# Sciunits: Reusable Research Objects

Dai Hai Ton That, Gabriel Fils, Zhihao Yuan, Tanu Malik
School of Computing, DePaul University, Chicago, IL, USA
Email: dtonthat,gfils1,zhihao.yuan,tmalik1@depaul.edu

*Abstract*—**Scientists often need to share their work. Typically, their data is shared in the form of Uniform Resource Identifiers (URIs) or Digital Object Identifiers (DOIs). Scientists' work, however, may not be limited to data, but can also include code, provenance, documents, etc. The Research Object has recently emerged as a method for the identification, aggregation, and exchange of this scholarly information on the Web. Several science communities now engage in a formal process to create research objects and share them on scholarly exchange websites such as Figshare or Hydroshare, but often sharing is not sufficient for scientists. They need to compute further on the shared information. In this paper, we present the sciunit, a reusable research object whose contents can be re-computed, and thus measured. We describe how to efficiently create, store, and repeat computational work with sciunits. We show through experiments that sciunits can replicate and re-run computational applications with minimal storage and processing overhead. Finally, we provide an overview of sharing and reproducible cyberinfrastructure based on sciunits, increasingly being used in the domain of geosciences.**

## I. Introduction

Recent requirements of scholarly communication increasingly emphasize the reproducibility of scientific claims. Given the computational nature of science, text-based research papers are considered poor mediums to establish reproducibility. Papers must be accompanied by "research objects"— aggregation of digital artifacts such as code, data, scripts, and temporary experiment results — that together with the paper provide an authoritative and far more complete record of a piece of research.

Toward the goal of reproducibility, several tools have been proposed to help researchers create research objects from a variety of digital artifacts. [1] provides a comprehensive list of these tools. Created research objects can be easily shared on websites that disseminate scholarly information, such as Figshare [2]. Once shared, however, the extent of reuse is often subject to the amount of accompanying documentation. If scanty, research objects may go unused. Research objects themselves, once created, provide little computational guidance to users as to how they should be reused, beyond simply downloading them and browsing their contents.

In order for a research object to include computational guidance for reuse, it must be created and maintained quite differently than a research object created merely for sharing. We provide two such distinctions using an example. Consider a typical research paper with an analysis based on large amounts of code and data, and assume that the researcher authoring the paper has used the code and data to conduct a number of experiments that produce the paper's target figures and results.

The example paper's digital artifacts relating to its experiments may be bundled together in a medium such as a file archive (.tar), compressed file format (.gz), virtual image, or container. A shared research object is free to use any of these mediums. A reusable research object, however, must use a virtual image or container, since it must produce a "computational research object" that, when downloaded and shared, will guarantee an instantly-executable unit of computation.

Metadata interspersed throughout the example project's written analysis and its code and data can take many forms, including annotations, version information, and provenance. A shared research object's metadata usually serves a purely informational purpose, and is seldom used literally in the paper's experiments. A reusable research object, however, utilizes literal metadata by directly linking it to the code and data of the experiments. In particular, provenance, if collected in standard form, can guide different forms of reusable analysis—exact, partial, or modified reuse. Keywords and annotations can provide reference to additional datasets for modified reuse. In other words, a reusable research object can execute conditionally based on its embedded metadata, instead of simply including it as a stand-alone digital artifact that requires more interpretive labor to reason about and reuse.

In this paper, we describe the *sciunit*, a reusable research object that has a lifetime beyond sharing. The sciunit does not simply bundle digital artifacts, but includes computational guidance that allows users to distinguish purely informational digital artifacts from reusable digital artifacts. The reusable artifacts are stored in containers. Similar to shared research objects, users can attach additional annotations to describe containers. Each container also incorporates associated provenance, and users can use the included provenance to create smaller containers or repurposed containers (i.e. they can create arbitrarily new containers). These containers enable exact or partial repeatability of the sciunit.

We use application virtualization (AV) methods to automatically create a container of an executable application. In AV, operating system calls during application execution are traced to automatically copy all binaries, data, and software dependencies into a container. The resulting container is portable and instantly reusable in that it can be run on any compatible machine[1] without installation, configuration, or root permissions.

While application virtualization facilitates more widely-reusable research objects, one implication of using this method

[1]our method, like most other AV methods, is OS-specific

is that as multiple containers are bundled together into a single sciunit, space consumption grows substantially due to the duplication of digital artifacts, such as system dependencies, common binaries, or even common data files in other containers. A large digital artifact within two containers, but that changes only slightly in content, will still consume its full amount of space in each container. Additionally, two slight variations of the same container will bundle copies of common dependencies into each container. This growing space is particularly of issue when a user shares different versions of an analysis or pipeline. We show how multiple containers can be stored efficiently in one sciunit using a common block-based storage based on content deduplication techniques [3].

When a sciunit embedded with several instantly-reusable containers is shared, guidance for using the containers must also be provided. Included provenance information can help: it can provide an overview of the overall workflow of the container. However, if AV techniques are used to create the container, the associated provenance information is usually at the file and process level, making it difficult to quickly absorb. We describe a technique that composes a graph temporally and presents an intuitive spatial summary of a container's provenance information. The user can use parts of this graph to create smaller containers that can be independently reused and repurposed.

This paper makes the following contributions:

- We propose the sciunit, a reusable research object that improves the conduct of scholarly communication. We present a Python/C-based tool for creating, sharing, and reusing sciunits.
- We describe application virtualization method to build a container and show how it can be associated with a sciunit for sharing and reuse.
- We show how content deduplication methods can be used to efficiently store multiple containers in a single sciunit.
- We create an interactive provenance visualization that allows the user to understand how containers within a sciunit were built, and use the visualization for repeating the container partially or modifying it.

The rest of the paper is organized as follows:

- Section II: evolution of research objects, and their creation and use in related applications.
- Section III: overall architecture of our work.
- Section IV: creating a container with embedded provenance using application virtualization.
- Section V: storing multiple containers in a single sciunit
- Section VI: utilizing provenance within sciunits to repeat and reproduce analysis.
- Section VII: optimizing the provenance graph.
- Section VIII: detailed experimental analysis.
- Section IX: conclusions.

## II. Evolution of Research Objects

Research objects are increasingly seen as the new social object for advancing science [4]; research papers are mostly used only for the dissemination of scholarly work, measuring research impact, and assessing credit and attribution [5]. The Research Object Model [6] is the most comprehensive standard defining the concept of a research object as a bundle of artifacts that provides the digital record of a piece of research. The research object, when shared, serves as the proof of a research outcome by supplying a more complete record of the scientific claims made in the research paper. [7]. Due to its academic origin, implementations of the standard have primarily focused on structured workflow objects [8], [9], [10], and have not yet encompassed general applications, *i.e.*, application executed without a formal workflow system. In this paper, we describe the sciunit client, a tool for creating a research object that includes containers created during run-time execution of an application.

