Kliko - The Scientific Compute Container Format

Gijs Molenaar[a,c], Spheshile Makhathini[a,c], Julien N. Girard[b,a], Oleg Smirnov[c,a]

[a]SKA SA, Cape Town, South Africa
[b]AIM/CEA-Saclay, Universit Paris Diderot, France
[c]Department of Physics & Electronics, Rhodes University, Grahamstown, South Africa

Abstract

Kliko is a Docker-based container specification for running one or multiple related compute jobs. The key concepts of Kliko are the encapsulation of data processing software into a container and the formalization of the input, output and task parameters. By formalizing the parameters, the software is represented as abstract building blocks with a uniform and consistent interface. The main advantage is enhanced scriptability and empowering pipeline composition.

Formalization is realized by bundling a container with a Kliko file, which describes the IO and task parameters. This Kliko container can then be opened and run by a Kliko runner. The Kliko runner will parse the Kliko definition and gather the values for these parameters, for example by requesting user input or retrieving pre-defined values from disk. Parameters can be various primitive types, for example: float, int or the path to a file.

This paper will also discuss the implementation of a support library named Kliko which can be used to create Kliko containers, parse Kliko definitions, chain Kliko containers in workflows using a workflow manager library such as *Luigi*. The Kliko library can be used inside the container to interact with the Kliko runner.

Finally to illustrate the applicability of the Kliko definition, this paper will discuss two reference implementations based on the Kliko library: RODRIGUES, a web-based Kliko container scheduler and output visualizer specifically for astronomical data, and VerMeerKAT, a multi-container workflow data reduction pipeline which is being used as a prototype pipeline for the commissioning of the MeerKAT radio telescope.

The Kliko library is open source. The documentation and source code can be found on the main website.[1]

Keywords: Docker, containerization, astronomy, scientific computing, pipelines, data reduction

1. Introduction

1.1. Software in science

The use of computer software in research has resulted in significant hardware and software developments in computing science. Nowadays, the number of different scientific software packages is overwhelming, and it has become progressively difficult for users (e.g. a scientist) to evaluate the relevance, usage and the performance of these packages.

Firstly, installing scientific software can be cumbersome, especially when the installation and/or compilation is poorly designed. The software code, the library dependencies, the host platform and the compilers may change over time, making it unclear how the original developer(s) intended to install and use the software. Secondly, conflicting dependencies may arise when different software packages are built together, making it difficult to install them on the same system. Thirdly, software packages have non-uniform interfaces as they have varying expectations of interaction with a user or with other packages on the same system.

Kliko is a Docker-based encapsulating and chaining framework that purports to mitigate these issues by creating a container of the software thereby solving the first and second issue above. The third issue can then be solved by building a Docker container that has minimal extra requirements, i.e. the Kliko definition.

Kliko consists of two parts: i) a set of utilities for creating a container, including parsers to check if all (meta) data is valid; and ii) a support library that can be used to schedule a Kliko container and run it from a command line or from a web interface.

Kliko is not a pipeline construction tool itself, nor a web interface, but it can assist in making these.

1.2. Software containerization with Docker

Containerization is a method for building self-contained environments (called "containers") for applications. These containers can then be distributed and used with minimal effort on a large variety of platforms.

Containerizing applications is not new. Similar techniques have been applied before, e.g. jail for FreeBSD [2],

---

[1]https://github.com/gijzelaerr/kliko

[2]https://www.freebsd.org/doc/handbook/jails.html

zones for Solaris [16] and chroot for GNU/Linux [3]. However, their application was mostly limited to enhancing security and to carry out clean builds of the UNIX system. The addition of operating system (OS) level process isolation, named control groups or cgroups[17]), to the popular Linux kernel (since 3.8, 2008) accelerated the adoption of containerization for the usage of software distribution.

There are multiple software projects leveraging cgroups, for example rkt[4], Docker[2][5], Singularity[10][6] and LXC[7]. Docker [11] is currently the most popular container technology with the largest community of users and the most momentum for future development and support. Kliko aims to be agnostic of the container technology, but since Docker has the biggest user community, we focus on this implementation.

In Docker, an image is built using a initialization script (a "Dockerfile") which contains the recipe to install or build the application. The Dockerfile is a series of commands applied to a basic and clean Docker image, typically a headless Linux distribution. These base images are retrieved from an online database provided by Docker, and stored locally. The Dockerfile, when executed, will create an "image" which is a "inactive" snapshot of the virtualized application. An image becomes a container when instantiated (e.g. the application runs). The difference between active and inactive is important, a container is an image with an unwritten (dirty) state.

An application that is containerized is self-contained and can be seen as a complete OS without kernel. The container could even only contain a statically compiled binary, but in practise it is useful to have the tools and package manager of a Linux distribution available inside the container. Theoretically, to run a Docker container using Docker on a host machine, the only requirement is to have the Docker daemon running on the host. Unfortunately, there are some hardware specific edge cases like CPU register usage optimization and GPU acceleration. These cases will be discussed in section §7.1.

