchers each year.

The ALS is representative of the data management lifecycle of many large instruments and research facilities. The data generated by a beamline are large, generated frequently, and requires substantial computational resources to analyze in a timely manner. Moreover, researchers using each beamline are granted short allocations of time in which to conduct their experiments. They typically run a number of experiments, collect a large amount of data, and then analyze those data at a latter point in time using large-scale resources.

In some cases, analyses are conducted throughout the experiment to guide the experimental process. Typically, data are transferred to a compute resource where a variety of quality control procedures and analysis algorithms are executed. In most cases, analysis requires data transformations, parameter and configuration selection, and creation of a batch submission file for execution. Upon completion, analysis output is then transferred back to the researchers.

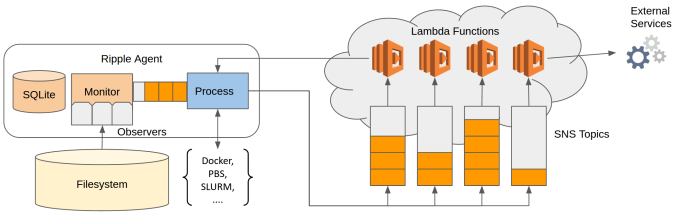

Fig. 1: RIPPLE architecture. Filesystem events are captured and evaluated against registered recipes. Lambda functions evaluate recipes and execute actions.

IV. RIPPLE

RIPPLE is comprised of a cloud-hosted service and a lightweight agent that can be deployed on various filesystems. An overview of RIPPLE's architecture is shown in Fig. 1. The local agent includes a SQLite database, a monitor service, a message queue, and a processing element for executing actions (in containers) locally. The SQLite database stores a local copy of all active recipes. The monitor, based on the Python module Watchdog, captures local filesystem events. The basic evaluation process compares each event against all filter conditions defined by active recipes. When an event matches a particular recipe, the event is passed to the processing component via a reliable message queue.

The processing component of the agent publishes triggered events to an Amazon Simple Notification Service (SNS) topic. The SNS topic triggers the invocation of a Lambda function. Lambda functions are used to evaluate the event and recipe, and then to manage the execution of actions.

RIPPLE supports three types of actions: 1) execution of processes on the local node; 2) invocation of external APIs; and 3) execution of lightweight Lambda functions. In each case RIPPLE uses a Lambda function to invoke the action. Execution on the local machine is achieved by initiating local Docker containers or submitting subprocess procedures.

A. Recipes

The flexibility of RIPPLE recipes enables the expression of a wide range of custom functions. To make our solution as accessible as possible, we have employed a simple if-trigger-then-action style definition of recipes. This representation allows even non-technical users to create custom programs. The usability of the trigger-action programming model has been proven by the If-This-Then-That (IFTTT) [11] service. Users have created and shared hundreds of thousands of IFTTT recipes [12] with vastly different functions, such as notifying of predicted weather events and automatically extracting and storing of images from Facebook.

A RIPPLE recipe is represented as a JSON document comprised of a "trigger" event and an "action" component. The trigger specifies the condition that an event must match in order to invoke the action. The action describes what function is to be executed in response to the event. An example recipe can be seen in Listing 1. In this recipe, the trigger defines the source of the event as the local filesystem; the type of event as file creation; and a path and regular expression describing conditions to execute the action (any file in the */path/to/monitor/* directory with a *.h5* extension). The action describes the service to invoke (in this case, globus); the operation to execute (transfer); and arguments for performing the operation. Target modifiers allow actions to be performed on files that did not raise the triggering event. In the given example, the target implies the operation should be performed on the file raising the event.

Listing 1: An example RIPPLE recipe.

```
"recipe":{
  "trigger": {
    "username": "ryan",
    "monitor": "filesystem",
    "event": "FileCreatedEvent",
    "directory": "/path/to/monitor/",
    "filename": ".*.h5$"
  },
  "action": {
    "service": "globus",
    "type": "transfer"
    "source_ep": "endpoint1",
    "dest_ep": "endpoint2",
    "target_name": "$filename",
    "target_match": "",
    "target_replace": "",
    "target_path": "/~/$filename.h5",
    "task": "",
    "access_token": "<access token>"
  }
}
```

B. Event detection and evaluation

RIPPLE relies on a flexible event monitoring model that can be used in different environments. Specifically, it uses the Python Watchdog module which offers multiple observers to detect filesystem events. Watchdog's observers include: Linux inotify, Windows FSMonitor, Mac OS FSEvents and kqueue, OS-independent polling (periodic disk snapshots), and support for custom event sources via third party APIs (e.g., the Globus Transfer API). The scope for a Watchdog observer is determined by the paths specified during its creation. RIPPLE evaluates all active recipes to determine filesystem paths of interest and collapses these paths to instantiate a number of observers to monitor them.