To create a research object, digital artifacts must be placed within it either manually with explicit commands such as those used in RO-Manager [11] (a tool that uses the RO-Bundle specification),[12] or automatically by using an application virtualization tool such as Code, Data, and Environment (CDE) ([13], [14]) that containerizes an application as it executes. In this paper, we have chosen the application virtualization tool Provenance-To-Use (PTU) [15], [16], which is built on top of CDE, to automatically capture provenance while creating containers. We exclude more recent methods such as [17] that necessitate a user to learn a new language, and instead focus on the integration of devops tools in research objects.

Toward this we enhance PTU in two specific ways. First, we extend it to include other research object digital artifacts such as authoring information (files, annotations, PDFs) and versioning information, conforming it the notion of a research object that can be defined with more emerging standards such as PAV [18] or OntoSoft [19] . Second, we give PTU the ability to store multiple containers efficiently in the same research object. While we haven't described sciunits in terms of the PAV or OntoSoft standard, the objective of this paper is to determine how such a research object can be computationally created, maintained, and reused for scientific experiments.

Recorded provenance can be made more conducive to new analyses by summarizing it using statistical [20] and non-statistical methods [21], [22]. Our sciunit client uses non-statistical methods [23] to summarize a research object's provenance and extends the methods to visualize the summarized provenance both spatially and temporally. We show how our sciunit uses summarized provenance to build modified containers that can be re-executed and analyzed.

Other methods to build and reuse containers such as Topology and Orchestration Specification for Cloud Applications [24] still rely on user creating the topology, relationship, and node specifications, which are eventually translated to Dockerfiles [25]. In our case, Docker is merely a wrapper for standardization since application virtualization creates a self-contained container and the translation to Dockerfiles from the collected provenance is fairly straightforward.

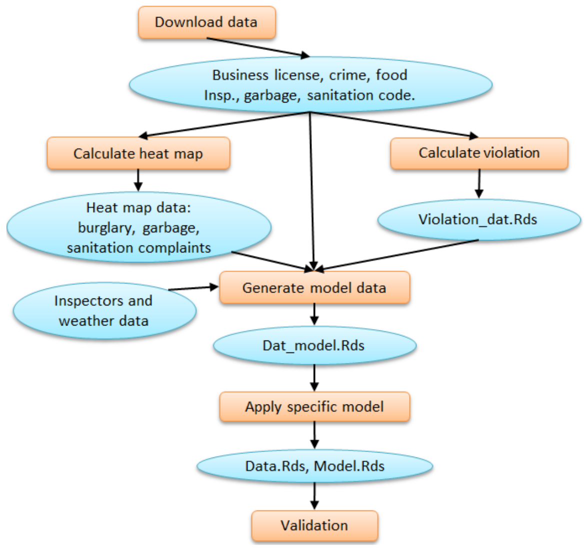


Fig. 1. A conceptual view of the steps required to run the Food Inspection Evaluation [26] predictive model.

```
1. > geounit start FIE
2. > annotate FIE author:Tanu Malik
3. > track FIE.sh 1 /default/data
4. > package FIE.sh 1 /default/data
package_hash = 04er667
5. > publish 04er667
enter abstract: FIE package
title: Food Inspection Evaluation
keywords (split with comma): food, FIE
do you want to make this resource
public (y/n):y
resource was created
6. > repeat 04er667
7. > repeat remote 04er667
8. > stop
```

Fig. 2. User interaction with sciunit client

## III. Architecture and Use of Sciunits

Our reference implementation is a client program that creates, stores, and executes reusable research objects. The client and accompanying server-side infrastructure that stores and manages sciunits forms a reproducible infrastructure, currently in use within the geosciences domain in the United States (http://geotrusthub.org). We use a real-world example from this hub to highlight the architecture and design of the sciunit client. We have intentionally kept the example small to be presentable in the given amount of space.

Figure 1 shows a real-world example of a predictive model used for forecasting critical violations during sanitation inspection [26]. The software consists of scripts written in different languages (R, Python, and Shell) that operate on input datasets acquired from the City of Chicago Socrata data portal [27]. The output of the predictive model is continually tested using a double-blind retrodiction; The Department of Public Health conducts inspections via its normal operational procedure, which are compared with the output of the model. The analysis is published here [28], the pre-processing code is shared on GitHub [29], and the data is available on public repositories [27]. Bundling these artifacts into a shared research object would simply aggregate them into one package, and would likely be inefficient given data from nine different sources, which changes periodically, making analysis conducted within a certain time range obsolete. A reusable research object, alternatively, would only consist of the identifiers of one or more re-executable containers, along with other listed digital artifacts. Given a container, the user would be able to exactly repeat the analysis for the given time frame as done by a data scientist originally, and verify with the inspection data.

Our Python/C command-line interface (CLI) client is used to build sciunits. Figure 2 shows a sample user interaction with this client. The user instantiates a namespaced sciunit titled *myro*, and can associate files and annotations with the sciunit using CLI commands (in italics). To create a container within the sciunit, bundling an application's digital artifacts, the user runs the application with the *package* command. The user application can be written in any combination of programming languages e.g. C, C++, Fortran, Shell, Java, R, Python, Julia, etc. In our example, the application consists of the data preprocessing scripts written in R and Python. Packaging an application also incorporates provenance information. Many such containers can be created within the same sciunit. We describe the container creation process in detail in Section IV, and discuss the content-based deduplication storage method used for storing multiple containers in the same sciunit in Section V.

The client works in a Git-like fashion in that the *myro* sciunit is stored only locally unless it is explicitly shared with a remote repository. This method of operation allows distributed collaborators to work offline on the same sciunit. When a user is ready to share, she can publish a container to a remote sciunit using the *publish* command, which instructs the client to upload the container to a Web-based repository. The repository reads the container's contents, stores the container's digital artifacts in the appropriate remote sciunit, and associates the container with an appropriate cloud execution server on which it can potentially re-execute.

A container within the sciunit can be re-run directly from the client, either locally on the local machine with the *repeat* command, or remotely on a remote execution server with the *repeat remote* command, as shown in Figure 2 (lines 6 and 7). In the remote case, the target container is downloaded from a Web-based repository to a remote execution server, and, if the container is compatible with the execution server's architecture, the execution server runs it and sends the results back to the user. Both local and remote executions may optionally be repeated as partial executions. This option will be described in detail in Section VI. Finally, the user can modify a container by downloading it, modifying its code or data and running it locally, and then uploading the modified container, at which point a new version of the container will be stored in the Web-based repository.

