DARIAH Meta Hosting: Sharing Software in a Distributed Infrastructure

Jedrzej Rybicki and Benedikt von St. Vieth
Forschungszentrum Jülich GmbH, Jülich, Germany
{j.rybicki, b.von.st.vieth}@fz-juelich.de

Abstract - Research infrastructures have become an everyday tool for doing science. They constitute a cost-effective, quick, and increasingly easy-to-use collaboration tool. So far the focus was on sharing resources (especially data) and on offering a (rigid) set of services for processing and accessing the resources. It becomes clear, however, that there is a demand from the users to share not only the data they have gathered or created but also the software they implemented. Such a sharing has the potential to speed-up the scientific discovery. But only if the software runs, i.e., if it can be applied by other researchers to new problems. Unfortunately, it happens often that the software implemented in a project is only understandable and deployable by the authors. In this paper we will address the problem of sharing services (e.g. new data analysis tools) between users and also touch on the problem of sharing services between infrastructures, for instance to facilitate cross-disciplinary exchange. Our goal was to increase the sustainability of the developed research software by enabling an easy extension of research infrastructures beyond the rigid set of services towards flexible Software-as-a-Service (SaaS) solutions. We share the initial experiences gained during the implementation of such a meta hosting service for DARIAH-DE research infrastructure.

I. Introduction

Digital methods have become commonplace in modern science. Their popularity is amplified by the recent establishment of the so-called data-driven science, which is expected to provide scientific insights by applying software solutions on digital data. The obvious prerequisite for such approaches is the availability and accessibility of the data. A problem which is acknowledged and addressed (if not-yet-completely-solved). The less obvious problem is how to share the scientific software used for analyzing the data. Such software is often a custom-made solution, developed in the course of a research project as an after-effect of answering concrete scientific questions. Other researchers might, nevertheless, benefit from using it to solve similar problems or for the validation of the results by applying the software to a different set of data [1]. Sharing software has the potential to accelerate scientific discovery. But even more important is the fact that software which is not sharable is, in the long run, not sustainable. Scientific discoveries done with such a software are, in worst case, not reproducible and thus useless. We believe that sharing data and software is a cornerstone for doing science today. In this paper we will brush aside the social challenges of sharing software and data, and concentrate on technical solutions to facilitate such sharing.

It should be stressed that the open software repositories like *GitHub* or *sourceforge* do not solve the stated problem completely. The software is only usable when it runs. Installing, configuring, and satisfying the software dependencies often proves to be harder than expected. Especially when it is conducted on a different platform than it was developed on. Furthermore the knowledge on how to perform these steps is often only present in the heads of the software authors or within the project it was developed for (e.g. closed wiki). Hence the first requirement for the effective software sharing is the availability of the abstract, platform-independent, yet executable deployment descriptions. Such descriptions could be, for instance, system images, executable installation scripts, configuration management tools, or container-based solutions. We will discuss the applicability of these approaches in Section II.

The proliferation of the distributed research infrastructures like [2] and [3] adds a new dimension to the problem of sharing scientific software. Such infrastructures offer a quick and cost-effective way of sharing resources. They excel at enabling access to data and often provide some compute resources. The later is offered either in a Infrastructure-as-a-Service or Software-as-a-Service manner (for the definition of the terms see [4]). None of these approaches is optimal from the user perspective. IaaS solutions put quite a large burden on the users: they have to install, configure, and maintain not only the software they require but also operating the whole system (e.g. do system upgrades). Furthermore, the created instances cannot be easily shared: this remains in the power of the user who created them and is usually technically challenging. The higher-level abstractions in form of Software-as-a-Service are much more attractive for researchers: The effort on their part is reduced and they can directly apply the software in their scientific endeavors. But the effort still exists: The installation work has to be done by the computer centers in the infrastructure. From software sharing perspective the problems are twofold. Firstly due to the rapid development of digital methods, researchers constantly require new versions or new types of software in the SaaS portfolio. Secondly, the distributed research infrastructures span across multiple resource providers, thus a high level of heterogeneity has to be coped with, making the installation harder. To this end, the demanded software deployment descriptions must be platform-independent and portable. To account for the constantly changing requirements, the descriptions should be executable. Finally the system administrators should be able to audit them to verify if they are correct, trustworthy, and secure. In this paper we will show how the deployment descriptions can be used to build a dynamic Software-as-a-Service solution, which allows for an easy extension of the offered software portfolio.