Multiple events can occur in response to an individual action on a filesystem. For example, the action of creating a file can cause `inotify` to raise events concerning the file's creation, modification, and closure, as well as the parent directory's modification. Thus it is crucial to filter events and minimize the overhead caused by passing irrelevant events to the RIPPLE service. The agent's filtering phase removes superfluous events by reporting only those that match active recipe conditions.

RIPPLE can also respond to events created by external services. For example, we have implemented a Watchdog observer that periodically polls the Globus Transfer service, checks for successful transfer operations (file movement, creation, deletion), and raises appropriate events. The events generated by this observer are modeled in the same way as filesystem events and are processed by the monitor.

C. Actions

RIPPLE aims to support an arbitrary set of extensible actions that may be invoked as a result of an event. The initial set of actions have been developed to address the requirements of the scenarios discussed in Section III. Specifically, RIPPLE currently supports execution of Lambda functions, external services (via Globus and AWS services), and containers located on the storage system. As part of the recipe definition users are able to specify additional state to be passed to the action for execution: for example, where data should be replicated or the HPC queue to use for execution.

Lambda functions provide the ability to execute arbitrary code, typically, for short periods of time. By supporting such functions, users can define complex actions that are executed within their own Amazon context as a result of a recipe. This is achieved by specifying a Lambda function's Amazon Resource Number (ARN) as the action of a recipe.

External services: RIPPLE supports two general services: Globus and Amazon Simple Messaging Service (SMS). Globus allows users to transfer, replicate, synchronize, share, and delete data from arbitrary storage locations. Globus actions can be configured to use any of these capabilities. To do so, we have developed a Lambda function that authenticates with Globus (using a predefined token included in the recipe definition) and executes the appropriate action using the Globus API. Depending on the operation to be performed information such as the source or destination endpoint is included in the recipe definition. In addition to transferring data, the same Lambda function is also capable of initiating Globus delete commands and constructing Globus shared endpoints—a means to securely share data between collaborators.

RIPPLE can send emails to notify users of events such as new data being placed in an archive or data being deleted. This capability is based on integration with Amazon's Simple Messaging Service (SMS) and again uses a Lambda function to perform this operation using a customizable destination email address.

Local execution: There are many scenarios in which the desired action of a rule is to perform an operation on a local file directly. RIPPLE provides this support in one of two ways, either by running a Docker container, or by initiating a subprocess procedure. Docker enables execution encapsulation such that it is reliably and securely executed on arbitrary endpoints. Examples of supported actions include extracting metadata, creating a persistent identifier for a dataset, and compressing data prior to archival. However, Docker is not always suitable due to its need for administrator privileges. For example, in situations where users do not have permission to run docker containers (such as the HPC resources used by the ALS scenario) we instead leverage Python subprocess commands to execute different scripts and applications. Examples of RIPPLE's subprocess execution include modifying files using Linux commands, creating batch submission files, and dispatching jobs to a supercomputer execution queue.

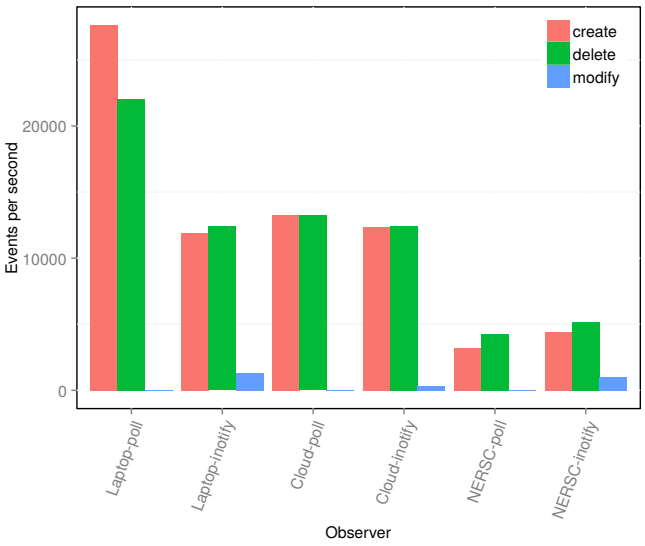

Fig. 2: Number of events processed per second on different machines with the inotify and polling observers.