## IV. Creating sciunits

Application virtualization tools typically run in two modes: an audit mode to create a container, and an execution mode to re-run a container. In AV audit mode, a container of a user application is created as the user executes the application (in the context of auditing, such an execution is termed a *reference execution*). We describe the audit process assuming that the application is running on a Linux machine. During execution, the Linux *strace* utility is used to monitor the running application process. *Strace* internally attaches itself to the process using the *ptrace* system call to monitor all the system calls of the running process. It intercepts each system call[2] to determine the running process' state and the arguments to the system call. For example, when a process accesses a file or a library using the system call *fopen()*, the *fopen()* call is intercepted. The intercepted system call is "paused" to examine input arguments and the process control block. For instance, in *fopen()*, the file path parameter is extracted. By intercepting all calls, AV auditing determines all[3] program binaries, libraries, scripts, data files, and environment variables that a user program is dependent on. The audit process is similar for Windows and macOS, except that different OS-specific debugging utilities are used.

The system call pause time is brief, requiring only two lightweight context switches added to the normal system call flow; experiments show that the overhead of intercepting system calls is minimal. During the pause, the identified dependencies are used in two ways: first, to create a "sandbox" application container that includes all identified dependencies, and second, to create an interaction log of the reference execution. The sandbox container is named with a package hash and placed in a special "root path" (as described in SectionV), and contains all the dependencies that were identified during the reference execution audit. The dependencies are placed at the same path within the special root path as they were identified in the original system. Figure 3 shows the contents of a container. This path-mirroring has the side effect of exposing user directories and file system layout when the resulting container is reused. Thus, as a practice, creation of a reusable research object is best done within a shared or public namespace.

The interaction log generated during the AV audit phase contains interactions between processes, or between processes and files. Figure 4 shows part of an example interaction log file. Since the log also stores the precise time of each interaction, when evaluated cumulatively it therefore stores the range of times that processes interacted with other processes or with files. These times are available as user-revealable details in the provenance graph, which is constructed by toplogically sorting the interaction log.

[2]There are approximately 50 of such calls defined in the POSIX standard

[3]Not all program dependencies can be detected through this method. But a program's static dependencies are much simpler to gather using programs such as *file, ldd, strings, and objdump*. Our client provides commands for users to find additional dependencies and include them, if necessary.

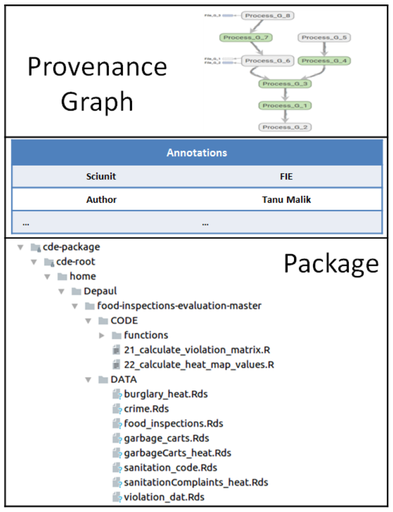
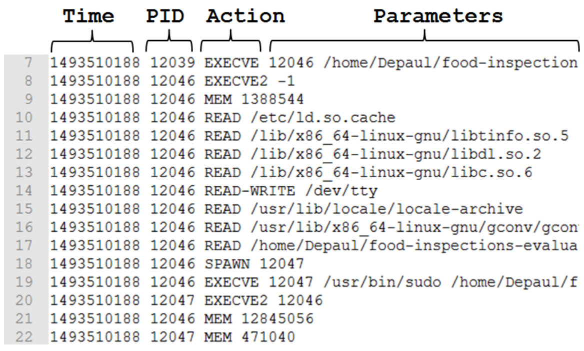


Fig. 3. An example sciunit container



```
1493510188 12039 EXECVE 12046 /home/Depaul/food-inspection
1493510188 12046 EXECVE2 -1
1493510188 12046 MEM 1388544
1493510188 12046 READ /etc/ld.so.cache
1493510188 12046 READ /lib/x86_64-linux-gnu/libtinfo.so.5
1493510188 12046 READ /lib/x86_64-linux-gnu/libdl.so.2
1493510188 12046 READ /lib/x86_64-linux-gnu/libc.so.6
1493510188 12046 READ-WRITE /dev/tty
1493510188 12046 READ /usr/lib/locale/locale-archive
1493510188 12046 READ /usr/lib/x86_64-linux-gnu/gconv/gcon
1493510188 12046 READ /home/Depaul/food-inspections-evalua
1493510188 12046 SPAWN 12047
1493510188 12046 EXECVE 12047 /usr/bin/sudo /home/Depaul/f
1493510188 12047 EXECVE2 12046
1493510188 12046 MEM 12845056
1493510188 12047 MEM 471040
```

Fig. 4. An example interaction log file

In AV execution mode, the application is executed from the container itself by monitoring its processes with *strace*, interrupting application system calls and extracting their path arguments, and redirecting all system call paths to paths within the special root path of the sandboxed container. By redirecting all application file requests into the container, the AV execution method fools the application program into believing that it is executing on the original audit-time machine with original file paths [15].

The advantages of using the AV method are the ease with which a reusable research object can be created, and the machine-agnostic reuse that such an object affords. The disadvantages of the method are that the generated provenance is too fine-grained (at the file and process level) for ready analysis, and that repeated containerization can lead to many redundant files in the same research object. We address these two concerns in the next two sections.

## V. Storing sciunits

A reusable research object may include many containers. If the AV audit method is used on an application to create a container for a sciunit, each time that same application is audited all the same file dependencies of the application will be copied into a new container. This copying takes place even if the same dependencies were present in other previously-created containers based on the same audited application.

One way to eliminate such dependencies is to check for duplicate dependencies during the AV audit phase (i.e. when the user uses the *package* command). However, this tactic significantly slows the phase down. Our technique is to reduce redundant storage deduplicates containers into a single storage unit when each is published with the *publish* command. In this section, we describe how such redundancies are detected, and how the sciunit, when shared, makes use of this optimized storage.

In content-based de-duplication [3], unique chunks of data, or byte patterns, are identified and stored. New chunks are compared to stored chunks, and whenever matches occur, redundant chunks are replaced with small references that point to stored chunks. To identify chunks in a file, data is usually split into lines or into fixed-size blocks. These are useful ways to split if the majority of files are text-based, and they are commonly used in UNIX utilities such as *diff* and *patch*. However, in our case, containers have many binary files, and thus line-based or fixed-size chunking is very slow in identifying duplication.

We use content-defined chunking that marks chunk boundary points based on patterns in the data. Rabin fingerprinting [30] detects patterns in data and determines block boundaries using a sliding window that scans over the data bytes and provides a hash value at each byte point, using a recurrence relation defined as:

$$H(X_{(i,n)}) \leftarrow (H(X_{(i-1,n)}) + X_i - X_{(i-n)}) \bmod M,$$

in which $n$ is the window size, $X_{(i,n)}$ represents the window bytes at byte position '$i$', and $M$ is the total length of the file. Using the recurrence relation, the hash value at any position $i$ can be cheaply computed from the hash at position $i-1$. There are two different ways of using this algorithm to deduplicate data: find duplicate hashes simply by iterating over all calculated hashes, or use a combination of fixed-size and rolling hashes as used in *rsync*. We use the former technique, since we expect each research object to be fairly modest in size, unlike most storage systems where *rsync* is commonly used.