A Docker container "image" is basically a file system snapshot of a minimal OS. The "target" application (i.e. the one to host) and its library dependencies are installed inside this virtual isolated file system. When the application is started, the container file system is exposed to the application as the working environment.

On a kernel level, cgroups and namespaces are used to create a new isolated environment for the application, limiting access to other processes on the host and presenting the isolated environment as if is a separate host to the application. Intuitively, this can be seen as similar technology as CPU level OS virtualization like VirtualBox, but in the case of containers, the kernel is shared by the host and the guest.

A Docker container also gets a private IP address on an internal network range. This makes the container appear as a separate networked machine to the host. By default, access to network ports are restricted and access needs to be granted per port. One can also forward the port to an external interface where it will appear as the service is running on the host itself.

All of the above might appear similar to simple virtualization, but containerization has some clear additional advantages. Firstly, when using Docker, available physical resources do not need to be partitioned between the host and the guest. While memory size allocated for a virtual machine is fixed or not easy to change, running containers does not require the user to fix this memory size, although it still is possible to limit the amount of memory allocatable by the process. Secondly, there is no CPU instruction emulation, as the process is directly executed on the host kernel. Thirdly, there is minimal startup and shutdown overhead for starting containers as the containerized OS is reduced to minimal consumption. Startup time is instantaneous (in the millisecond range) and loading time will only become noticeable when high numbers of containers are spawned.

In addition to containerization, Docker also offers other features: it uses an "union" file system to join multiple layers of file systems together. The intermediate result of each command in the Dockerfile is cached and stored in layers. These layers can be reused by other containers, allowing data sharing between them, which reduces the size of the storage requirements. These layers can also be stored in a central location, where they can be distributed and reused in both a public or private way.

2. The Kliko specification

The Kliko specification is designed to extend containerization with an uniform interface resulting in simplified interaction with the containerized application.

The Kliko specification describes how a Kliko container should look like and what a Kliko container should expect during runtime. The relevant terminology is listed below:

Def 1: The Kliko Image
A Docker image complying to the Kliko specification. An image is a read-only ordered collection of root file system changes and the corresponding execution parameters for use within a container runtime.

Def 2: The Kliko Container
A container in an active (or inactive if exited) stateful instantiation of a Kliko image.

Def 3: The Kliko Runner
A process that can run a Kliko image to make a container. For example the Kliko-run command line tool, or RODRIGUES (see §5.2).

Def 4: The Kliko Parameters
A list of parameters that can influence the behavior of the software in the container. The list can be arbitrary in size

---

[3] http://man7.org/linux/man-pages/man2/chroot.2.html
[4] https://github.com/coreos/rkt
[5] https://www.docker.com
[6] http://singularity.lbl.gov
[7] https://linuxcontainers.org

and consists of any combination of primitive types listed in Table 2.

2.1. The Kliko Image

A Kliko image should contain a `/kliko.yml` file in YAML[8] syntax following the Kliko schema 2.4. YAML is a human-readable data serialization language and stands for *YAML Ain't Markup Language*. The Kliko image should also contain a `/kliko` file which is called during runtime by the Kliko runner. This Kliko script can be anything executable, but in most cases, it will be a Python script using the Kliko library to check and parse all related Kliko tasks during runtime. Note that we have deliberately chosen not to use the ENTRYPOINT or CMD statements supported by Docker. This way, Kliko is non-intrusive and can be easily added to existing containers that already set an ENTRYPOINT or CMD.

2.2. Expected runtime behavior

During runtime, the Kliko runner will gather the parameters and expose them to the Kliko container. The content of the variables is exposed by the Kliko runner in the `/parameter.json` file, which should contain a flat dictionary in JSON syntax[9]. JSON and YAML are structurally very similar, but YAML is designed to be more human-readable, hence our choice of YAML for the Kliko definition. Future versions of Kliko will support both formats.

While reading this text, one might get confused by the context of the file location (inside or outside the container). As a rule of thumb if a path in this text starts with a slash (`/`) it is *inside* the container.

If one or more of the parameters is a file, those will be exposed by the Kliko runner in the read-only `/param_files` folder during runtime. It is the responsibility of the Kliko container to parse the `/parameters.json` file, perform potential the run-time housekeeping and convert the parameter keys, values and/or files into an eventual command do be executed.

It is recommended to write logging to stdout and stderr. This makes it easier for the Kliko runner to visualize or parse the output of a Kliko image.

2.3. Flavors of Kliko Images

We distinguish two flavors of Kliko containers, *joined Input/Output* (read-write) and *split IO* (read-only). The style of container is specified in the `io` field in `/kliko.yml` file inside the container, see §2.4.