V. EVALUATION

We explore the performance and scalability of the RIPPLE system from three perspectives: event detection, event filtering, and execution of actions when Amazon Lambda functions are in different states of readiness.

A. Event Detection

We deployed RIPPLE over three machines: a personal laptop, a c4.xlarge AWS EC2 instance, and a supercomputer login node (NERSC's Edison). RIPPLE's event detection rate has been determined for each of these machines by timing how long it takes for 10,000 events to be detected. To minimize overhead we disable RIPPLE's filtering (rule condition matching) capabilities and simply count the events that are detected. Fig. 2 shows the performance of two distinct event observation methods: inotify and polling. In each experiment, we first established the observer and then created 10,000 files in a monitored directory, touched each file to raise a modification event, and deleted each file to raise a deletion event. The results show that the personal laptop and EC2 instance are capable of detecting more than 10,000 events per second with both the polling and inotify observers. We note that modification events are not detected as reliably as creation and deletion events with either observer. This is shown in the figure as less than 1000 modification events are recorded when 10,000 files were modified in the space of 0.2 seconds. These results are expected when using the polling observer as it is configured to take snapshots just once a second. The NERSC experiments demonstrate the lowest event throughput as they were conducted on a large networked file system.

B. Filtering Cost

In order to understand the overhead incurred by the RIPPLE agent filtering events we investigated the rate at which events

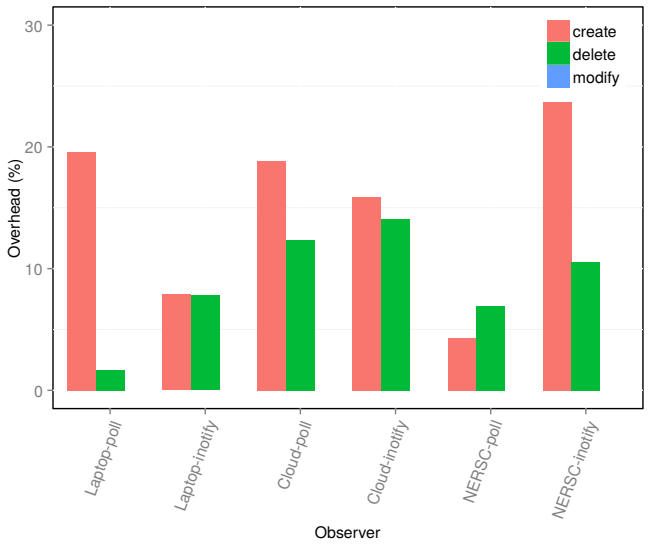

Fig. 3: Event filtering overhead.

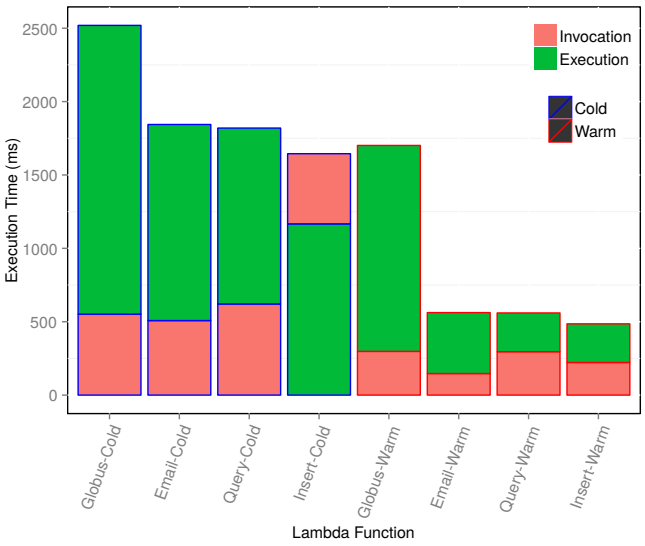

Fig. 4: Execution and invocation time of various Lambda functions in warm and cold states.

are detected when filtering is conducted. Fig. 3 shows the overhead incurred by the filtering process. The figure shows that the overhead is typically largest for creation events. This is due to the filters enabled in the experiment. The experiment employs a two-step filtering process where each event is first evaluated to see if it is of type *create* and if successful, the filename is compared to a condition. The overhead caused by filtering is negligible for modification events due to the limited detection rate.