Once rolling hashes have been computed from a file, and a different block is detected, the difference itself can be be stored either as a delta or as a distinct block. The delta method is typically used when the predominant use case is to efficiently obtain a specific version of a file. In our case, we need to strike a balance between storing multiple overlapping containers and storing versions of a single container. Thus we choose the distinct block method, as shown in Figure 5, in which all unique blocks across all containers, versioned or not, are stored.

Given this optimization, a container then is just a symbolic view over deduplicated storage, as shown in Figure 5. However, for the user this optimized storage is opaque. The

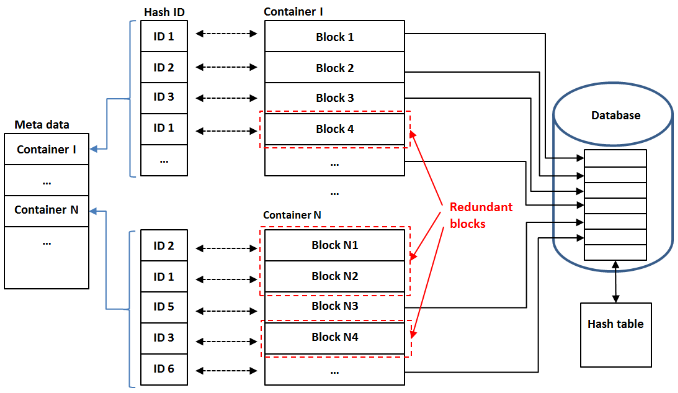


Fig. 5. Block-based deduplication of containers

user still simply runs a specific container, and given multiple containers, can use an included manifest to select a container to run. Internally, the system first materializes the selected container using the enumeration of blocks corresponding to a container, and then executes the assembled container. We would like to emphasize that the materialization is simply a concatenation of blocks and requires no further processing. This procedure is fundamentally different from a delta-based mechanism that checks out each version iteratively.

## VI. Reusing Sciunits

When a sciunit is published, the server distinguishes between the computational part (i.e. the application container) and the non-computational part (i.e. the informational digital artifacts) of the sciunit. The computational part is associated with a cloud instance that can remotely execute the container on user request. A new user can reuse a published sciunit in one of three ways: (i) exact repeat-execution, (ii) partial repeat execution, or (iii) modified repeat execution. To exactly repeat, a container is simply downloaded and then run locally with the *repeat* command, or run remotely with the *repeat remote* command: the container will execute exactly as it did when it was created with the *package* command. To partially repeat or run a modified repeat, a container is downloaded, processed for partial or modified execution, and then either run locally or published to run remotely. We now describe the processing required for partial and modified repeat executions in detail.

### A. Partial Repeat Execution

To partially repeat, a user selects one or multiple processes within a container. These processes are identified by their short pathname or PID, and the user can also use the provenance graph to aid in identification. While the provenance graph can be quite detailed for a user to choose specific processes, in Section VII we describe how a user can see a summarized application workflow akin to the workflow presented in Figure 1 from the provenance graph. Thus, for example, using the container from Figure 1, a user selects the processes "Calculate violation" and "Generate model data" as the group of processes to be partially repeated (starred in Figure 1). Since this user-selected group of processes may not include all related

**Algorithm 1:** Build sub-container for partial execution

```
1  BuildSubContainer($selectedProcs$, $container$):
2      $subContainer$ = initialize($container$)
3      $allProcs$ = getAllProcs($container$)
4      $requiredProcs$ = getProcs($selectedProcs$,
         $allProcs$)
5      $reqProcDeps$ = getDeps($requiredProcs$)
6      foreach $dep$ in $\{reqProcDeps\}$ do
7          /*add dep to correct location in subContainer*/
8          add($dep$, $container$, $subContainer$)
9      return $subContainer$
10 getProcs($selectedProcs$, $allProcs$):
11     $result$ = $\{selectedProcs\}$
12     foreach $proc$ in $\{allProcs\}$ do
13         foreach $selProc$ in $\{selectedProcs\}$ do
14             if isDescendant($proc$, $selProc$) then
15                 $result$ = $result \cup proc$
16                 break
17     return $result$
18 getDeps($requiredProcs$):
19     $result$ = $\emptyset$
20     foreach $reqProc$ in $\{requiredProcs\}$ do
21         /*retrieve all related files and dependencies*/
22         $deps$ = relevantResources($reqProc$)
23         $result$ = $result \cup deps$
24     return $result$
```

processes needed for re-execution, we must calculate these related processes, along with the data files they reference. The calculated processes and files will constitute the new "partial repeat" container or "sub-container." Algorithm 1 shows the procedure for building the sub-container. It starts with the list of user-selected processes (*selectedProcs*), and progresses to include all relevant processes and files by traversing the lineage of the graph (Lines 10-16). The *getDeps* function assumes that any intermediate data files, if included as dependencies, still exist as generated from previous execution runs. The execution of this algorithm ensures that the data file "Heat map data," generated from the previous run of the process "Calculate heat map," is included in the sub-container, even though in the new partial repeat execution the process "Calculate heat map" will not be re-executed.

### *B. Modified Repeat Execution*

To run a modified repeat of a sciunit container, a user examines a downloaded container and determines how particular computations within it should be modified (e.g. by modifying certain sections of code or input data). The sciunit's included provenance graph aids this modification task greatly. Next the user runs the modified container. To share the modification, the user would simply run it with the *package* command, and then publish it with the *publish* command. Enabling modification through a visualization mode, in which users can specify alternate processes or input data files assisted by a GUI, is part of future work.

## VII. Provenance graph visualization

Provenance information generated by AV audit methods is very fine-grained. A graph created from a complete set of generated provenance, using normal visualization structures such as tree or list representations, would be far too replete to be of real practical value. When viewed, this graph would present significant system-level detail that would inhibit a basic comprehension of the overall application workflow. For example, the intuitive workflow of Figure 1, consisting of 12 nodes and 13 edges, would be represented fully as a dense graph of 146 nodes and 321 edges (Figure 6(a) shows a part of this replete graph). Thus, to create a more intuitive graph, we use a graph summarization method that condenses the low-level details of the full generated provenance information. The graph summarization method is explained in detail in [23], and is briefly described in this section. We further describe how we extend the summarization method to create a graph that presents dynamic workflow cross-sections in a responsive visual interface.

