A Framework for the Preservation of a Docker Container

Iain Emsley
Oxford e-Research Centre

David De Roure
Oxford e-Research Centre

Abstract

Reliably building and maintaining systems across environments is a continuing problem. A project or experiment may run for years. Software and hardware may change as can the operating system. Containerisation is a technology that is used in a variety of companies, such as Google, Amazon and IBM, in addition to scientific projects to rapidly deploy a set of services repeatably. Using Dockerfiles to ensure that a container is built repeatably, to allow conformance and easy updating when changes take place, are becoming common within projects. It's seen as part of sustainable software development. Containerisation technology occupies a dual space: it is both a repository of software and software itself. In considering Docker in this fashion, we should verify that the Dockerfile can be reproduced. Using a subset of the Dockerfile specification, a domain specific language is created to ensure that Docker files can be reused at a later stage to recreate the original environment. We provide a simple framework to address the question of the preservation of containers and its environment. We present experiments on an existing Dockerfile and conclude with a discussion of future work. Taking our work, a pipeline was implemented to check that a defined Dockerfile conforms to our desired model, extracts the Docker and operating system details. This will help the reproducibility of results, by creating the machine environment and package versions. It also helps development and testing by ensuring that the system is repeatably built and that any changes in the software environment can be equally shared in the Dockerfile. This work supports not only the citation process, but also the open scientific one by providing environmental details of the work. As a part of the pipeline to create the container, we capture the processes used and put them into the W3C PROV ontology. This provides the potential for providing it with a persistent identifier and traceability of the processes used to preserve the metadata. Our future work will look at the question of linking this output to a workflow ontology, to preserve the complete workflow with the commands and parameters to be given to the containers. We see this provenance as useful within the build process to provide a complete overview of the workflow.

Introduction

Reliably building and maintaining systems across environments is a continuing problem, as a project or experiment may run for years. Software and hardware may change, as can the operating system. When a paper is written, the software, in addition to its environment and parameters, may not be fully described, contributing to problems with reproducibility. Various approaches are suggested to tackle this, such as providing virtual machines, with the full installations and containers, with the software described.

Containerisation is a technology that is used in a variety of companies, such as Google, Amazon and IBM, and scientific projects to rapidly deploy a set of services repeatably. The long-term science archives of the SKA are designed to store data for 50 years, or the lifetime of the SKA. What is not defined is how the software used to process the data and create the science data objects might be preserved to show the processes and version used at a particular moment.

Software containers, virtualised processes running within an operating system, provide an alternative method of glossing the contained software and processes. Running within a jailed environment directly on a host environment, containers have become popular as a method of virtualising services. Docker[1] is currently the dominant container project.

As the Docker project is adopted into workflows, it is being used to publish their construction and to provide the environments to repeat experiments and show parameters. Though this provides the raw data to re-run an experiment, containerisation technology occupies a dual space: it is both a repository of software and it is software itself. In considering Docker in this fashion, we should verify that the container can be reproduced, that the Dockerfile is linked and its original environment is recorded.

We present related work before discussing our framework: beginning with the preservation of the Dockerfile, the container's provenance, and capturing the build environment. The pipeline is run as part an experiment in the Integration Prototype for the Square Kilometre Array (SKA). We discuss future work before concluding. Our contribution is showing the workflow to create the Docker container and linking it to a scientific workflow.

Related Work

The Recomputation Manifesto (Gent, 2013) argues for the use of virtual machines to contain software artefacts for future reproducibility. Maintaining the virtual environment also requires having software to run it again, and the hypervisors used to run the Virtual Machine provide sandboxes to limit access the underlying hardware.

Kuzak (2015) describes the use of containers within sustainable software processes.

The Bioinformatics field uses Docker as a method of reliably building pipelines that may require older software versions (Di Tommaso, 2015). Boettiger (2015) argues for the project's use in the reproducibility, prevention of code rot, and documentation for how a system is built.

In Radio Astronomy, Square Kilometre Array (SKA) precursor projects, such as the Low Frequency Aperture Array (LOFAR) (Molenaar, 2014) and Meerkat telescopes, use

---

1   The Docker Project: https://www.docker.com/

Docker containers for their pipelines. The SKA South Africa team have released Kliko, a manager for input and output parameters that can be loaded within a Docker container, but this assumes that the process runs in one container.