The high development pace of cooperative science together with the increasing importance of interdisciplinarity, bears one more use case for sharing research software: exchanging software between e-Infrastructures. This use case reinforces the previously mentioned requirement of making software deployment descriptions executable, auditable, portable, and sharable. The potential users of research software from different disciplines will have to invest some time to understand how to use the software. The time should not be increased by making the user understand how to install, and deploy the software.

This paper is structured as follows. Related work is discussed in Section II. We provide a short introduction in the technology which we will use as a basis for the software sharing in Section III. Section IV presents how we used executable software deployment descriptions to implement an extensible Software-as-a-Service solution for DARIAH-DE. We conclude our paper with a summary in Section V.

II. Related Work

Let us now reflect on how software reusing is typically achieved nowadays. In the first step the scientific software has to be made discoverable. This step involves some social challenges, for instance willingness to share and use software, or efficient communication between the parties, involving software citation. In this paper, however, we focus on the technical challenges and claim that open-source repositories are sufficient for discovery. As stated above, the software is only useful if it can be applied to a new problem, and this is only possible when the software runs. Hence, the discovered software has to be installed. The process is conducted either locally on the researcher's machine or (in a more modern fashion) on a remote machine running somewhere in the e-Infrastructure. The installation usually involves the configuration of the operating system (OS) and the provision of the run-time environment, additional libraries, and actual software. The knowledge on how to achieve this might be, to some extend, conveyed in the service documentation. The experience shows that the documentation is seldom up-to-date and does not cover all the corner cases encountered in the real-life deployments e.g., library dependencies or support for differing operating systems. When it comes to a migration of the service, the process must be started from scratch again, often by a different person. Even such a down-to-earth, recurring situation like the OS upgrade for the underlying machine might render a service unusable and result in the need for a re-installation. Since the process described above is quite laborious, let us now discuss ways in which it can be either simplified or done in such a way that other researchers can use the software without repeating the installation process.

A. *Virtual machines*

One way of sharing "already-installed" software is to prepare images from which virtual machines can be created. The most obvious case for such applications are clouds offering an IaaS interface [5][6], although local deployments of images are conceivable as well. VM image preparation involves repeating the above steps in a virtual machine while accounting for special properties of the given cloud. The later is often subsumed under the term: contextualization. When an image is created and uploaded to a cloud image repository it can be instantiated by other users to obtain running version of the application. VM images suffer, however, under some limitations. First of all, images are black boxes: one has to trust the creator and has very limited capabilities of auditing the actual content of the image. Thus it is hard to safe-guard that the images do not include some malicious software or are not misconfigured so that running instances can be easily hacked or destroyed by malicious third parties. Virtual images include a complete software stack comprised of the required software, dependencies, and operating system. This results in a high overhead in terms of both image size (what makes the sharing of images harder) and performance. The inclusion of the operating system results in the need for periodic updates of the image: for instance to apply new security patches. The further problem with images is, that they are infrastructure-specific: they strongly depend on the hypervisor used and contextualization solution provided in the particular cloud. Clouds include their own image repositories but moving images between the infrastructures i.e. between different clouds, remains an unsolved problem. A problem with obvious ramification for software sharing.

B. *Configuration management tools*

One way to create trustworthy and reproducible service deployments that are, to some extent, infrastructure, hypervisor, and OS agnostic, is to use configuration management tools like Puppet [7] or Chef [8]. With these tools it is possible to describe the service deployment in a declarative way. One defines which packages should be installed, which configuration changes should take place or which commands should be executed instead of applying all these steps manually. The tools can be used to provide a generic description of the service deployment that interested partners can apply to virtual machines or servers as the configuration management tools do not offer any isolation nor virtualization themselves. Here the same limitations with respect to overhead of running VMs apply. On the upside, the descriptions are human-readable and can be reviewed before the actual deployment, this contributes to their trustworthiness. Configuration management tools enable reproducible service deployment, but they fail to some extend at abstracting the underlying platform, for this the developer has to spent additional effort.