Given a directed graph $G = (V, E)$, where $V$ is the set of vertices[4] and $E$ is the set of edges, we denote $Input(u)$ and $Output(u)$ as the sets of input and output edges of vertex $u$. Respectively, $Input(u) = \{e | \ \exists v \in V, \ e = (v, u) \in E\}$, and $Output(u) = \{e | \ \exists v \in V, \ e = (u, v) \in E\}$. The direction of an edge characterizes the dependency of its vertices. For example, a process $u$ spawned by process $v$ is represented by the edge $(u, v)$, and a file $u$ read by process $v$ is represented by the edge $(v, u)$. The graph $G$ is summarized based on the following two rules:

*Definition 1 (Similarity):* Two vertices $u$ and $v$ are called *similar* if and only if they share the same type and have the same input and output connection sets: $Type(u) = Type(v)$, $input(u) = input(v)$ and $output(u) = output(v)$.

The similarity rule groups multiple vertices into a single vertex if the vertices have same type and are connected by the same number and type of edges. Additionally, edges of similar vertices will be grouped into a single corresponding edge. When applied to our provenance graph, this rule groups different files in the same directory.

*Definition 2 (Packability):* A vertex $u$ belongs to $v$'s $generalization\ set$ if and only if vertex $u$ connects to $v$ and satisfies one of following conditions:

- Vertex $u$ is a file that has only one connection to process $v$: $Type(u) = file$ and $\{\exists! e \mid e \in E \wedge (e = (u, v) \vee e = (v, u))\}$.
- Vertex $u$ is a process that has only one output connection to process $v$: $Type(u) = process$ and $\{\exists! e \mid e \in E \wedge e = (u, v)\}$.
- Vertex $u$ is a file that has only two connections – an output connection to process $v$ and an input connection an-

[4]in our graph, a vertex is of type "file" or of type "process"

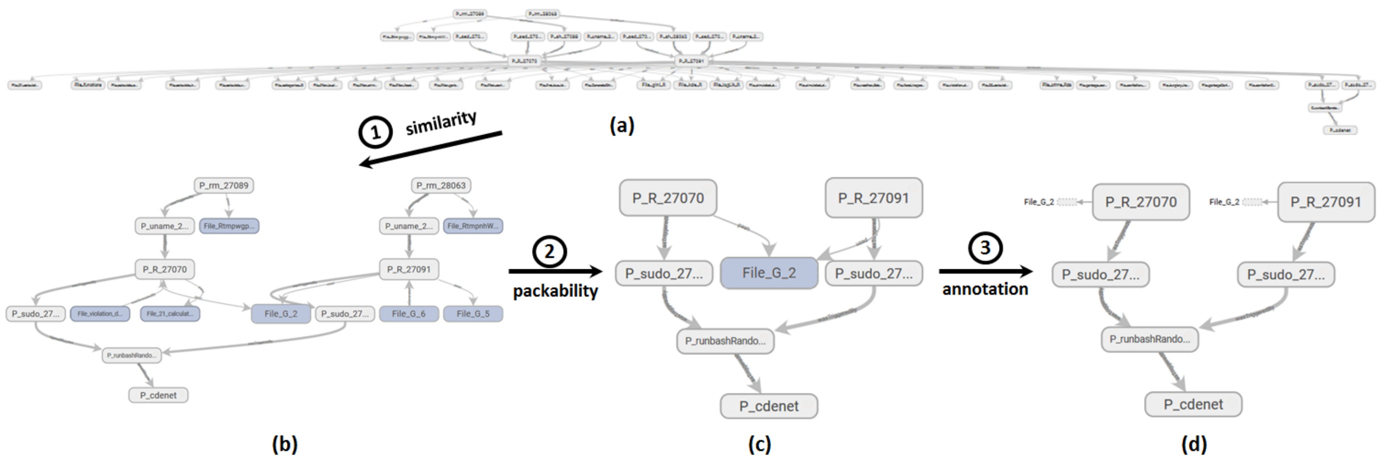


Fig. 6. Graph summarization of a replete graph

other process $x$: $Type(u) = file$ and $\{\exists!(e_1, e_2) \mid (\exists x \in V,\ v \neq x)\ \wedge (e_1 = (u, v) \in E, e_2 = (x, u) \in E)\}$.

The packability rule identifies hubs in the provenance graph by packing files or processes that are connected by single edges into their parent nodes. It also packs files that are generated by a single process and consumed by a single process into their parent processes by producing a process-to-process edge.

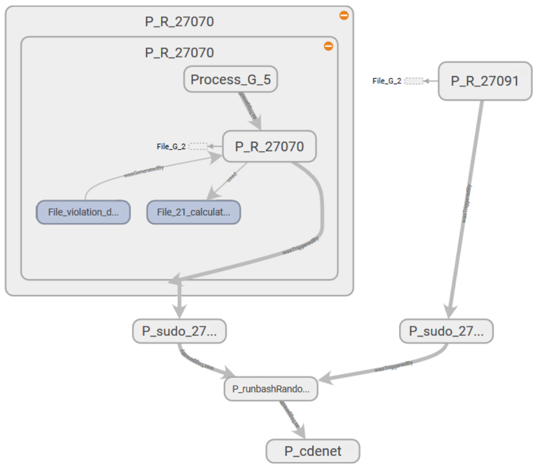


Fig. 7. Expanded view of concealing node "P_R_27070."

When applied in sequence, the similarity and packability rules condense the detail-level of a graph while preserving its core workflow elements. Figure 6 illustrates how applying these two rules to a replete graph produces a graph summary that shows the primary processes in a workflow. Figure 6(a) presents original provenance graph of a subset of FIE workflow of Figure 1 consisting of the data processing steps: "Calculate Viotation" and "Calculate Heat Map". When the rules are applied, it leads to the final graph in Figure reffig:Graph Optimization(d), which is similar to the original graph, except due to the nature of provenance data flow, is presented upside-down.

To lay out the summarized graph, we adopt two visualization techniques: scoping and annotation. In scoping, nodes similar to each other or packed together are represented as single nodes, which can be expanded on user action to reveal the details they conceal. For example, in Figure 7, similarity and packability rules group the nodes within the box into the single node "P_R_27070" (process 27070 runs a subprocess using file "21_calulate_violation_matrix.R" and writes data to file "violation_data.Rds"). The expanded view within the box was obtained by clicking on the concealing node "P_R_27070". Here "Process_G_5" is another concealing node hiding all the dependencies of the R process calculating the violation matrix.

To further improve the layout of the graph, we use an annotation method that assigns higher visualization precedence to process nodes, but annotates them with corresponding file nodes. Figure 6(d) shows how the annotation 'File_G_2', which is a library dependency used both by "P_R_27070" and "P_R_27091" is attached to the two process nodes that generated it. Thus, given a file with $n$ edges ($n \geq 2$), we replace this file with $n$ annotations. A user can always toggle the expanded view to see how the file and process nodes were originally connected. We choose to annotate files – instead of processes – since an application workflow is typically defined by the primary processes that it runs.

