2019 IEEE/ACM Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)

Long-term Preservation of Repeatable Builds in Occam

David Wilkinson, Luís Oliveira, Daniel Mossé, and Bruce Childers

University of Pittsburgh
{dwilk,loliveira,mosse,childers}@pitt.edu

***Abstract*—In order to provide transparency, wide availability, and easier reuse of scholarly software, there is a need for greater emphasis on code preservation. Yet, not just mirroring the source code, but preserving the ability to build it. Occam is a tool that offers preservation and distribution using containerization to provide repeatable execution in both building and running software. This paper gives detail about the design of Occam and its potential use within the scholarly community and beyond.**

I. Introduction

Scientific reproducibility is a problem that likely will require a few hops of progress before we land on a complete solution. One major problem is software, burdened by unbridled imperfection and ephemerality. There are bugs to detect. There are aspects of randomness and noise to control. There are external actors and unregulated data sources that may interfere with intended behavior. In truth, there is an unknown yet appreciable chaos surrounding software determinism and preservation. And yet, we need to confidently tackle these problems if we ever wish to preserve our scholarly record. The urgency of this problem only increases as scientific research becomes more and more dependent on code. Software we not only need to keep around but keep running.

Preservation may be thought of as a task within the domain of archaeologists. That there are people curating interesting objects and placing them within museums perhaps long after obsolescence. However, software degrades faster than its potential usefulness. Within their research work, scientists may find themselves acting as their own Indiana Jones if they need to resurrect an older tool.

This is not a problem of digging up the distant past, however. In a 2016 paper [1] students were given the task of repeating software-backed research from just four years prior. Although these students were comprised of computer scientists, they still suffered two major obstacles. The first was the availability of code. Only 35% of the studies had their software publicly available. The second was getting the code to build. In this case, given they had the code, only 58% of the time could students build the software within 30 minutes. Of the rest, only 28% of the studies could be built when given considerable effort, and the students gave up on the remaining 14%.

Here, just four years passed before the reuse of scientific software became a burden. For the sake of accountability and scientific reproducibility, we need to reduce that effort to encourage researchers to revisit older results. Occam is a system that sees such accountability as a preservation problem. That is, there is a need to keep software running and to maintain its behavior.

Beyond the scientific cases, the software community at-large has also been considering the problem of software determinism and trust. Recently, there have been a few notable supply-chain attacks, which occur when a malicious actor introduces bad code into existing software via a dependency. In response, package managers and software repositories, such as Arch Linux's AUR [2] and the JavaScript community's npm [3], have added signature validation as an installation step. This is just one solution to a problem that is largely about accountability.

One question this addresses is whether or not the software you are using is the same others are using. A secondary question manifests as tools age: "If I need to, can I fall back to an earlier version?" Both concern the reliable distribution of software and mitigating issues when software is no longer available or able to run. What does it mean to preserve scholarly software, and to provide a sense of repeatability? What responsibility does a repository of scholarly software have?

It is clear the community needs to build a software distribution platform that proactively addresses both active use and long-term repeatability. Considering the importance of transparency when assessing scholarly artifacts, the preservation of the build process is also notable. Occam [4] is a system that implements such a package management system. It is driven by a goal to be actively useful in disseminating interesting tools and, at the same time, also preserve the ability to both build and run those tools in the future. In the following sections, this paper will address specifically how the system maintains, deploys, and distributes software in an accountable manner.

II. Related Work

The 2016 reproducibility study was itself motivated by growing concern of an impending reproducibility crisis [5]. The scientific community has certainly not relegated itself to just talking about the problem. There is been a significant push in research and development of a variety of potential solutions [6] to this important issue. Occam is an ambitious project that addresses several domains of concern.

The first aspect of Occam or any preservation system is the storage and distribution of digital objects. In this way, Occam resembles a package manager and has been influenced by many such systems already in wide use. Operating system Debian, including the popular Ubuntu, is arguably in widest use with its Advanced Package Tool (APT) [7] package manager and repositories. APT is noted for using GPG signatures to verify packages and their maintainers, a feature borrowed here. Arch Linux has the similar Arch User Repository (AUR) [2] and its package manager tool Pacman. Occam has seeded its own software repositories by translating Arch packages only changing them when necessary for stronger preservation properties. The Node Package Manager (stylized as npm) [3] is focused on maintaining packages for use by developers and not system maintainers. Yet, there is a strong overlap in features between npm, similar package managers, and Occam. For example, we have made direct use of npm's versioning standard [8] to tag software objects.

While these package managers are generally for keeping a system merely up-to-date, there are others that apply more rigorous standards. Both Nix [9] and Guix [10] are two package managers which allow for multiple versions of software and their dependencies to be managed simultaneously. These tools have support for keeping track of software histories, allowing rollback, and cloning software environments across systems. Within the scientific space, which has stronger requirements as hardware becomes specialized, package managers such as Spack [11] have also become popular. Spack targets supercomputing domains and allows for installing software objects into their own environments and restricting packages to those using particular compilers. Like these three tools, Occam also emphasizes packages that are built from source, and goes to somewhat greater lengths to preserve code. Occam also manages isolated software environments, albeit in a somewhat different manner, which will be addressed in Section VI. Overall, Occam focuses not just on package distribution but also the preserved execution of software.