C. Lambda Performance

We explore the performance of four distinct Lambda functions which perform one of the following tasks: initiate a Globus transfer, send an email, log data in a database, and query a database. Lambda functions are said to be in a *cold* state if they are first started in response to an invocation. Following an invocation the function becomes *warm*, or cached. Fig. 4 shows the average execution time for each of the four Lambda functions in both cold and warm states. The invocation time is computed as the difference between the reported time a request is placed in a SNS queue and the time that the Lambda function starts. The execution time is the reported duration of the Lambda function's execution. The results show a significant overhead incurred by cold functions and that invoking Globus actions requires substantially more time than AWS services. This is in part due to the requirement of the Lambda function to import the globus-sdk module.

VI. Use Cases

To explore Ripple's ability to meet application requirements we have deployed testbeds of the scenarios discussed in Section III. In each case we have deployed Ripple agents and developed a suite of recipes to automate their respective data lifecycles.

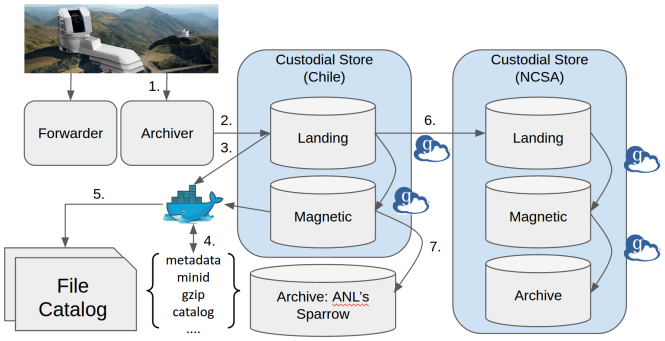

Fig. 5: LSST testbed and automated data lifecycle.

A. Ripple and LSST

To represent the LSST scenario we deployed a testbed with three AWS instances and a Sparrow, a Blackpearl tape storage system at Argonne National Laboratory. The three AWS instances represent the observatory and two custodial stores (Chile and NCSA). Within each instance, we created multiple Globus shared endpoints to represent different storage facilities. For example, the instance representing the observatory has shared endpoints mounted at */telescope* and */archiver*. The custodial storage instances mount shared endpoints to represent the landing zone, lower performance storage (magnetic), and archival tape storage. The instance representing the Chilean store uses Argonne's Sparrow, to archive data. Each instance has an individual Ripple agent deployed and is configured with recipes to monitor the local shared endpoints. Finally, to represent the LSST file catalog we use an AWS RDS database that manages information about each file stored in the custodial stores.

Fig. 5 provides an overview of the testbed and the associated data lifecycle. The data flow is initiated by (1) the telescope

generating an image, typically stored in a *FITS* format, which is then placed at the archiver. RIPPLE filters filesystem events within the archiver directory for those with the FITS extension. Creation of a file in this directory triggers a recipe to (2) perform a high-speed HTTPS data upload to the Chilean custodial store. Once data arrive in the custodial landing zone, (3, 4) recipes are triggered that launch local Docker containers to perform metadata extraction and cataloging for each new file. Metadata regarding the file are placed in a local JSON file so that Globus' search capabilities can index the file. Additionally, a unique identifier for the file is generated using the Minid [13] service before being (5) inserted into the file catalog. The data are then (6) automatically synchronized to the NCSA custodial store where similar metadata processing occurs. The use of the Minid service allows us to use the same unique identifier for the file regardless of location, meaning the file catalog is consistent between stores. As data are propagated down the storage tiers (7) within each custodial store, the metadata and file catalogs are dynamically updated. Prior to archiving the data, a RIPPLE recipe triggers file compression; creation of the compressed (gzip) file triggers the final recipe that transfers the file to Sparrow.

Recovery and consistency are of utmost importance for LSST as files cannot be recreated. The LSST testbed has been instrumented with RIPPLE recipes to detect the deletion and modification of FITS files. Any file that is found to be deleted or corrupted is automatically replaced by a copy from the other custodial store.

B. RIPPLE *and ALS*

The ALS testbed has been constructed by deploying a RIPPLE agent on both an ALS machine and a NERSC login node. For exploratory purposes, we were given access to a heartbeat application to reproduce data being generated from an ALS beamline. We implemented recipes to manage the data lifecycle of beamline datasets from creation, through execution, and finally share the results with specific collaborators.

Once the heartbeat application finishes writing results to an HDF5 file a new file, which signifies the process's completion, is created. The system first detects the subsequent file creation and initiates a transfer of the HDF5 output to NERSC. On arrival at NERSC (detected by the Globus Transfer API observer) a recipe ensures that a metadata file and a batch submission file are dynamically created using the input. The creation of the batch submission file triggers the workload to be submitted to the Edison supercomputer's queue. Once the workload completes the output file is detected and is transferred back to the ALS machine. On arrival at the ALS a shared endpoint is created in order to expose the resulting dataset to a set of collaborators. Finally, an email notification is sent to inform collaborators of the new data and directs them to the shared endpoint.