## VIII. Experiments

The true usefulness of sciunits can only measured by their adoption. Efficiency of creating sciunits can be a driving force in adopting the use of sciunits over traditional shared research objects. When an efficiently-versioned, easily-created sciunit is shared, along with an embedded, self-describing application workflow, we believe the probability for resuse will greatly increase. In this section, through two real-world complex workflows, we show the time and space overhead of creating, storing, and reusing sciunits.

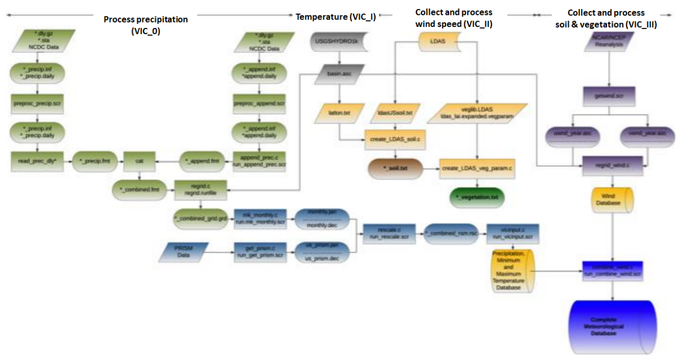

Process precipitation (VIC_0)
Temperature (VIC_I)
Collect and process wind speed (VIC_II)
Collect and process soil & vegetation (VIC_III)
*.dly.gz *.sta NCDC Data
*_precip.inf *_precip.daily
preproc_precip.scr
*_precip.inf *_precip.daily
read_prec_dly*
*_precip.fmt
cat
*_append.fmt
*.dly.gz *.sta NCDC Data
*_append.inf *append.daily
preproc_append.scr
*_append.inf *append.daily
append_prec.c run_append_prec.scr
*_combined.fmt
regrid.c regrid.runfile
*_combined_grid.grd
mk_monthly.c run.mk_monthly.scr
monthly.jan monthly.dec
PRISM Data
get_prism.c run_get_prism.scr
us_prism.jan us_prism.dec
rescale.c run_rescale.scr
*_combined_rsm.rsc
vicinput.c run_vicinput.scr
Precipitation, Minimum and Maximum Temperature Database
USGSHYDRO1k
basin.asc
lation.txt
LDAS
ldasUSsoil.txt
create_LDAS_soil.c
*_soil.txt
veglib.LDAS ldas_lai.expanded.vegparam
create_LDAS_veg_param.c
*_vegetation.txt
NCAR/NCEP Reanalysis
getwind.scr
uwnd_year.asc
vwnd_year.asc
regrid_wind.c
Wind Database
combine_wind.c run_combine_wind.scr
Complete Meteorological Database

Fig. 8. A conceptual view of the VIC workflow [31]

## A. Use cases

We consider two real-world use cases for experimental evaluation: (i) The Food Inspection Evaluation (**FIE**) [26] Workflow, a computationally-intensive use case which has been the running example in our paper, and (ii) the Variable Infiltration Capacity Model, an I/O intensive data preprocessing pipeline for a hydrology model (**VIC**) [31] taken from GeoTrustHub.org. The two examples are interesting use cases for reusable analyses. The first due to transparency in inspection audits owing to open data movement within the City of Chicago, and the second due to the popularity of the VIC model in the Hydrology community with its data preprocessing pipeline relying on heavily on legacy code and begin notoriously difficult to reassemble.

Table I describes the details of the two use cases in terms of programming languages uses, number of application source code files, data and non-data dependencies, and the total size of the application. From a conceptual view, Figure 1 shows the application workflow for FIE and 8 shows the application workflow for VIC.

With each use case, we assume a shareable model in that some steps are assumed to be conducted independently by a user, and subsequently shared with another user who builds upon or forks the shared workflow and adds further to the workflow. Thus the FIE workflow is broken into the following sub-tasks: (i) FIE_0 Calculate only heat map on downloaded data; (ii) FIE_I Share heat map, re-run heat map and also calculate violation to generate model data; (iii) FIE_II Share processed model data with analyst to apply specific model and run cross-validation; (iv) FIE_III Share the entire pipeline with another user to repeat, who downloads data and reruns it. The download is often the most time-consuming step.

All sciunit *package* and *repeat* experiments, along their baselines of normal application runs, were conducted on a laptop with an Intel Core i7-4750HQ 2.0 GHz CPU, 16 GB of main memory, and a 1 TB SATA SSD, running the Arch Linux operating system (current to the rolling release date of 20 June 2017). To more closely examine the purely disk-centric performance of versioning and storing sciunit containers, on a workstation with an Intel Core i7-3770 3.4GHz CPU, 8GB of main memory, and a 1TB SATA HDD, running the Ubuntu 16.04 64bit operating system. The main sciunit client was implemented in Python and C. Sciunit's versioning tool was written in C++, using the block-based deduplication techniques proposed in [3] and [30]. Sciunit's provenance graph visualization was written in Python, using libraries from the TensorBoard [32].

TABLE I
THE FOOD INSPECTION EVALUATION APPLICATION

| | **FIE_0** | **FIE_I** | **FIE_II** | **FIE_III** |
|---|---|---|---|---|
| **Language** | R | R | R | R |
| **# of Source files** | 19 | 20 | 24 | 29 |
| **# of Data files** | 2 | 8 | 14 | 14 |
| **# of Dependencies** | 255 | 255 | 411 | 659 |
| **Total size** | 133.2MB | 178.4MB | 289.7MB | 306.6MB |
| **Time for Execution** | 52.046s | 238.833s | 295.785s | 7200s |

TABLE II
THE VARIABLE INFILTRATION CAPACITY APPLICATION

| | **VIC_0** | **VIC_I** | **VIC_II** | **VIC_III** |
|---|---|---|---|---|
| **Language** | C, C++, Python, C shell, Fortran | | | |
| **# of Source files** | 35 | 61 | 77 | 97 |
| **# of Data files** | 3689 | 6313 | 11460 | 11481 |
| **# of Dependencies** | 247 | 260 | 314 | 357 |
| **Total size** | 1.2GB | 1.3GB | 2.2GB | 2.3GB |
| **Time for Execution** | 158.734s | 306.069s | 363.147s | 377.29s |

## B. Overhead of Creating Sciunits

Table I and Table II present the normal execution times of the two application tasks. Each application is divided into several tasks (e.g. **FIE_0**, **FIE_I**, **FIE_II**, **FIE_III**), or parts, that must be run independently to produce the target computational results. We note that each application encompasses substantial resources (in the form of code and data), has many external dependencies, and is also characterized by lengthy CPU-and-memory-intensive tasks. Additionally, the nature of FIE's processing tasks differ significantly from those of VIC. FIE front-loads its input data sets into memory, and then utilizes machine-learning logic to process its data. VIC also runs many intricate calculations, but differs from FIE in that it interlaces file input and output operations regularly throughout its code. This difference will be key in understanding that sciunits have minimal performance impact on most – but not all – types of applications.