These package managers do not put a heavy focus on the software preservation problem, which is another essential aspect of reproducibility-oriented archive systems. Instead, they focus on description and distribution of up-to-date software for everyday use with an occasional feature to maintain a software environment. Occam builds off of these ideas to provide stronger preservation properties, yet it is not the only system to do so. There are quite a few standards for specifying archival metadata and provenance. The Open Archives Initiative Object Reuse and Exchange (OAI-ORE) [12] defines an RDF-based standard for representing collections of digital objects. The BagIt standard [13] focuses on file layout conventions toward more consistent packaging of digital artifacts. Occam has a JSON metadata scheme, discussed in Section IV, that borrows some of the ideas expressed by these descriptive standards. In addition, however, Occam has added specific details that provide flexible and repeatable execution of archived objects.

Umbrella [14] is a reproducibility system that encapsulates executable objects with metadata that is also used to repeatably deploy that object using Docker or EC2. The metadata Umbrella captures specifies the execution environment including the necessary operating system and provenance information for externally sourced packages. Although Umbrella can track information about such packages, and cache them locally, it does not have nor advise any method of distributing or maintaining these data dependencies. Occam takes the stance that distribution and mirroring is integral to the digital reproducibility problem. In Section V, we will look at the similar process Occam uses to track but also preserve and distribute external content.

In contrast to Occam and Umbrella, which largely require some manual input of metadata, ReproZip [15] is another tool that actively tracks the execution of running software. This tool detects when files are interacted with and can create a distributable package containing all necessary data to repeat that execution. The value comes from the ease of the tool, which greatly eliminates the excuse that packaging research objects is too much work. However, the automation yields a result that is difficult to dissect and reuse since the package is effectively a black-box that reruns only the exact input initially used. Occam takes the stance shared by Umbrella that mere repeatable execution is not good enough. The core design of Occam is to promote the interactivity of archived software; to allow new inputs and data to be used with old software. Every feature present in Occam, including the work described throughout this paper, is in service to this goal.

III. Occam System Model

There are several goals toward providing a trusted, repeatable software repository. First, there is the aspect of origination, which considers verifying an object came from a particular origin. Second, there is integrity, which is the assurance that the transferred content of the object is identical to the one provided by that origin. Last, there is verification where one can repeat the work of another actor and see the same output. With respect to software builds, these all play a role to provide both accountability and confidence in retrieving the requested package.

In our system, software is described and stored within a generic structure we will now refer to as an Object. Any entity that creates or modifies an Object will be referred to as an Actor. Each Actor is defined by a private and public key-pair and referred to with an identifier formed by a hash of that public key. This identifier universally refers to this specific Actor throughout the system. Any Object is also universally identified by hashing together the Actor's identifier alongside metadata for that Object. This way, every Object is inherently associated with its originating Actor, and no Object can exist without having been created by an Actor. Furthermore, upon receipt of any Object, a secondary Actor can easily check both the integrity of the metadata and the id of the originating Actor by recomputing this hash.

The impact of this design is tremendous. Considering each identifier does not rely on any actual name or centralized server, our system model allows for a federated network. In particular, Actors can be created (i.e., enter the system) at any instance, and then migrated to any other instance within the federation as long as they preserve the ownership of their private key. Similarly, Objects can be mirrored and distributed from any node while still allowing for the verification that their data came from a specific Actor. In a later section dedicated to distribution, a more detailed description of this mechanism is discussed.

Indeed, it is an overall goal to not rely on a centralized repository for packages and research artifacts. Due to this ability to mirror/acquire any such object from any available server, Occam increases the value of research objects, such as simulators and scientific data, via wider availability and reuse. To add such federation support, Occam currently works alongside the IPFS protocol [16] [17] with consideration toward supporting the similar Dat protocol [18] in the near future.

Furthermore, when an Actor develops a research tool or curates a useful dataset, they do not also need to individually or even institutionally host that object. This lessens the burden by eliminating the need to offer the required bandwidth for distribution. Dedicated long-term preservation is, instead, a burden shared among all participating institutions and any that can contribute the resources.

IV. Defining Software Objects

In this section, the definition and scope of an Object are defined. An Object is comprised of a set of files collected together. Currently, each Object is stored within a Git [19] repository such that the history of metadata changes are tracked. This provides a mechanism to cite and make use of an Object at a particular point in time without hindering future development. Git was chosen because it is a technology/protocol designed around distribution. Furthermore, Git is designed around forking, which specifically allows an Object to be cloned and altered by independent developers. Beyond technological advantages, this also furthers Occam's goal to promote a reuse culture within and beyond the sciences.

For each Object, there exists at least one JSON file containing metadata defining the object. This information defines both what the Object does in an abstract sense and a detailed description of the computing environment required, if any, to execute the Object. Both software and data Objects are crafted in response to a particular problem, but this general way of describing software will make it easier for others to find and reuse it in their own work.

With that in mind, the Objects need to be discoverable. Toward this, several fields are used to define human-legible labels including a name and description, as seen in Listing 1. These are intended to be used when searching for relevant artifacts as any need arises. For instance, searching for "memory simulator" should discover DRAMSim2 [20], a memory simulator we use as an example throughout this paper.

```json
{
  "name": "DRAMSim2",
  "type": "simulator",
  "organization": "University of Maryland",
  "website": "https://wiki.umd.edu/DRAMSim2",
  "summary": "A cycle accurate model of a DRAM
   ↪  memory controller, the DRAM modules which
   ↪  comprise system storage, and the bus by which
   ↪  they communicate.",
  ...
}
```

Listing 1: General metadata for describing the basic Object

Each Object also has a type which acts as both a general categorical identifier and as a semantic tag used for determining when compositions of multiple objects are allowed. The value is not as obvious when considering any object in isolation. However, any Object can specify, within its metadata, its possible inputs and outputs. This simulator can take a trace as input in order to perform the requisite simulation, as seen in Listing 2. The "trace" object can then be fulfilled by any Object with that particular type. In this case the further subtype "DRAMSim2" constrains this to trace objects of a particular format this simulator understands.

```json
{
  "name": "DRAMSim2",
  "type": "simulator",
  ...
  "inputs": [ {
    "name": "Memory Trace",
    "type": "trace",
    "subtype": "DRAMSim2"
  } ],
  ...
}
```

Listing 2: Specification of inputs makes use of the type and subtype fields to provide semantic filtering.