The Smart Container ontology (Huo, Nabrzyski and Vardeman, 2015) links the Dockerfile to the container, but does not look at the feasibility of preservation through migration, or linking this as a citable object. Our work develops the preservation aspect and provides a provenance of the processes employed to create the container and from which the container is created.

A Framework for Docker Containers

Unlike other container projects, such as LXC[2] or Rkt[3], Docker provides two forms of providing metadata to build the system: Dockerfile or the Docker Composer tool. Our work focuses on the Dockerfile, as it:

> '...serves not only as the recipe for Docker to build your image but as a comprehensive description of the complete environment your software requires to be able to run' (Haines and Jay, 2016).

Dockerfiles are being used as part of experimental notebooks and to share the software. Using these files ensures that a custom container is built repeatably, to allow conformance and easy updating when changes take place. This is becoming common within projects and pipelines (Kuzak, 2015). Haines and Jay (2016) associate digital object identifiers with the container for citation purposes.

Placing the Dockerfile in version control follows good software engineering practices (Matthews, McIlwrath, Giaretta and Conway, 2008) and provides a history through versioning. Our work builds on this by looking at preservability aspects of the Dockerfile and container, whilst providing a provenance for the processes used to build the container.

We present a framework that builds on existing practice to begin considering the container as a Research Object, following (Bechhofer et al., 2013). Current practice sees the Dockerfile as a recipe for the installation, or providing a link between the container and Dockerfile. Our approach links these steps together and also provides a provenance of the build environment, treating the container as software and a container, and allows for inference upon the described files to see if software and system policies are being maintained.

We derive the properties of software composition, provenance and ownership from Matthews, Shaon, Bicarregui, and Jones (2010) in a given Dockerfile and processing pipeline. We add the container environment of the container and workflow to our framework in Table 1.

---

2   LXC: https://linuxcontainers.org/
3   Rkt: https://coreos.com/rkt/

Table 1. Docker container properties.

Container Properties	Definition
Software composition	• Description of the components in the container
Provenance and ownership	• Processes used to build the container
	• The maintainer of the Dockerfile
Container environment	• The environment used to construct the container
Workflow	• The workflow with the configuration details for the pipeline

The software composition is not only the description of the components, but also the commands defined by the metadata and its maintainer. The owner of the file and container may be two different entities: one being an individual, the other an organisation or project. We also capture the environment in which the container was built.

As a part of the pipeline to create the container, we capture the processes used and put them into a file using the W3C PROV (Lebo, 2013) ontology. Provenance is defined "as a record that describes the people, institutions, entities, and activities involved in producing, influencing, or delivering a piece of data or a thing"[4]. As argued, the Dockerfile provides the entities to make the software in the container and the consideration of the entities and processes used to make the container itself.

An automated build system, such as Jenkins or Travis CI, can be used to transform the metadata file into the container system, tag it and store it in the repository. Though the build system provides a provenance of the commands given to it, it cannot capture the environment used if the machine uses a distributed build across slave machines. The provenance can be used not only to provide the provenance, but also to infer the integrity of the machines as social entities running the commands.

Software Composition

Using a subset of the Dockerfile reference[5], a domain specific language is created to ensure that Dockerfiles can be reused at a later stage to recreate the original environment.

Our focus is on the operating system, maintainer details and the package versions required to allow for the preservation of the information for a migration to a different system and to allow the exact packages to be built and reused. This is done to limit the chance of the environment changing the performance of the software.

The initial model is to extract the operating system name and version and the packages from the Dockerfile and to mark these up within the Dockerfile namespace. The operating system is extracted from the FROM command to check that the operating system name is defined and that a version is installed. The Docker specification allows

4 W3 - Provenance: https://www.w3.org/TR/2013/REC-prov-dm-20130430/#dfn-provenance
5 Dockerfile reference: https://docs.docker.com/engine/reference/builder/

for the operating system version to be defined, such as 16.04 for Ubuntu, or to have the keyword 'latest' to install the latest version. Both install the image of the operating system but the use of 'latest' harms repeatable builds and the container as a research object, as the latest version may be different across installations over time.