One example for this is the deployment of a web server like Apache HTTP server [9], a software often used to provide web services. Listing 1 shows an excerpt from a deployment description that is used to deploy the software by installing (section *package* on Listing 1), configuring (*file*), and starting (*service*) it. Puppet brings an abstraction for package managers and it also provides wrappers for different *init* systems, but it has problems with the different package, path, and service naming. To keep it platform-independent and portable, one has to prepare deployment descriptions for all the supported platforms. The first lines of the Listing 1 show how to cope with the fact that the same software is available under different names in RedHat and Debian repositories.

```
if $::osfamily == 'RedHat'{
   $user = 'apache'
   $name = 'httpd'
 } elsif $::osfamily == 'Debian' {
   $user = 'www-data'
   $name = 'apache2'
 } else {
   fail("Unsupported os: ${::osfamily}")
 }
 file { "/etc/$name/$name.conf":
   owner => $user
   ensure => 'present'
 }
 package { $name:
   ensure => 'latest'
 }
 service { $name:
   ensure   => 'running'
 }
```

Listing 1. Puppet example for deploying Apache httpd,

We establish that tools like Puppet solve a slightly different problem. They are perfectly suited for managing the configuration of software running in a distributed infrastructure. For this purpose, they allow for configuration changes of the running machines (such changes are then automatically loaded on all relevant machines). The scientific software sharing, on the other hand, will be rather used to create quickly disposable instances of given services. These instances will be used e.g. to verify the researcher's hypothesis or validate previous results and they will be discarded afterwards. We argue that there will be no need to manage the configuration changes in one instance, for the modified setup new instance will be spun off. Researcher will not be willing to spend much effort into the configuration management when it would be possible to simply, quickly, and cheaply create new instances.

III. CONTAINER-BASED VIRTUALIZATION

Container-based virtualizations like Linux VServer [10] or Docker [11] became very popular for building, shipping, and running applications across many machines. They constitute the perfect basis for an efficient sharing of scientific software. In this section we will first provide the reader with some basic information on how Docker works, and argue why it is better than the previously described system images and configuration management tools. The section will be concluded with a proposal on how the software sharing life cycle could be implemented in a distributed infrastructure.

A. *Introduction to Docker*

Docker is used as the basis for our software-sharing solution. This section will equip the reader with enough information about this technology to understand the rest of the paper. Readers interested in more details are referred to the extensive Docker documentation [11].

Docker is a lightweight, open-source, virtualization solution. In opposite to the full-stack virtualization solutions, Docker images do not include a guest operating system kernel. It lowers the overhead in terms of both performance and size of the images allowing near-native performance. Docker is based on mature Linux kernel technologies (*cgroups* and *namespaces*, among the others) to isolate independent application containers. Docker images use AuFS which is a layered file system, enabling sharing libraries between images on one hand, and inspecting the changes done in the images, on the other. The latter feature permits rudimentary provenance tracking of images.

Let us now briefly discuss the Docker terminology and usage. Applications running with Docker are called containers, they are created from images. Images should be understood as hierarchical templates. It is possible to start with one image, add some software, and save the result as a new image. It is also easily possible to share images by using image repositories either private or public ones, like the *Docker Hub*. There are two main ways of shipping applications with Docker. In the first one (let us call it interactive), the developer starts a basis Docker image (e.g.: official Debian Linux image) and installs application, its dependencies, and everything else by issuing ordinary Linux commands. As soon as the application is installed, a snapshot of the running container can be created. Such a snapshot is, in fact, an image from which new containers can be created. Another way of creating images follows a imperative approach. The developer can describe the installation steps in a special *Dockerfile*. An image is created from such a description by issuing the *docker build* command. A generalization of the imperative approach are automatic builds. To setup an automatic build one has to publish the *Dockerfile* in a public code repository (like *GitHub*). Afterwards a web hook has to be created, so that each time the *Dockerfile* is modified, a new image build is automatically triggered. Regardless of the way in which an image was created, it is possible to view all the changes done in the image during the installation of the software. The automatic builds go an extra mile to improve the trustworthiness of images: each user can review not only the content of the image but also the way they were created, since the description is publicly available.

Docker has a very important abstraction of data volumes which allows to separate software and data. The data are not included in the image, a problem often encountered in VM images. Data volumes, on the other hand, enable an easy injection of the data into the running containers. Hence the software in the container can easily be used for analyzing various sets of data. The separation of the software and the data diminishes one of the common barrier for sharing software. Researchers are afraid of leaking out the data in the software they made available.