The remaining fields are relegated to describing aspects relevant to the execution of the Object. Such objects often have two distinct contexts between building and running. At the end of the day, however, building and running do not dramatically differ in how they operate. When you build some software, you are effectively just executing a different command. For instance, to build the DRAMSim2 simulator, the process is executing a script which, in turn, executes a compiler. Between building the simulator and running it, the system would simply map in different dependencies into their respective environments when necessary.

In order to provide each context, the metadata has a "build" and "run" section containing specific details about each environment. This subsection includes a list of dependencies, potentially a set of data that must be copied into the machine, and a command to run. Specific to our example, in Listing 3, we need to create an environment with g++ available, pull down a Git repository, and invoke the provided build script contained within our Object. The Git repository is the source code for the simulator, which is being actively developed outside of the preservation system in a traditional manner. Since it is from GitHub, an external resource, it has its own preservation problem we will specifically discuss in the next section. Upon executing the build script, this code is placed in the directory specified, in the given case the "package" directory, during the initialization of the build environment.

After the build environment is established based on the metadata, the system invokes the given command. Here, it will run a shell script that contains the operations required to build the software. Generally, in many cases, this is some short configuration followed by invoking a build engine, such as "make". We will discuss more of the specifics of how the environments are executed in an upcoming section, and discuss the merits of different containerization tools for this task.

V. Resource Mirroring

Although our goal is primarily the preservation of software builds, in order to accommodate different development styles and team structures it is not expected that software be actively maintained within Occam. Like Arch Linux and similar package managers, the development occurs within its own ecosystem. For instance, developers may congregate around a service such

```
{
  ...
  "build": {
    "dependencies": [ {
      "id": "QmVseooVZ4dn2Vo4ikwKXz...tKmujP4KC8",
      "name": "g++",
      "type": "compiler",
      "version": ">=5.0"
    } ],
    "install": [ {
      "type": "resource",
      "subtype": "git",
      "to": "package",
      "name": "DRAMSim2 Source",
      "source":
        ↪  "Git://Github.com/dramninjas/DRAMSim2",
      "revision": "5drSurGACruKgke...ZYejLu4TBYX"
    } ],
    "command": ["/bin/sh", "build.sh"]
  },
  ...
}
```

Listing 3: Build specific metadata to create a specific build environment. *Note*: simplified for the sake of space.

as GitLab [21] or GitHub [22], which are private companies that host source code repositories. Development teams push changes to code to their own repository without involving any of the systems that build and distribute it.

Relying on externally sourced data, however, poses a threat to the goal of long-term software preservation. If source code disappears, the ability to build it will be drastically damaged. As an added consequence, modifying or extending that software will also become more difficult. Ultimately, build systems make a choice about whether or not they mirror these resources to mitigate such issues. The Linux distributions Debian and Ubuntu offer repositories [23] that mirror the source code of software packages. You can then choose to build many open-source packages from their archived source code. Repositories such as Arch Linux's AUR choose not to do so [24], relying on detecting when resources are no longer available, flagging them, and updating them as they break.

The trade-off weighs the long-term ability to use or rebuild software at any time over the storage and labor cost of maintaining a source code mirror. In these two cases, the reasoning behind each choice is clear. Ubuntu offers a Long-Term Support contract [25] designed around organizations and businesses that would be interrupted by constant software changes. Ensuring packages are static and available, even if software becomes stale, provides that form of stability. On the other hand, Arch Linux is oriented around individuals and built around ensuring the latest version of software. In this case, security and feature development are preferred even when bumping software to a new version might cause a problem. Therefore, in Arch Linux, packages are commonly short-lived before updated, replaced, and rebuilt. Since there is little emphasis on revisiting older versions, there is less of a need to mirror source code.

When reflecting on the needs of scientific reproducibility, neither model seems particularly apt. A researcher will generally want to use the latest software and keep research tools up-to-date. The same researcher, when collaborating, would want others to use the same exact software. Furthermore, the scientific method is a centuries old idea that relies on experimentation being well-described and repeatable. This suggests that there is value in the ability to recreate a software environment exactly as it was. In this case, we want to have the accountability of rebuilding any part of our experiment apparatus from its source code along with the ability to recreate the software environment for that build. This has value even years beyond a work's initial publication. However, this requires preserving a mirror of external data used in the build.

Focusing on our ongoing example, the DRAMSim2 simulator is built from source code found on GitHub. Code repositories hosted on such platforms are relatively common. GitHub in particular has millions of repositories. [26] Alternatively, developers may choose to release source code periodically in the form of compressed archive, such as zip or tar. Regardless of method, the software build is reliant on this data.

Occam provides a mechanism to preserve and mirror this information. As seen in Listing 4, the "install" section tells the system to place files in the given path. Here, a resource is provided which has a "source" field that points to the externally hosted Git repository on GitHub. When an Object is created and pulled into the Occam repository and it encounters a resource that it has seen for the first time, it will download and locally store the linked content. The system no longer needs the remote content as it now makes use of the local mirror instead.

```
{
  ...
  "build": {
    ...
    "install": [ {
      "type": "resource",
      "subtype": "git",
      "to": "package",
      "name": "DRAMSim2 Source",
      "source":
        ↪  "Git://Github.com/dramninjas/DRAMSim2",
      "revision": "5drSurGACruKgke...ZYejLu4TBYX",
      "id": "QmesFhk5cjGB1LxD9QMxa1...Pg8peaXGTB",
      "uid": "QmfXe8DzxrFuqUDnYcAea...BsbHegNEpS"
    } ],
    ...
  },
  ...
}
```

Listing 4: External resources are defined and then mirrored within the system and assigned universal identifiers, which are truncated here for brevity.