VII. CONCLUSION

RIPPLE aims to simplify the management of complex data lifecycles. In the same way that home automation systems simplify and automate tedious tasks associated with managing a large number of home appliances, RIPPLE supports automated actions in response to various events. In this paper we described how RIPPLE can achieve these goals and investigated the scalability challenges associated with deploying such a service in practice. We showed that RIPPLE can be used to automate two real-world workflows that manage data in two large scientific scenarios. These examples used RIPPLE to automate data transfer, replication, cataloging, recovery, HPC execution, archival, and sharing.

In future work we aim to apply RIPPLE to additional scientific use cases to gather requirements and generalize our model. We ultimately aim to integrate RIPPLE in the Globus platform, enabling thousands of users to create custom data flows. We will also investigate developing a programming model, inspired by IFTTT, to simplify the definition of flows.

ACKNOWLEDGMENTS

This work was supported in part by the U.S. Department of Energy under contract DE-AC02-06CH11357.

REFERENCES

[1] D. Atkins, T. Hey, and M. Hedstrom, "National Science Foundation Advisory Committee for Cyberinfrastructure Task Force on Data and Visualization Final Report," National Science Foundation, Tech. Rep., 2011.

[2] J. P. Birnholtz and M. J. Bietz, "Data at work: Supporting sharing in science and engineering," in *International ACM SIGGROUP Conference on Supporting Group Work*. ACM, 2003, pp. 339–348.

[3] K. Chard, S. Tuecke, and I. Foster, "Efficient and secure transfer, synchronization, and sharing of big data," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 46–55, 2014.

[4] A. Rajasekar, M. Wan, R. Moore, and W. Schroeder, "A prototype rule-based distributed data management system," in *HPDC workshop on Next Generation Distributed Data Management*, vol. 102, 2003.

[5] A. Rajasekar, R. Moore, C.-y. Hou, C. A. Lee, R. Marciano, A. de Torcy, M. Wan, W. Schroeder, S.-Y. Chen, L. Gilbert, P. Tooby, and B. Zhu, "iRODS Primer: Integrated rule-oriented data system," *Synthesis Lectures on Information Concepts, Retrieval, and Services*, vol. 2, no. 1, pp. 1–143, 2010.

[6] R. Schuler, C. Kesselman, and K. Czajkowski, "Data centric discovery with a data-oriented architecture," in *1st Workshop on The Science of Cyberinfrastructure: Research, Experience, Applications and Models*, ser. SCREAM '15. New York, NY, USA: ACM, 2015, pp. 37–44.

[7] T. Leibovici, "Taking back control of HPC file systems with Robinhood Policy Engine," *arXiv preprint arXiv:1505.01448*, 2015.

[8] "SPADE." [Online]. Available: http://nest.lbl.gov/projects/spade/html/. Visited March, 2017

[9] J. Deslippe, A. Essiari, S. J. Patton, T. Samak, C. E. Tull, A. Hexemer, D. Kumar, D. Parkinson, and P. Stewart, "Workflow management for real-time analysis of lightsource experiments," in *9th Workshop on Workflows in Support of Large-Scale Science*. IEEE Press, 2014, pp. 31–40.

[10] M. Jurić, J. Kantor, K. Lim, R. H. Lupton, G. Dubois-Felsmann, T. Jenness, T. S. Axelrod, J. Aleksić, R. A. Allsman, Y. AlSayyad *et al.*, "The LSST data management system," *arXiv preprint arXiv:1512.07914*, 2015.

[11] "If This Then That," http://www.ifttt.com. Visited March, 2017.

[12] B. Ur, M. Pak Yong Ho, S. Brawner, J. Lee, S. Mennicken, N. Picard, D. Schulze, and M. L. Littman, "Trigger-action programming in the wild: An analysis of 200,000 IFTTT recipes," in *CHI Conference on Human Factors in Computing Systems*. ACM, 2016, pp. 3227–3231.

[13] K. Chard, M. D'Arcy, B. Heavner, I. Foster, C. Kesselman, R. Madduri, A. Rodriguez, S. Soiland-Reyes, C. Goble, K. Clark, E. W. Deutsch, I. Dinov, N. Price, and A. Toga, "I'll take that to go: Big data bags and minimal identifiers for exchange of large, complex datasets," in *IEEE International Conference on Big Data*, Washington, DC, USA, 2016.