Docker is available on almost all Linux-based platforms, there is a support for MacOS and recently announced support on Windows platforms. Since the images encapsulate the application and all the dependencies, starting new containers on a new system is like starting statically linked, self-contained programs: no modification in the host system are required.

B. *DARIAH-DE software sharing life cycle*

DARIAH, the Digital Research Infrastructure for the Arts and Humanities, aims to enhance and support digitally-enabled research and teaching across the arts and humanities. It comprises a number of national initiatives, there is also a Germany-based effort, DARIAH-DE [2], where the described work was done. Efficient software sharing is essential for digitally-enabled research of its users. Also sharing software between infrastructures is becoming relevant in this context [12]. Software developed in one part of DARIAH might be passed over to other national efforts.

The technology for creating executable and deployable descriptions of software must be integrated into the scientific workflows and play well with existing e-Infrastructures. On Fig. 1 we present how such a integration could be conducted. We differentiate between three roles: developers, infrastructure operators, and researchers. We have also two central components: a code repository, where the deployment descriptions are stored, and Docker Hub repository where the images are stored and pulled from.

The process of making scientific software available kicks off with the developer wanting to advertise and make his software available. He can propose the deployment description by forking the official *GitHub* repository of DARIAH-DE Docker files, adding a new description (or updating an existing one), and creating a pull request[1]. The operators of the repository are notified about the pull request. They can review the proposed solution, adjust it if required, or even reject it if it does not work. The *GitHub* repository is connected to the Docker Hub via web hooks. In short, this mechanism triggers an HTTP request to the Docker Hub each time the *GitHub* repository is modified. Upon such a request, a new build of the provided Dockerfile is started. The infrastructure operators can view the details of the build and act accordingly in case of a failure. Successfully built images are published into the official DARIAH-DE image repository in the Docker Hub. Finally, the researchers can explore the repository, select the software they are interested in, pull the images and start it on their machines. Pulling and starting requires just one command (*docker run*). The Docker Hub allows for storing additional information about the software. The developer can describe the typical applications or provide a link to a more extensive documentation.

The workflow described above can be repeated multiple times e.g., each time a new version of the software is made available. Moreover, the roles can be distributed or handed-over. Both repositories allow for registration of organizations. Such organizations can have multiple admins. Hence it is not required to have just one

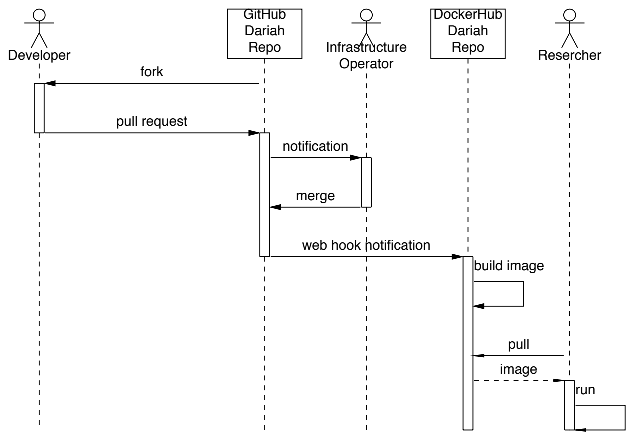

Figure 1. Proposed software sharing life cycle

infrastructure operator but rather a group of people can be delegated for this task. Similarly, all *GitHub* users can fork, and modify the Dockerfiles. The trustworthiness of the images is guaranteed by the reviews conducted by the operator who has to accept the pull requests. Of course each researcher can view the publicly available descriptions. The images (and Dockerfiles) are publicly available and they can be used by researchers from outside of DARIAH-DE.

Fig. 1 depicts the simplest workflow, which ends with a local deployment of the software. It is sufficient to explain how the workflow works but clearly the local deployment is not always the optimal solution. In the next section we will show how a dynamic SaaS solution can be built, based on the proposed workflow.

IV. IMPLEMENTING META HOSTING

We have shown that Docker provides a means for a fast deployment of software. In the optimal case, the user has to issue just one command to obtain a running instance on her system. The optimal case requires some changes on the machine, at least the installation of Docker is required. Even if easily possible, the local deployments are not always the best option. One reason for remote deployments is efficiency: remote instances can run close to the resources they require. Many data-driven projects require long-lasting software runs. This is hard to perform on the local researcher's machine. Fortunately, e-Infrastructures like DARIAH-DE provide resources which can be used by the researchers for this purpose.