Just as any Actor and Object in the system have a universally unique identifier, resources have an identifier directly tied to its source URL. This ensures that if two people on two different servers pull down the same resource, it has the same universal identifier and each immediately become a mirror of the other. Furthermore, since content at a URL may change, the data itself is hashed such that it refers exactly to the data retrieved at a given moment. When a build is invoked on the system, the identifier and hash of all used resources are noted.

There are several ways of preserving external repositories.

For instance, mirroring Git repositories could be done by creating a zip file of the contents. This defeats some of the purpose of Git, which is designed to capture an entire history of changes. However, by specifically handling Git repositories, as Occam does, the system can be more efficient in preserving them while retaining such interesting properties. Occam takes this one step further and adds the ability to clone the Git repository directly from any particular Occam server, as though it were GitHub. The system is extendable via a plugin system, and alongside Git support, there is also a similar extension for Mercurial [27] repositories.

Careful repository preservation is important with respect to builds. Hosting services such as GitHub allow you to effectively forget or rewrite the history of your source code. This is not malicious, but, in fact, a common aspect of development. When finishing a new feature, there may have been dozens of changes that constitute the work. Upon releasing a new version of the software, some teams elect to squash all of these small changes into a new history that shows large, well-defined progress. This makes the repository much easier to follow, but it, to some effect, creates a fake history as to how the software actually changed. If an artifact were built referring to a point in time that was erased, that old reference would no longer exist at its source.

This would absolutely prevent repeatable execution of that build for systems that do not provide a mirror. This means that inspecting or modifying the source code of a prior experiment would no longer be possible, hindering both accountability and reuse. Most devious is to note that such systems may not know of this damage until somebody attempts to repeat the work. This means the damage is only noticed when it would most hurt the community.

Therefore, updating our local mirror involves managing both histories at once. To mitigate this preservation issue, Occam leverages the "alternates" feature within Git that allows it to manage multiple, potentially divergent, histories of the same Object. This is the same mechanism used behind the scenes at both GitLab [28] and GitHub [29]. With this implementation, when Occam pulls a Git repository, it spends some extra effort to retain the particular reference.

In the end, this scheme strongly supports the goals of build preservation with respect to scholarly work. First, the mirroring of repositories is important to the health of the scientific community. It strives to treat code with the preservation respect it deserves when code is so often the impetus to a published paper which should not outlive it. Furthermore, the careful handling of mirrored repositories allows for preservation to be a burden shared among a federated community. That is, by having repositories retain universal identifiers, it encourages mirroring across many institutions, and reduces reliance on hosting services staying active. Finally, Occam implements resource mirroring as a plugin providing an extensible way for the system to grow with the needs of the community moving forward. Overall, this serves to increase the confidence that useful code remains available.

VI. Repeatable Deployment

Of course, assuring access to the data necessary to build software is only one half of the puzzle. In order to build or run an Object, the system needs to translate the process defined by the Object into something that can be understood by the native machine. Furthermore, the method it uses needs to be backed by a system that can reliably repeat that process.

Toward this, Occam takes an Object and using its metadata generates a virtual machine manifest that targets a particular virtualization backend. This manifest contains a short record of each and every necessary software dependency that must be available within the machine to complete the requested task. When the manifest is passed off to an execution backend, it will spin up the virtual environment and execute the requested command. As mentioned, this process is identical with respect to both building and normal execution of any Object, it simply looks at different sections of the metadata.

Occam is inspired by many modern package management systems. In these systems, a software package is defined by a document that describes requirements in terms of versions of other software that the base package is dependent upon. The JavaScript implements a dependency system using the npm [3] package manager, which has greatly inspired the design of Occam's dependency resolution. When such a package system installs a package, it will install each dependency. To do so, it recursively installs each sub-dependency, and so on, until every software package is resolved.

It works the same in Occam. Specifically, each object lists its dependencies as part of its overall metadata, as already seen in Listing 3. DRAMSim2 is written in C++ and therefore requires the GNU C++ compiler, g++, version 5.0 or later. The compiler itself has its own dependencies, which now must also be included within the virtual machine. Satisfying every requirement will result in the creation of a task manifest. This manifest lists the exact version of every Object required.

As established, the algorithm to generate the manifest is a recursive process. It is given a set of objects tagged with version requirements and finds a superset of objects that satisfy them. Each Object has version tags attached to particular points in its history. The version tags are in the form specified by the Semantic Versioning standard [30]. Version constraints can be specified alongside each dependency in the same form used by npm [8], which allows it to denote ranges of valid versions. Note that in our example, the tag uses certain symbols to denote that any version greater than or equal to 5.0 is valid when building our simulator.

More generally, Occam will recursively resolve each listed dependency by selecting a possible version and attempting to resolve each sub-dependency. Each time it visits an Object in the dependency list, it will make a note about each version requirement along the way. If a version requirement contradicts one that was already seen for the same Object, it will fail to resolve the pending dependency and attempt to take a different path by selecting a different version somewhere up the chain. It may also fail if it can select a valid version of an Object but that Object has not been built. When this is the case, it may conservatively try to find a version of the Object that has a known build, or elect to restart the whole process in the new context of building that Object first. Assuming each required Object is satisfied and has been previously built, the result will be a list of all necessary objects. This defines the execution environment and is essentially what comprises the task manifest.