Figure 9 shows the total run times of auditing each application component using the sciunit *package* command and repeating it with the sciunit *repeat* command, in comparison with the original execution times of running the applications normally (i.e. without using sciunit). We note that the performance impact of auditing and repeating on FIE's run times is negligible: auditing FIE with *package* results in only a 3.6% time increase, and executing FIE with *repeat* adds only a 1.3% increase to run time. Conversely, both packaging and repeating VIC with sciunit each nearly doubles the original application run times: as noted in the preceding paragraph, one can see that using sciunit with certain kinds of IO-intensive applications can affect application performance significantly.

We obtain one further observation from these experiments by comparing each application *package* time with its corresponding *repeat* time. Clearly, compared to re-execution increases, auditing increases are slightly higher. which is expected since in the audit mode all the files are monitored and copied into the container. This difference can be understood by considering sciunit's behavior during AV audit-time: auditing entails copying an application's code and data into a sciunit container, but running the sciunit container with *repeat*, however, only redirects to these copied files, and therefore precludes the file copy time.

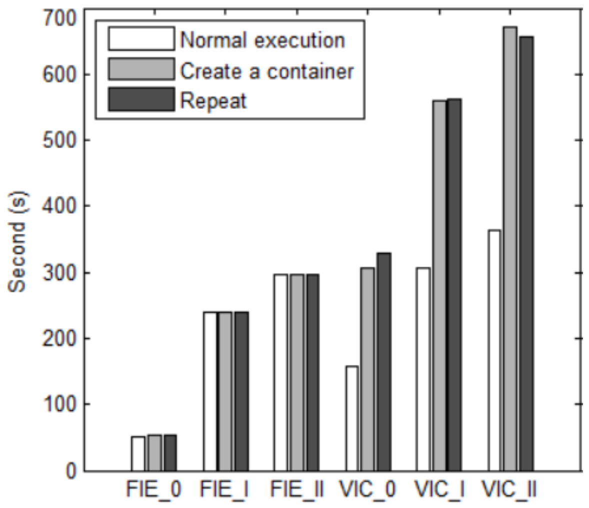


Fig. 9. Application run times for normal run, package, and repeat

## C. Storing sciunits

Figure 10 the space saved due to storing multiple containers of a sciunit in a deduplicated storage. The total space consumed by FIE versions 0-III is 907MB, while when storing our system only takes 333MB. Similarly, VIC versions 0-III take 7GB in total but storing them in our storage takes 3GB. In FIE and VIC the versions have a difference of up to 50% with its prior version and the difference is in both execution files and input/output data.

We also measure the computational complexity of committing and reconstructing a version. Our library to commit a package is implemented to take as input a container, construct an archive so as to have all data blocks in a single file for generating rolling hashes, and then perform deduplication with the storage blocks. Consequently, commit times are a function of the size of the container. The reconstruction process is extracting the relevant blocks from the storage, and creating a package. Even though reconstruction is merely a block concatenation process, it also involves recreating the original file entries from the block and so can take some more time. Irrespective, both commit and reconstruction times are far less than the normal execution time of an application and so are imperceptible to the user. Figure 11 shows the time in seconds

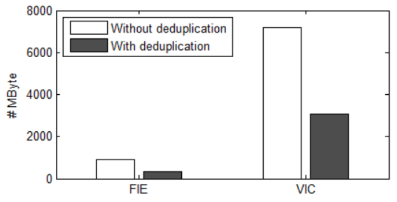


Fig. 10. The saved space with content deduplication

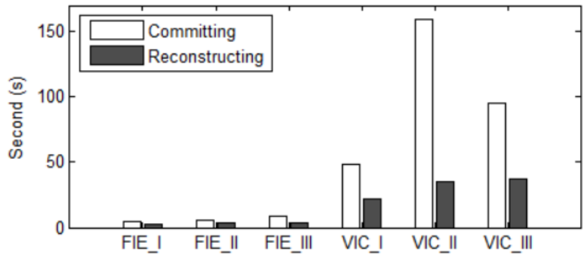


Fig. 11. The execution time for committing and reconstructing a version

for different versions of the use case. As shown commit time is always greater than reconstruction due to computation of rolling hashes, but reconstruction time is somewhat non-negligible due to unarchiving individual files. But the time are less than 10% of the execution time of the application. We do not include an accuracy results of deduplication as the inaccuracy of detecting a duplicate is extremely low since we use cryptographically-safe hash functions.

## D. Provenance Graph Summarization

Since application virtualization leads to fine-grained provenance graphs, in this sub-section we determine if our summarization rules provide a prospective provenance graph that is relative close to the user application workflow. For lack of space, we only show the results for the FIE workflow.

To evaluate the effectiveness of summarization, we first calculate the number of nodes (process and files/resources) and edges in four different containers (i.e., **FIE_I, FIE_II, FIE_III**) in the provenance graph obtained from auditing the application, and then by summarizing the graph with the similarity and packability rules. Figure 12 presents the the number of processes (top), resources (middle) and edges (bottom) using the original graph and using a summary.

As expected, the number connections and resources are significantly reduced: 90% and 86% respectively in average. Meanwhile the number of processes drops by 46%. In fact, the annotation technique only applies on the resources and their connections, thus the decrease in the number of resources and connections is larger than that of processes. Further, the reduction only applies to **FIE_III** and not to **FIE_0** since we only benefit from the graph summarization or generalization if the graphs are large and complex. Results for the **VIC** workflow are similar.

We also measure the number of clicks needed to expand to a full graph. For **FIE_III**, which has the largest graph, expanding a summary node may at most require 4 levels to

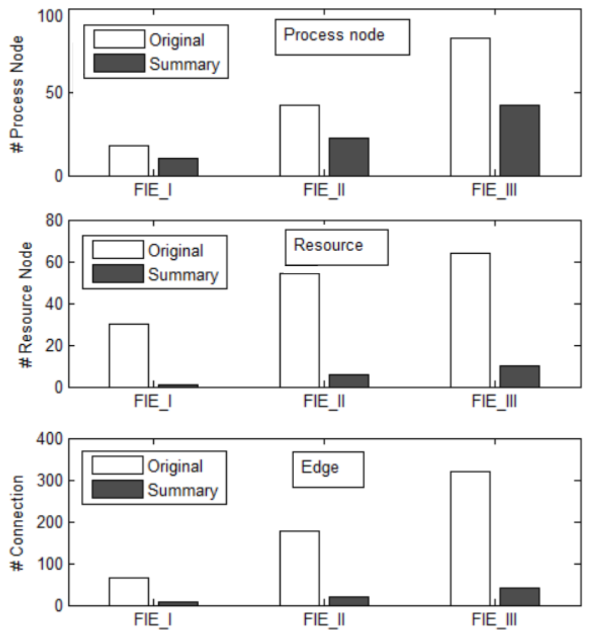


Fig. 12. The number of processes, resources and connections in original technique and Sciunit

its full view. Expanding all the nodes in total can take almost 45 clicks, showing that the graph is summarized very well spatially.