In this section we will show how the presence of deployable software description (as those provided by Docker) can facilitate a dynamic Software-as-a-Service solution. This service will offer all the benefits of the SaaS approach (i.e. very low effort is required to use scientific software shared by other researchers) whilst allowing for quick extensions of the software portfolio. We present our prototype solution which runs on low-level compute resources already available in the DARIAH-DE infrastructure. Our goal was a system which provides the researchers with an instance of the software she requires in just one request through a web page.

1 Readers not familiar with the *GitHub* commands are referred to the documentation [13]

A. *Actors*

Fig. 2 depicts the architecture of the implemented dynamic Software-as-a-Service. It includes lots of components loosely coupled with each other by a messaging bus. Messaging is used for exchanging information between components in an asynchronous way. In production we use an *amqp*-based product (RabbitMQ [14]). Let us briefly discuss the roles and responsibilities of the single elements of our architecture.

The *Facade* is the entity facing the end-users. It is an entry point from which the commands (like create a new instance of a given software product) are issued. The Facade is stateless and it is possible to run multiple Facades. They can offer interfaces of different types, e.g.: an html-based for browser access or a json-based programmatic API.

The Facade has access to the messaging bus and a read-only access to the *Store*. Store is a system-wide cache with information about:

1. types of service registered in the system,
2. instances running in the system.

The content of the cache is updated by the *Store Updaters*. These entities subscribe for information about creation of new instances and registration of new types. Upon reception of such messages, Store Updaters write the information in the Store. It should be stressed that the content of the Store itself can be lost as it will be periodically republished. The Store is solely used to speed-up the response times.

The *Workers* embody the main functionality of the system. Those are the entities which can manage instances of given service types (e.g. instances of eXist or neo4j databases). Each Worker upon its creation (and later periodically) informs other components about its capabilities (i.e. supported service type) and its current state (i.e. managed instances). Workers receive commands from users (via Facade) through the messaging system.

B. *Communication with workers*

The critical part of our system is the communication between the components. It is dynamic, asynchronous, and subject-based. We require the possibility to change the system capabilities during run-time. In particular, to add new software to the portfolio of the offered Software-as-a-Service. This is done by adding (or removing) Workers. To become active in the system, Worker needs access to the messaging bus. The incoming Worker publishes its "specialty" (i.e. instance type that it can provide) and subscribes for the messages in this instance-type-queue (like create new instance, delete instance). As can be seen on Fig. 2, for each instance type supported by the system, there is a separate queue. When a user requires a new instance of a given type it sends a message to the respective queue (via Facade). The message is routed to the proper Worker, it creates an instance and sends a response (via messaging) with details required for connecting to the just-created instance. Such details typically include IP address, protocol and port, user name and password, etc. Workers republish their state periodically (i.e. instance types that they support and state of managed instances). Each worker has a dedicated queue for the incoming commands and all workers share a common response queue (the *info* queue on Fig. 2).

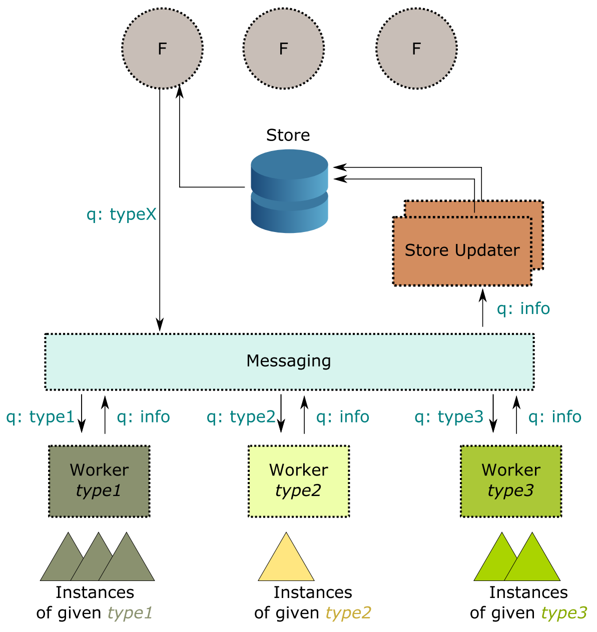

Figure 2. Architecture of the dynamic Software-as-a-Service

C. *Implementation and operation*

We have created a prototype of the dynamic Software-as-a-Service called DARIAH Meta Hosting. The implementation was done in Python and uses a set of frameworks and well-known products: Flask for the front end, MongoDB as storage, and RabbitMQ for messaging. We offer two interfaces: one is a human-friendly web site, the other is a programmatic json-based API. We envision the later one to be used to script automatic deployments of software e.g., in data processing workflows.