Much like the resource subsystem described in the previous section, Occam also has a dedicated, extensible subsystem for deployment. One can provide, independently and at any time, a plugin to execute a task manifest on any virtualization stack. Currently, we have plugins for Docker and Singularity, and an experimental plugin providing Vagrant [31] support. Each of these implements an interface where a task manifest is given as input, which it then executes after establishing the described environment. This function handles some lower-level details to track the running process, capture its output, and detect if the process crashes or reports an error.

One general rule is that the task manifest is a snapshot of the environment. Each requested version of each requested Object was previously determined and frozen within the task manifest. That is, giving the same manifest to the backend should result in that machine having the same file structure and, therefore, the same exact software and, thus, the best chance for deterministic behavior. Repeating a build, and, more generally, repeating any execution is succinctly handled by distributing and executing the original manifest. One can then compare the result of each run to evaluate determinism.

Let's once again consider DRAMSim2. Our task manifest to build our Object will contain our compiler, g++, along with its necessary dependencies. The system gives that manifest to the backend system to have it actually execute. That backend respects the manifest by loading into that environment each given Object and then executes the given command. While it runs, it compiles each source code file as listed in its build script. At the end, it links together each compiled file to form a single DRAMSim2 executable. With that, it will store that executable such that we can now run the Object via a new manifest.

To validate build reproducibility, we need several pieces of information. We would need the task manifest, in order to provide the same environment. We would also need a previously built binary to compare against. In Occam, we simplify this problem by hashing all of the built binaries as a final step. This is determined by hashing the resulting binaries and any other files using a SHA-256 variant of the hashdeep algorithm [32]. The algorithm was modified in a trivial way to order files alphabetically when it hashes directory structures. This allows hashdeep to have deterministic behavior across file-systems.

In Occam, we have several hundred software packages. So far, we have maintained the vast majority of them to provide build determinism. As noted by other similar package repository systems, some software have build systems that are non-deterministic. Therefore, these builds would produce a unique hash each time. Occam, like those systems, cannot handle verified build repeatability for these cases. Instead, those software packages will have to be patched in order to mitigate this problem.

Another avenue to be explored in the future is repeatable behavior verification. In this case, the system would execute a set of tests and check that the output generated matches a previous invocation. Generally, a combination of the two methods is ideal. A set of tests is not necessarily complete. Also, malicious activity may introduce a false-positive test or circumvent the testing in some way. Nevertheless, ensuring the integrity of software as it moves around to different machines will be a core requirement of our broad software ecosystem.

VII. Flexible Virtualization

In terms of execution environment, Occam makes substantial use of high-level virtualization tools such as Docker [33] and Singularity [34]. Each of these tools provide a mechanism to run software within an environment isolated from the rest of the machine. The benefit of using these tools is that a system can recreate the same necessary environment on two different systems without worrying about how software may interact with the existing system. For preservation use, this means an environment can be locked to a particular point in time even when system software elsewhere is more up-to-date.

However, our system does not need many of the features supplied by these tools in order to provide build repeatability. Since Occam provides its own object storage and distribution and has a very specific use-case in mind, it can very carefully leverage just a subset of each tool. For instance, Occam does not actually build any independent volumes or containers in either tool. Instead, it simply builds one container with effectively nothing but an initialization script in both Docker and Singularity. Every other Object that must be present within the running container is simply mounted in read-only direct from object storage. Thus, it is rather unconventional since it technically runs the same container for absolutely every case.

This allows the system to craft very divergent virtual machines quickly, without incurring a cost to generate each container as needed. It also very efficiently reuses any Object among multiple running containers, avoiding the cost to redundantly store them. This is especially true for any Object which is created by one process and then immediately used as input to the next.

We do not miss the supposed benefit of independent Docker containers, either. Distributing such containers is needless since the task manifest already contains the equivalent information to recreate the environment elsewhere. It has the benefit, in fact, of the manifest only taking up a few kilobytes of space. This design considers the effect of software degradation on container infrastructure by allowing future invocations to use different tools. In fact, by not relying on a particular containerization tool, the system may gracefully, in the future, repeat older work with more rigorous virtualization or emulation options.

However, our implementation is certainly an unusual usage for both of these tools, and it is unclear which is best suited to this particularly peculiar practice. Even with all cases considered, Docker and Singularity do not differ in any substantial way. From a software design perspective, neither Docker and Singularity benefit from language choice as both are written in Go, a compiled systems language. They both make use of specific Linux kernel functionality to provide isolated process spaces. These ensure that a process sees a certain determined view of the underlying system, a particular view of the file-system, and obscures information about other processes. The trade-off is that certain aspects of the machine are not hidden, such as the kernel software, some drivers, and the characteristics of the processor it is running upon. Yet, this is true for both tools, and is largely the allure of these containerization platforms, since simple virtualization often works well enough. In summary, both tools provide process isolation while avoiding the burden and inefficiency of emulating hardware.

However, one difference between Docker and Singularity is the way processes are spawned and managed. Docker is structured around a daemon task that is always running as root. When a container is executed, the daemon interjects and runs it on a user's behalf, potentially giving that user root access. This design emphasizes multiple container orchestration within a single organization where you run particular software and trust the users. Singularity, on the other hand, runs only when invoked, requiring much less privileges to run most containers. From that perspective, Singularity is a strong choice on more constrained or more security-focused environments. Particularly, it works well in situations where tasks are scheduled to run and deployed within distributed systems with tools such as Slurm. Singularity has indeed found a stronger foothold within shared clusters and HPC environments, whereas Docker has a strong following within industry for deployment at scale and use with respect to testing, staging, and continuous integration.