The maintainer details are represented using the Friend of a Friend (FOAF) namespace. The name and mailbox details can be extracted from the MAINTAINER command and placed into the graph.

The software package details are added to the graph so that dependencies can be inferred across other containers.

Provenance and Ownership

The Functional Requirements for Bibliographic Records (FRBR) ontology is also used to provide a link between the container and the Dockerfile, instead of PROV Agent. The Group 1 and Group 2 options are used to link the artefacts together and the people who are involved in the process.

Using FRBR provides two senses of ownership: the maintainer of the metadata file and the organisation that owns the container. In a distributed project, the two may not be the same entity, or the maintainer may work for another institution, such as a university or company, that is distinct from the project identity.

Container Environment

The machine environment on which the container is created is recorded as part of the workflow. The version of the Go language and the underlying operating system details are captured, providing further information about the environment that built the container. As the pipeline knows the Dockerfile and the container it is to build, we can provide a provenance between the Dockerfile and that particular container at that moment.

This provides a framework to address the question of the preservation of containers and its environment, storing it as a file. Capturing the machine environment and package versions helps development and testing by ensuring that the system is repeatably built and that any changes in the software environment can be equally shared in the Dockerfile. This work supports not only the citation process, but also the open scientific one by providing environmental details of the work.

Capturing the processes also provides information that can be used to invite further questions about the underlying system and people involved in the production of the container.

This provides the potential for giving it a persistent identifier and traceability of the processes used to preserve the metadata. Building on this allows not only the preservation of the particular container, but all the metadata files to be queried. This supports the use for publications and the science support but also the engineering processes.

Workflow

The SKA is reviewing various options for creating the workflow. The SKA South Africa team created Kliko[6], a Scientific Compute Container specification, to capture the

---

6 Kliko: http://kliko.readthedocs.io/en/master/introduction.html

parameters being used within a container. Parsing the Kliko file and linking it to the container graph, provides a method of viewing the linked containers and also the processes linked to them. We must be flexible enough either to link one container to a workflow, or many containers for one workflow.

Working with the SKA Integration Prototype

Methodology

A build and deployment system was created as the outcome of our work with the SKA Science Data Processor (SDP) Integration Prototype team, using machines within the Oxford test bed.

The driving purpose of the build system is the repeatable build of machines: to install the operating system with a standard user and starting state that allows for connections to a central Docker registry instance. Although this creates a known base state, we are still challenged by the ingestion of software into the central registry from geographically spread out and heterogeneous SDP development teams.

Github is used by the SDP development teams. We experiment with using Continuous Deployment pipelines to ensure that the Dockerfile could be automatically built into a Docker container and stored, as shown in Figure 1. Using a build tool such as Jenkins provides some metadata that is stored, however this is limited.
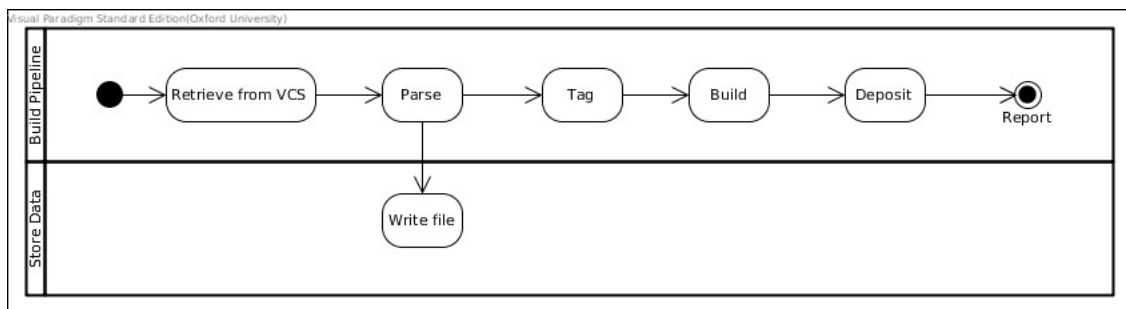

Figure 1. Activity Diagram for the Framework Pipeline.