## IX. Conclusion

In this paper, we presented the sciunit, our reference implementation of a reusable research object. We demonstrated how a sciunit can store multiple versioned computational research objects, or executable containers, with a relatively low storage cost. We showed the ease with which sciunit containers can be repeated for exact computational reproducibility, partially repeated for correct execution of a subset of the original computation, or modified and shared for the purpose of allowing others to build on existing scientific work. We also described how a provenance graph, when integrated into a sciunit container and generated with specific methods, makes a shared sciunit self-documenting, and provides a new sciunit user with a dynamic, interactive means of understanding and developing derivative research.

## Acknowledgment

The authors would like to thank Tom Schenk and Gene Leynes for the FIE use case and Jonathan Goodall and Bakinam Essawy for the VIC use case. The authors would also like to acknowledge support for this work from the National Science Foundation under grants NSF ICER-1639759, ICER-1661918, ICER-1440327, and ICER-1343816.

## References

[1] T. Malik, Q. Pham, and I. T. Foster, "SOLE: Towards Descriptive and Interactive Publications," *Implementing Reproducible Research*, vol. 33, 2014. [Online]. Available: \url{https://osf.io/w6fp4/files/}

[2] Figshare.com, "Figshare," https://figshare.com/, 2017, [Online; accessed 2-May-2017].

[3] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 174–187, Oct. 2001.

[4] D. De Roure, "Towards Computational Research Objects," in *DPRMA'13*. ACM, 2013, pp. 16–19.

[5] Y. Roundtable, "Reproducible Research," vol. 12, pp. 8–13, 2010.

[6] K. Belhajjame, J. Zhao, D. Garijo *et al.*, "Using a suite of ontologies for preserving workflow-centric research objects," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 32, pp. 16–42, 2015.

[7] S. Bechhofer, I. Buchan, D. De Roure, P. Missier *et al.*, "Why linked data is not enough for scientists," *Future Generation Computer Systems*, vol. 29, no. 2, pp. 599–611, 2013.

[8] O. Corcho, D. Garijo Verdejo, K. Belhajjame *et al.*, "Workflow-centric research objects: First class citizens in scholarly discourse." 2012.

[9] D. De Roure, K. Belhajjame, P. Missier, J. Gmez-Prez *et al.*, "Towards the preservation of scientific workflows," in *8th International Conference on Preservation of Digital Objects (iPRES)*, 2011.

[10] I. Santana-Perez and M. S. Pérez-Hernández, "Towards reproducibility in scientific workflows: An infrastructure-based approach," *Scientific Programming*, 2015.

[11] https://github.com/wf4ever/ro-manager, 2016, [Online; accessed 2-May-2017].

[12] S. Soiland-Reyes, M. Gamble, and R. Haines, "Research object bundle 1.0," https://researchobject.github.io/specifications/bundle/, 2014, [Online; accessed 2-May-2017].

[13] P. J. Guo and D. Engler, "CDE: Using system call interposition to automatically create portable software packages," in *USENIX'11*. Berkeley, CA, USA: USENIX Association, 2011.

[14] P. J. Guo, "CDE: Run any Linux application on-demand without installation," in *LISA'11*. USENIX Association, 2011.

[15] Q. Pham, T. Malik, and I. Foster, "Using Provenance for Repeatability," in *TaPP'13*. USENIX Association, 2013, pp. 2:1–2:4. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482949.2482952

[16] H. Meng, R. Kommineni, Q. Pham, R. Gardner, T. Malik, and D. Thain, "An invariant framework for conducting reproducible computational science," *Journal of Computational Science*, vol. 9, pp. 137–142, 2015.

[17] P. Ivie and D. Thain, "Prune: A preserving run environment for reproducible scientific computing," in *e-Science'16*. IEEE, 2016, pp. 61–70.

[18] P. Ciccarese, S. Soiland-Reyes, K. Belhajjame, A. J. Gray, C. Goble, and T. Clark, "PAV ontology: Provenance, Authoring and Versioning," *Journal of biomedical semantics*, vol. 4, no. 1, p. 37, 2013.

[19] Y. Gil, D. Garijo, S. Mishra, and V. Ratnakar, "Ontosoft: A distributed semantic registry for scientific software," in *e-Science (e-Science), 2016 IEEE 12th International Conference on*. IEEE, 2016, pp. 331–336.

[20] P. Macko, D. Margo, and M. Seltzer, "Local clustering in provenance graphs," in *CIKM'13*. ACM, 2013, pp. 835–840.

[21] Y. Tian, R. A. Hankins, and J. M. Patel, "Efficient aggregation for graph summarization," in *ACM SIGMOD'08*. ACM, 2008, pp. 567–580.

[22] S. Cohen, S. Cohen-Boulakia, and S. Davidson, "Towards a model of provenance and user views in scientific workflows," in *Data Integration in the Life Sciences*. Springer, 2006, pp. 264–279.

[23] X. Li, X. Xu, and T. Malik, "Interactive provenance summaries for reproducible science," in *IEEE e-Science'16*, Oct 2016, pp. 355–360.

[24] O. Standard, "Topology and orchestration specification for cloud applications version 1.0," 2013.

[25] R. Qasha, J. Cała, and P. Watson, "A framework for scientific workflow reproducibility in the cloud," in *e-Science'16*. IEEE, 2016, pp. 81–90.

[26] City of Chicago, "Food Inspection Evaluation," https://chicago.github.io/food-inspections-evaluation/, 2017, [Online; accessed 7-May-2017].

[27] ——, "Chicago data portal," https://data.cityofchicago.org/, 2017, [Online; accessed 7-May-2017].

[28] ——, "Food Inspection Evaluation predictions," https://chicago.github.io/food-inspections-evaluation/predictions/, 2017, [Online; accessed 7-May-2017].

[29] ——, "Food Inspection Evaluation predictions-source code," https://github.com/Chicago/food-inspections-evaluation, 2016, [Online; accessed 7-May-2017].

[30] M. O. Rabin *et al.*, *Fingerprinting by Random Polynomials*. Center for Research in Computing Techn., Aiken Computation Lab, Univ., 1981.

[31] M. M. Billah, J. L. Goodall *et al.*, "Using a data grid to automate data preparation pipelines required for regional-scale hydrologic modeling," *Environmental Modelling & Software*, vol. 78, pp. 31–39, 2016.

[32] Tensorflow.org, "Tensorflow," https://www.tensorflow.org/, 2017, [Online; accessed 2-May-2017].