Yet, as mentioned, the use of these tools within Occam does not fall neatly into either of those categories. However, both tools provide the same necessary behavior and Occam is purposefully designed to use either in order to be a more general tool for the overall community. To determine what impact, if any, the choice of backend had on performance, we had Occam exhaustively build many different versions of software within our repository. We chose GCC, a widely used free and open-source C and C++ compiler, due to its complexity, long build times, and heavy disk usage during its build process. Also, it had the advantage of having strong provable build determinism, which indeed was maintained throughout our experimentation. That is, each build produced the exact set of files in both Docker and Singularity when determined by hashing the resulting binary as described previously.

Using a modest dedicated server with a quad-core Intel i5 running at 3.10 GHz with 16 GiB of RAM, we built different versions of GCC that span over 5 years of development. Stepping each version, from 4.7.4 to 8.1.0, we built each package ten times with Occam using Singularity. Then, we repeated the entire suite once over using Docker instead. The average execution time, measured by clock time, of each build is never dramatically different between the two container platforms. This is unsurprising considering how each tool essentially performs the same function with the same mechanism. Furthermore, this corroborates more rigorous performance tests [35] between Docker and Singularity, where the latter only outperforms Docker in specialized cases while otherwise showing no significant difference. However, Occam simply does not use such complex features that differentiate any of these tools.

Indeed, Occam avoids the more complex features of both tools and still provides substantial value in build management and software preservation. Simply, in our model, containers are not the product. Instead, containers are always built as needed from a set of software packages. The container, as a result, remains ephemeral as it can be built the same way at any time, which reduces much of the responsibility required of the containerization tool. This implies simpler tools to replace Docker and Singularity may become more valuable in the context of more specialized software packaging. Furthermore, as such a smaller, more specialized containerization tool is developed, its performance can be dedicated toward specific environments. That is, similar to how Singularity is essentially a more specialized version of Docker, there is likely room for other specialized tools. Since our system is agnostic to the backend being used, Occam can easily support every such tool as it arrives.

VIII. Distribution and Verification

In the end, a software repository is only useful as far as it enables reuse. After all, the scientific method calls for such transparency in any experimental process. Therefore, a goal of any such system is to allow software to be distributed while still maintaining the accountability and repeatability already demonstrated. There are such existing tools for encapsulating scholarly software. However, one overlooked aspect to the publishing of digital science artifacts has been the independent verification of software builds.

Tools such as ReproZip [15] and Umbrella [14] are valuable and certainly offer substantial relief to the distribution of experimental artifacts. ReproZip traces a program or script as it executes and creates a repeatable executable environment in the form of a container or tar archive containing any file that is touched. Umbrella is closer to Occam in that it encapsulates artifacts with metadata that describes the environment and creates that context upon its execution. Yet, their use-cases center around the reproduction of some computational process, such as a single script, and do not incorporate knowledge about the environment in which certain software is built.

Occam, in contrast, emphasizes using software that has been packaged within the system and built in a supervised manner. As we have established, each Object has metadata describing the process for building and running it. However, it is impractical to expect that an Object always be built when it arrives on a new machine. It is also possible to simply distribute the result of a build, much like a traditional package manager. That is, the rigorous verification of a build is now optional.

Generally, when an Object is requested at a particular machine, the first action is to determine a server that contains that Object. From there, the metadata and file content is distributed from that server to the one making the request via the Git protocol. Depending on its intended use, the system can then elect to recursively pull down related objects, such as dependencies, build dependencies, and mirrored resources. Each time an Object is pulled, this server, if made public, then becomes a mirror where others can also discover and retrieve it. This is the federated model at work.

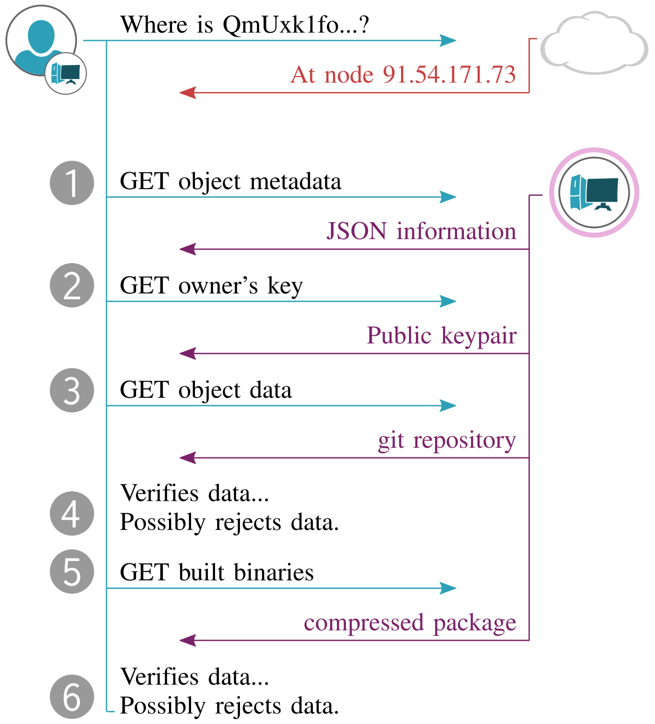

Fig. 1. The simplified steps taken when distributing a build across the federated network. In both Step 4 and Step 6, a key and signature infrastructure is used to verify both metadata and the built software.