Rather than extracting the metadata outside of the automated pipeline, we derive it within the pipeline as the containers are being built, tagged and stored and also capture the environment. Attaching this process to a particular branch of the version control system forms a discrete, automated pipeline per container and Dockerfile to create the local metadata. The aim is to place provenance as part of the automated software processes instead of it being an adjunct process or reconstructed at a later stage, such as from log files when data is processed.

Two Jenkins projects are set up: one to retrieve the Dockerfile from version control, and the second to run the build of the container on success of the first one. The latter runs the Python scripts (Emsley, 2016) to write the build process and software metadata files as part of the project.

An instance of the Jenkins server was set up to build an existing Dockerfile for the OSKAR[7] project. A Freestyle project is set up to download and install the Dockerfile from version control and is linked to a build project called Continuous Build that is triggered when the file is installed correctly.

The two generated files provide a snapshot of the container at the moment. The software graph, as shown in Figure 2, supports the browsing of the packages contained for installation. The desired outcome would be the ability to extract the version number of the package from the file to ensure that a repeatable build can be constructed, instead of assuming that the package does not change between installations. Over long running projects build systems may change, so the metadata should support migration from one system to another without losing precision of installation. This does place an onus on the package and software maintainers to ensure that versions numbers are put into the software.
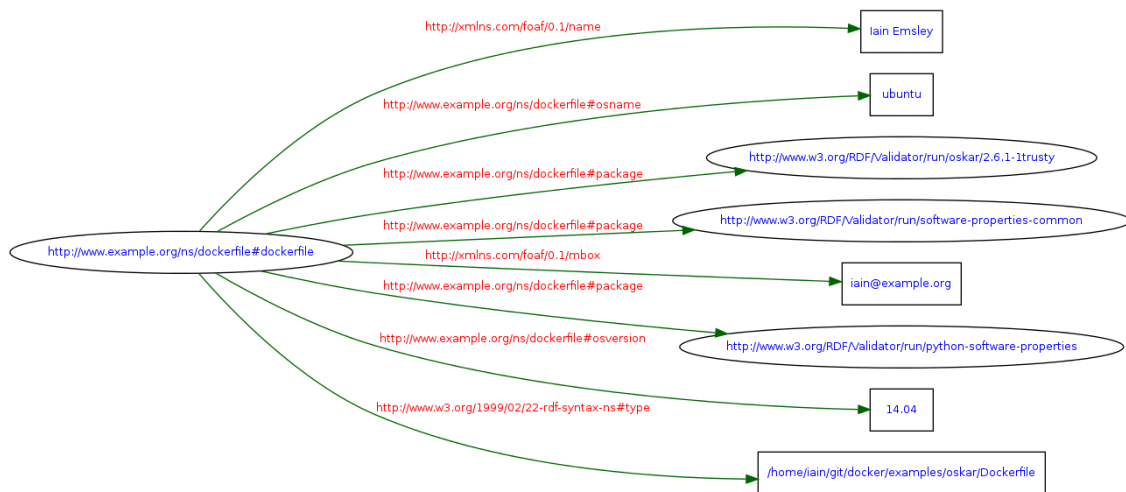
Figure 2. Graph of the software generated.

The process is not only about providing support for future systems, but may also support a project's existing software practices and policies. Although the SKA is still defining processes, an operating system has been chosen. If the metadata is being used for integrity of the container, as software it should also follow similar guidelines to the software development process. For the SKA, Ubuntu is chosen as the operating system, using Long Term Support builds. Our pipeline would prevent builds where an operating system does not have a version or if the version is not one of the accepted versions and where a maintainer is not provided. In so doing, we use the provenance to not only support potential migration of software but to also support inference for the project software processes being followed.


Figure 3. Graph of the process, entities and links in the container generation process.

Providing the entities, environments and processes in the build process, as in Figure 3, in the timestamped file develops a versioned history. The environment information for the slave machine is captured for reference, instead of only preserving the container and its version number. As preserving entire machines may be out of scope for a project, the information about the environment and tools is kept. Capturing both tools and environment provides not only information about the process but also allows it to be the recipe for rebuilding the container and its components.