One feature of our design was the division of the system into small, autonomous parts connected by messaging. The advantage of such a design was the ability to change the parts independently, for instance to scale up. We use Docker for deploying single parts and the Docker-based orchestration tool *docker-compose* [15] to create test deployments including all the parts required. Docker-Compose is also able to scale out the system by spinning off additional instances. In this self-test, the solution we use for the software sharing was successfully applied for the software we have implemented. Our prototype is deployed on a OpenStack Infrastructure-as-a-Service cloud which we offer in DARIAH-DE. So far the complete system is placed in one data center but it should be possible to extend it beyond this one entity. This would only require to make the messaging accessible from the outside and deploy additional workers in different data centers.

For our prototype we have implemented a generic Docker worker. The deployment of Docker-based instances is usually conducted in the same fashion, regardless of what software is confined in the image. Hence the Worker is initialized with a name of the Docker image containing the software it will be responsible for. We create an instance of Worker for each software package offered in the dynamic Software-as-a-Service. To scale out the system it is also possible to have multiple workers offering the same type of the software. The requests will then be divided among the workers in a Round-robin fashion, allowing for load balancing.

At this stage it should be stressed, that the proposed architecture is extensible. The Workers abstract the technology used for creating the instances. It would be perfectly possible to write a worker which would use Puppet-based deployment descriptions or any other technology that might come around in the future.

V. Conclusion

The motivation for this paper was the challenge of scientific software sharing. A solution for that problem is crucial for the further development of reproducible data-driven science. We have argued that the problem is many fold. It involves both technical and social challenges. In this paper we have focused on the former ones and firstly reviewed the technologies which could be used to leverage the software sharing. Subsequently we have presented a workflow for sharing software in the context of e-Infrastructures like DARIAH-DE which ensures high grade of trustworthiness. In the second part of our paper we have shown how the presented software sharing solution can be used to implement a dynamic Software-as-a-Service system. It enables a quick and easy deployment of the software on IaaS resources available in the e-Infrastructure. The proposed architecture is extensible and we have also shared some experiences gained during the implementation and operation of a working prototype.

Acknowledgment

The work has been supported by DARIAH-DE, which is partially funded by the German Federal Ministry of Education and Research (BMBF), fund number 01UG1110A-M.

References

[1] C. Goble, “Better software, better research,” IEEE Internet Computing, vol. 18, no. 5, pp. 4–8, Sept 2014.

[2] T. Blanke, M. Bryant, M. Hedges, A. Aschenbrenner, and M. Priddy, “Preparing DARIAH,” in IEEE 7th International Conference on E-Science, pp. 158–165, Dec 2011.

[3] D. Lecarpentier, P. Wittenburg, W. Elbers, A. Michelini, R. Kanso, P. Coveney, and R. Baxter, “EUDAT: A new cross-disciplinary data infrastructure for science,” International Journal of Digital Curation, vol. 8, no. 1, pp. 279–287, 2013.

[4] P. Mell and T. Grance, “The NIST definition of cloud computing,” National Institute of Standards and Technology, Tech. Rep. 800-145, 2011.

[5] “Amazon Web Services,” http://aws.amazon.com/.

[6] “OpenStack,” http://openstack.org/.

[7] “Puppet,” http://puppetlabs.com/.

[8] “Chef,” https://www.chef.io/.

[9] “Apache HTTP server project,” http://httpd.apache.org/.

[10] “Linux-Vserver,” http://linux-vserver.org/.

[11] “Docker,” https://www.docker.com/.

[12] M. Hedges, H. Neuroth, K. M. Smith, T. Blanke, L. Romary, M. Küster, and M. Illingworth, “TextGrid, TEXTvre, and DARIAH: Sustainability of Infrastructures for Textual Scholarship,” Journal of the Text Encoding Initiative, Jun 2013.

[13] “GitHub,” https://github.com/.

[14] “RabbitMQ,” http://www.rabbitmq.com/.

[15] “Docker Compose”, https://docs.docker.com/compose/.