Once a system has a copy of the Object data, that system must now provide a build before it can be used. Of course, the system could build the Object itself after collecting the dependencies required to do so. Alternatively, one can elect to pull down a previously built version. Similar to the pulling of metadata, a build is located and then pulled from the participating server, and then stored. Again, such a server can then potentially act as a mirror.

However, pulling a build from anywhere is potentially problematic. Any system can build an Object and provide the binary. To ensure proper accountability, the pulling system can verify the build by repeating the build process and comparing the resulting hash. However, another avenue we provide is to trust any particular Actor to provide a build. In this case, that Actor signs the build metadata, including the hash of the resulting binaries, with their personal key. Currently, Occam uses the widely-implemented PKCS#1 signature algorithm [36] as defined within the RSA standard. As described in our definition of Actor, the origin of their RSA key can be independently verified. However, nothing in the design prohibits future use of other types of signature methods. In the end, this creates a difficult-to-forge document that asserts the resulting binaries came from the given task as built by the given Actor. Considering this and that the signature is public information, any independent server acting as a mirror of data can also provide a mirror of the signature.

With this scheme, there are several levels of accountability that one can lean on. If you trust particular Actors, for instance the developers of known software, you can use these keys to trust binaries provided by them and them alone. After pulling a build of an Object, you hash the binaries independently and verify the signature matches both that hash and the key of the trusted Actor, as seen in the final step of Figure 1. Yet, you can still inspect the source code and rebuild the Object yourself, if desired. Since Occam can reuse the build task manifest, which lists each and every piece of software within the build environment, it can independently repeat that build. Any Actor that does so successfully may also elect to sign the build as well which will serve to echo that trust to others. Build repeatability, therefore, offers the flexibility to not rely on a centralized trust in a single entity to provide a built package.

Build repetition also ensures the burden to augment, tweak, and/or remix the source code is also reduced. Once you are able to reliably repeat builds, you also ensure that most minor modifications to the code will also build. This is a benefit of a preservation-focused system that is able to lock the build environment. Any developer can change the software code as though modifying it at its original point in time. Since the environment does not change between edits, the build, given the new code is valid, should still succeed. Of course, nothing stands in a developer's way if they wish to modernize the environment. Yet, considering the 28% of difficult builds and 14% of failed builds in the previously mentioned repeatability study, [1] the community has a strong motivation to avoid that being necessary.

IX. Overview

This paper has described several important aspects to a software preservation system geared toward the accountability required by scientific use. Such a system inherits many of the problems faced and solved by modern package managers. Yet, it also acquires a stronger need to preserve the reliable and deterministic repeatability of execution. In summary, there are aspects of discoverability, availability, and integrity of data.

In the preceding sections, with a focus on these particular issues, we have described Occam. Ultimately, this system is a set of tools that focus on providing software preservation and repeatable execution environments. Specifically, this tool provides a federated object storage tuned to the needs of software distribution. It also provides a flexible way of generating virtual machines or containers to run this software without committing to any particular tool. With consideration to its goal of avoiding existing as a centralized repository, it has a design based around a federated model. Every object, build of an object, and author in the system is represented with a universal key. Furthermore, such data is nomadic and can be mirrored at any location on the network.

Considering Occam's focus on general execution, there is still room to improve with respect to HPC concerns. There is much to be inspired by when looking at more specialized package managers, such as Spack [11]. This tool has mechanisms that allow one to ask not just for a particular version of a tool, but one that has been built a particular way. For instance, to use a simulator that was built with a particular version of a compiler built on or for an exact architecture. Occam does have limited support for constraining which builds are valid when specifying dependencies, but cannot do so at the same granularity as Spack. Occam has, instead, put more effort into providing features toward long-term preservation. Yet, features to give this level of control over build provenance are currently under development.

Nevertheless, when you have a robust enough system to distribute software and repeatably execute it, you can then build infrastructure on top of it. Occam itself extends its capabilities in several different directions on the foundation described in this paper. One logical step, after handling a single piece of software, is to provide a means of composing several together. Occam has a web-based portal that allows the graphical construction of workflows that simplify composing multiple pieces of software together. A program can take input in the form of a data object and create new objects in the system as a consequence of executing. These output objects can then be attached to others as input, and the process repeats. Each step of such a workflow generates its own repeatable virtual environment to run within, eliminating the headache of library compatibility between two programs. In fact, nodes on this workflow can have different architectures. For instance, programs running on Arduino that create data passed to a Python script running elsewhere.

Yet, many other tools and specialized science gateways can be formed as well. The important point is that a strong foundation is the first step to providing a stable platform on which such portals can be built. For instance, the climate science community could build a website that allows scientists the chance to use computational infrastructure to run their own experiments. Perhaps this allows citizen scientists to explore public data that is otherwise out of their technical reach.

This is a laudable goal that is only made stronger when built on top of a preservation system. The developers can focus on the needs of their particular community without developing features related to the preservation of scholarly work. Indeed, the question that arises when the portal must close is retroactively answered. All experiments will still be mirrored and repeatable since they were created by a platform derived from such a system.

This is the direction that seems most appropriate for any community. Specifically, a progressive path to let others build out infrastructure making use of HPC resources such as large clusters and supercomputers while limiting the technical burden required. All while maintaining the scientific importance of accountability and preservation as a byproduct. In particular, what we should want, and perhaps what we absolute need moving forward, is the ability to create this bridge.