This data can then be used to interrogate the artefacts and entities of the build system. The generated file can be linked to the stated scientific workflow using the Kliko specification to a container that can then be linked to its own build process (Figure 4). The metadata extracted links the container to the maintainer, who may be different from the organisation.
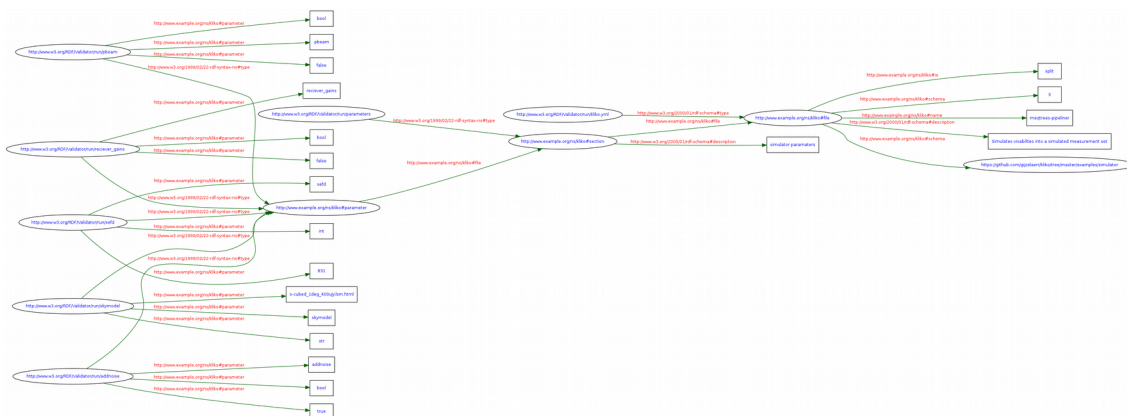
Figure 4. Graph of a Kliko Workflow specification.

Discussion

The workflow specification for the larger project has not been decided and there are various options. The Kliko metadata file, developed in South Africa, allows us to extract

the parameters and the contained processes. It does not account for any logged errors. Querying the workflow presents a combinatorial issue for much larger pipelines, larger than the ones that we have been exposed to through various sub-projects.

Identifying those responsible for software at different levels can be challenging. Moving from the organisational links, the graphs allow us to discover the maintainers from the workflow, allowing their work and software to be cited.

Although we may not achieve the ideal state of the container and metadata being a complete Research Object as we do not describe how the data is being used, we do however describe the parameters given to the contained processes.

The reliable use of time becomes a key object in storing the software so that the relevant metadata can be queried and the flow of information is captured correctly, as maintainers may change over a long running project and we should support historical queries to determine the provenance of the container but to also provide support for citation of the software.

Using this latter graph, we approach the use of provenance to infer integrity of the pipeline and the system. Over the lifetime of a project, processes change and are updated. These graphs may also be used to view the status of software and whether it is following mandated procedures, such as defining the correct operating system or inferring if old libraries are being used.

As automated processes become increasingly linked together to create and update software, it becomes more complicated to maintain the entities and artefacts in the pipeline.

Future Work

Our future work will refine the formal semantics of our presented model and any required extensions, such as the software citations where the maintainer may not be the original developer. Refining the formal semantics will help us to develop more powerful queries and inferences.

We see this work as supporting the integrity of the build environment through querying the files and inferences to show the origins and trustworthiness of the containers. Reliable archiving and provenance supports the reputation of the science project through maintenance of the software's integrity as well as processes for the software that supports the science. Having shown that this work can be implemented as part of the deployment process, we should look at its integration with the data archiving and operations processes.

A Docker-Compose file can be used to compile multiple containers and to build them with a configuration for each part. Although not used in the Integration Prototype, it provides a second model for building complex applications with more complex workflows to be modelled. Although the parameters presented at the moment are relatively simple, combinations of these will provide challenges for discoverability for reproducible research or finding those who should be cited.