The difference is the way the contained software interacts with the working data. In the case of *split IO* the Kliko runner exposes the input data to the container in the `/input` folder. This folder is read-only, to prevent accidental manipulation of the data. The Kliko container is expected to write any output data into the `/output` folder. The Kliko Runner will then handle this output data after the container reaches the end if its lifetime. A *split IO* Kliko container should always yield the same results for multiple independent runs when presented with the same data and parameters (formally is called, "having no side effects"). This is basically the essence of the functional programming paradigm.

In the case of *joint IO* there is only one point of interaction with the Kliko host, `/work` which is exposed read/write. Basically, the input and output folders are combined into one that is mounted with read/write permissions. Contrary to the *split IO* flavor, this might be potentially dangerous for data processing as it can alter the original data.

From a run-time parallelization perspective, the *split IO* flavor is preferred. A container without side effects enables the Kliko Runner to do graph-based logical inference of dependencies and execution scheduling, reuse results and also run various containers in parallel, potentially resulting in faster execution. In practice, existing software does not always support this type of operation, or it is simply not feasible to create a copy of the data. In that case, the *joined IO* style has to be used.

2.4. The `/kliko.yml` schema

A *kliko.yml* file is a YAML file and it *should* contain the fields listed in Table 1.

Each section contains a list of fields. Each fields statement should contain a list of field elements. Each field element has two mandatory keys, a name and a type. Name is a short reference to the field which needs to be unique. This will be the name for internal reference. The type defines the type of the field, possible types are listed in Table 2. Depending on the type there are optional extra fields, listed in Table 3.

The schema described above is defined in the Kwalify format. Kwalify is a parser and schema validator for YAML and JSON[10]. The definition itself is also written in YAML. The Kliko library pykwalify[11] is used to validate the YAML file against a schema. The full Kliko version 2 schema is listed Listing 10 in Appendix B.

2.5. The `/parameters.json` file

When a container is started, the Kliko runner will mount a `/parameters.json` file into the container. This file contains all parameters for the container in the JSON format. The `/kliko` script supplied by the container author should read and parse the `/parameters.json file`. The Kliko library (3.2) supports helper functions and scripts to parse and validate this file. Validation is done based on the `/kliko.yml` definition, which is useful for preventing or tracking down problems.

---

[8] http://www.yaml.org
[9] https://www.json.org
[10] http://www.kuwata-lab.com/kwalify
[11] https://github.com/Grokzen/pykwalify

Table 1: Required Kliko fields

field	description
schema_version	The version of the Kliko specification, independent of the versioning of the Kliko library
name	Name of the Kliko image. For example "radioastro/simulator" for RODRIGUES.
description	A more detailed description of the image.
url	Website of project or repository where project is maintained
io	"join" or "split". See the two flavors of Kliko Containers in §2.3
Sections	a list of one or more sections, grouping fields together.

Table 2: Kliko variable types

type	description
choice	field with a predefined set of options, see the optional choices field below
str	string value
float	float value
file	A file path. This file will be exposed in `/param_files` at runtime by the Kliko Runner
bool	A boolean value
int	An integer value

An example parameters file that could be generated based on the kliko.yml definition is shown in Listing 1.

```
{
    "int": 10,
    "file": "some-file",
    "char": "gijs",
    "float": 0.0,
    "choice": "first"
}
```

Listing 1: Example `parameters.json` file

Note that the sections are not supplied since they are only used for grouping and representation to the user.

3. Running Kliko containers

3.1. Running a container manually

As an example, we will describe an extremely simple Kliko container named "fitsimagerescaler", available on our github repository described in §6. This container takes a FITS image file which resides in the `/input` directory, opens it, multiplies the pixel values by a parameterized value (2 by default) and exports the result as a new FITS image in `/output`. The actual code that is run in `/kliko` is shown in Listing 5.

Starting the Kliko container is nothing more than starting the container using Docker with some specific flags. If the `parameters.json` file already exists, starting the container from the command line looks like this:

'pwd'/input and 'pwd'/input are input and output folders in your current working directory outside the container. 'pwd' is required since the Docker engine can only work with absolute paths.

```
$ docker run -t -i \
   -v `pwd`/parameters.json:/parameters.json:ro \
   -v `pwd`/input:/input:ro \
   -v `pwd`/output:/output:rw \
   kliko/fitsimagerescaler /kliko
```

Listing 2: Command for running a Kliko container manually. The `/parameters.json` file is mounted as well as the input and output directories, in the "split" mode. The `kliko.yml` is already inside the container.

This command fires up the `fitsimagerescaler` container, mounts the `parameters.json` file as well as input/output directories and runs the `/kliko` script located in the root directory. In our case, the FITS image file has to be present in the local input directory for the script to run properly.