References

[1] C. Collberg and T. A. Proebsting, "Repeatability in computer systems research," *Commun. ACM*, vol. 59, no. 3, pp. 62–69, Feb. 2016. [Online]. Available: http://doi.acm.org/10.1145/2812803

[2] "Arch user repository - archwiki," Aug. 2019. [Online]. Available: https://wiki.archlinux.org/index.php/Arch_User_Repository

[3] "About npm." [Online]. Available: https://docs.npmjs.com/about-npm/

[4] "OCCAM: Open Curation for Computer Architecture Modeling," http://occam.cs.pitt.edu, 2016, [Online; accessed 12-Aug-2016].

[5] M. Baker, "1,500 scientists lift the lid on reproducibility," *Nature*, vol. 533, pp. 452–454, May 2016.

[6] D. Wilkinson, L. Oliveira, B. Childers, and D. Mosse, "Evaluating Interactive Archives," 10 2017. [Online]. Available: https://figshare.com/articles/Evaluating_Interactive_Archives/5483836

[7] "Advanced package tool," https://salsa.debian.org/apt-team/apt, [Online; accessed 10-Oct-2019].

[8] "Npm - about semantic versioning," https://docs.npmjs.com/about-semantic-versioning, [Online; accessed 07-Sep-2019].

[9] "About nix," https://nixos.org/nix/about.html, [Online; accessed 11-Oct-2019].

[10] "Gnu's advanced distro and transactional package manager — gnu guix," http://guix.gnu.org/, [Online; accessed 11-Oct-2019].

[11] "Spack," https://spack.io/, [Online; accessed 07-Sep-2019].

[12] C. Lagoze, H. Van de Sompel, P. Johnston, M. Nelson, R. Sanderson, and S. Warner, "Ore user guide - primer," Open Archives Initiative, Tech. Rep., http://www.openarchives.org/ore/1.0/primer. [Online]. Available: http://www.openarchives.org/ore/1.0/primer

[13] J. A. Kunze, J. Littman, L. Madden, E. Summers, A. Boyko, and B. Vargas, "The bagit file packaging format (v0.97)," Working Draft, IETF Secretariat, Internet-Draft draft-kunze-bagit-13, January 2016, http://www.ietf.org/internet-drafts/draft-kunze-bagit-13.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-kunze-bagit-13.txt

[14] H. Meng and D. Thain, "Umbrella: A portable environment creator for reproducible computing on clusters, clouds, and grids," in *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*, ser. VTDC '15. New York, NY, USA: ACM, 2015, pp. 23–30. [Online]. Available: http://doi.acm.org/10.1145/2755979.2755982

[15] "ReproZip - About," https://vida-nyu.github.io/reprozip/, [Online; accessed 29-Aug-2016].

[16] "IPFS is the Distributed Web," https://ipfs.io/, [Online; accessed 29-Aug-2016].

[17] J. Benet, "IPFS - Content Addressed, Versioned, P2P File System," https://ipfs.io/ipfs/QmR7GSQM93Cx5eAg6a6yRzNde1FQv7uL6X1o4k7zrJa3LX/ipfs.draft3.pdf, [Online; accessed 03-Sep-2019].

[18] M. Ogden, K. McKelvey, M. Buus Madsen, and C. for Science, "Dat - Distributed Dataset Synchronization And Versioning," https://github.com/datprotocol/whitepaper/blob/master/dat-paper.pdf, [Online; accessed 03-Sep-2019].

[19] "Git Protocols," http://git-scm.com/book/en/v2/Git-on-the-Server-The-Protocols, [Online; accessed 05-Nov-2014].

[20] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, jan.-june 2011.

[21] "Gitlab." [Online]. Available: https://about.gitlab.com

[22] "GitHub: Build software better, together," github.com, [Online; accessed 12-Dec-2014].

[23] "Debian worldwide mirror sites." [Online]. Available: https://www.debian.org/mirror/list

[24] "Arch linux developerwiki - reproduciblebuilds," Apr. 2019. [Online]. Available: https://wiki.archlinux.org/index.php?title=DeveloperWiki:ReproducibleBuilds&oldid=571386

[25] "Lts - ubuntu wiki." [Online]. Available: https://wiki.ubuntu.com/LTS

[26] "GitHub Blog: 10 Million Repositories," https://github.com/blog/1724-10-million-repositories, Dec. 2013, [Online; accessed 29-Aug-2016].

[27] "Mercurial scm," https://www.mercurial-scm.org/, [Online; accessed 07-Sep-2019].

[28] "How git object deduplication works in gitlab," https://docs.gitlab.com/ee/development/git_object_deduplication.html, [Online; accessed 07-Sep-2019].

[29] V. Mart, "Github blog: Counting objects," https://github.blog/2015-09-22-counting-objects/#your-very-own-fork-of-rails, [Online; accessed 07-Sep-2019].

[30] "Semantic versioning 2.0.0," https://semver.org/, [Online; accessed 07-Sep-2019].

[31] "Vagrant by hashicorp," https://www.vagrantup.com/, [Online; accessed 07-Sep-2019].

[32] "hashdeep." [Online]. Available: https://github.com/jessek/hashdeep

[33] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[34] "Singularity," http://singularity.lbl.gov/, [Online; accessed 29-Aug-2016].

[35] P. Saha, A. Beltre, P. Uminski, and M. Govindaraju, "Evaluation of docker containers for scientific workloads in the cloud," *CoRR*, vol. abs/1905.08415, 2019. [Online]. Available: http://arxiv.org/abs/1905.08415

[36] J. Jonsson and B. Kaliski, "Public-key cryptography standards (pkcs) #1: Rsa cryptography specifications version 2.1," Internet Requests for Comments, RFC Editor, RFC 3447, February 2003, http://www.rfc-editor.org/rfc/rfc3447.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3447.txt