Conclusion

We introduce existing work that uses Dockerfiles as part of the reproducible software output. Our work builds on this by linking the versions of the Dockerfile to the relevant

container, extracting the entities and environments: machines, organisations, and people involved, and linking to a workflow file.

This allows the container to be built on another machine and the experiment to be re-run. It supports the data archiving by recording the recipe for recreating the experiment but also how that recipe is built and its parameters. As the Integration Prototype is at early stages, the amount of data that can be used is small so a challenge is scaling this to larger deployments.

Our focus shows how a container can be built within a Continuous Deployment for archiving to preserve both software and creation environments and be linked to workflow metadata. This supports publication and citation efforts.

From these files, inferences may be made upon the artefacts in the automated build process, supporting the process and its artefacts and also supporting any existing software policies. This provokes questions regarding the increasing use of machines as social entities. Building on the existing processes and ideas for preserving the container, we demonstrate a method of preserving the process and also linking this to a workflow file provided by another process.

Using SPARQL queries, we are able to map parameters to the container used but are aware that we are using a small dataset at present. This also supports software metadata archiving and the use of containers as repositories. Providing the metadata provides transitive links between the expression of the container and the container itself, allows us to view the Docker container as software that may also be preserved to support sustainability for the project. This metadata provides a context to the artefact as an object within the research flow, implying that it is more than a software object but an integral part of processes.

Acknowledgements

References

Bechhofer, S., Buchan, I., De Roure, D., Missier, P., Ainsworth J., Bhagat J., Couch, P. et al. (2013). Why linked data is not enough for scientists. *Future Generation Computer Systems 29*(2).

Boettiger, C. (2015). An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review 49*(1).

Di Tommaso, P., Palumbo, E., Chatzou, M., Prieto, P., Heuer, M. L., & Notredame, C. (2015). The impact of Docker containers on the performance of genomic pipelines. *PeerJ* 3 (e1273).

Eglen, S., Marwick, B., Halchenko, Y., Hanke, M., Sufi, S., Gleeson, P., Silver, R.A., Davison, A., Lanyon, L., Abrams, M., Wachtler, T., Willshaw, D. J., Puzat, C., Poline, J-P. (2017). Towards standard practices for sharing computer code and programs in neuroscience. *BioRxiv 045104*. doi:10.1101/045104

Emsley, I. (2016). Docker-Hawser. Github. Retrieved from https://github.com/iaine/docker/commit/355843a6ea98d471268f4e0109d46f08357fabe0

Gent, I.P. (2013). The recomputation manifesto. *arXiv preprint arXiv:1304.3674*.

Haines, R. & Jay, C. (2016). Reproducible research: Citing your execution environment using Docker and a DOI. Software Sustainability Institute Blog. Retrieved from http://www.software.ac.uk/blog/2016-03-29-reproducible-research-citing-your-execution-environment-using-docker-and-doi

Huo, D., Nabrzyski, J. & Vardeman II, C.F. (2015). Smart container: An ontology towards conceptualizing Docker. *Proceedings of the International Semantic Web Conference Posters and Demonstrations Track.* Retrieved from http://ceur-ws.org/Vol-1486/paper_89.pdf

Kuzak, M. (2015). Software sustainability checklist from the Netherlands e-Science Centre. *Software Sustainability Institute Blog.* Retrieved from http://software.ac.uk/blog/2015-11-26-software-sustainability-checklist-netherlands-escience-center

Lebo, T., Sahoo, S., McGuinness, D., Belhajjame, K., Cheney, J., Corsar, D., Garijo, D., Soiland-Reyes, S., Zednik, S., & Zhao, J. (2013). Prov-o: The prov ontology. *W3C Recommendation* 30.

Matthews, B., Shaon, A., Bicarregui, J., & Jones, C. (2010). A framework for software preservation. *International Journal of Digital Curation 5*(1).

Matthews, B., McIlwrath, B., Giaretta, D., & Conway, E. (2008). The significant properties of software: A study. *JISC Report.* Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.456.122&rep=rep1&type=pdf

Molenaar, G. (2014). Docker and radio astronomy: Containing fragile scientific software. In *DockerCon EU 14*. Retrieved from https://www.youtube.com/watch?v=K98cbiQg-A8