For the reader unfamiliar with Docker this command might look cumbersome and error-prone but the command constitutes the fundamental principle of Kliko (in addition to specification and the extensive test suite). This can be used a base to create your own Kliko runner in any language that has Docker bindings.

This way of implementing inputs, outputs and running generic scripts, demonstrates that it becomes fairly easy to connect the input parameters and data (generated by scripting and/or a web form) to software living inside the container. Kliko implements a set of tools to insure the robustness of this implementation.

3.2. Inside the Kliko container

The `/kliko` script is the first entry point into the specifics of the container. We can easily parse the `/parameters.json` file using a JSON parser in python, by performing the commands in code listing 3.

Table 3: Kliko field types

field	description
initial	supply a initial (default) value for a field
max_length	define a maximum length in case of string type
choices	define a list of choices in case of a choice field. The choices should be a mapping
label	The label used for representing the field to the end user. If no label is given the name of the field is used
required	Indicates if the field is required or optional
help_text	An optional help text that is presented to the end user next to the field

```python
import json
parameters = json.load(open('/parameters.json', 'r'))
```

Listing 3: Example of how to parse `parameters.json` file with standard python packages.

However, at this point, the parameters file is not yet validated. We can be sure that the parameters file is actually generated from our Kliko definition by installing the Kliko library inside the Kliko container and using it from our `/kliko` script4. Validation helps reduce human or programming error.

```python
from kliko.validate import validate
parameters = validate()
```

Listing 4: Example how to parse `parameters.json` using the Kliko library

After the Kliko validation is performed, a dictionary is created and all values can be used freely inside the script itself (by passing them to functions) or passed directly to the container OS as environment variables. All this validation is intended to reduce human or programming error as early as possible.

```python
import kliko
from kliko.validate import validate
from astropy.io import fits

parameters = validate()
file = parameters['file'] # filename in /input
factor = parameters['factor'] # multiplying factor

print('welcome to fits multiply!')
print("'%s' multiplied by '%s':" % (file, factor))

data = fits.getdata(file)
multiplied = data * factor
output = path.join(kliko.output_path, path.basename(file))
fits.writeto(output, multiplied, clobber=True)
```

Listing 5: Example of `/kliko` file scale the values in a FITS image

3.3. kliko-run

Instead of calling Docker directly, Kliko is bundled with `kliko-run`, a command line utility that enables a user to

run a Kliko container in a seamless way. It also assists in exploring the parameters that a given Kliko container supports. Code listing 6 shows the docstring of the kliko-run command for a simple container (available as a test container shipped with Kliko). The optional arguments are generated automatically from the YAML file. It shows how any shipped application can be easily interfaced with the host system, in a way that part (or all) of the variable names of the application can be modified directly from the command line. This enables Kliko, with the help of Docker, to ship a complex software as an application that is equipped with a simple interface. Kliko provides a simple way to implement this interface in a controlled and robust way while being completely agnostic about the mechanics happening inside the container.

```
$ kliko-run kliko/fitsimagerescaler --help

  usage: kliko-run [-h] [--target_folder TARGET_FOLDER] --choice
↪ {second,first}
                  --char CHAR [--float FLOAT] --file FILE --int
↪ INT
                  image_name

  positional arguments:
    image_name

  optional arguments:
    -h, --help           show this help message and exit
    --target_folder TARGET_FOLDER
                         specify output or work folder
↪ (default: ./output)
    --choice {second,first}
                         choice field (default: second)
    --char CHAR          char field, maximum of 10 chars
↪ (default: empty)
    --float FLOAT        float field (default: 0.0)
    --file FILE          file field, this file will be put in
↪ /input in case
                         of split io, /work in case of join io
    --int INT            int field
```

Listing 6: Output of the `kliko-run` command

4. Chaining containers

Kliko containers can also be chained. Chaining means that the output of a container is connected to the input of a consequetively executed container. This enables the creation of workflows. Additionally, if the Kliko containers are "split IO", we can execute containers that do not

depend on each other in parallel. Their intermediate results can be cached, which can speed up execution time of future workflow runs and can help debugging problems with the workflow by examining intermediate results.

There are various workflow creation frameworks and libraries available. We evaluated two popular Python-based workflow management libraries, airflow[12] and Luigi[13]. Although Kliko is designed to be workflow management independent, Luigi is a better fit. Airflow is intended to visualize automated repetitive tasks like cron jobs, while Luigi is more oriented towards once-off batch processing. Luigi is an open source Python library that handles dependency resolution, does workflow management, optionally visualizes data in a web interface and can handle and retry failures. At the core of a Luigi workflow is the Task, which is a Python class that defines what to be executed, how to check if this task has already been executed and optionally if it depends on the result of another task. This is a very simple but powerful concept that integrates fluently with Kliko. The Kliko library contains a KlikoTask definition which can be used to integrate Kliko in a Luigi pipeline.

5. Example usage of Kliko
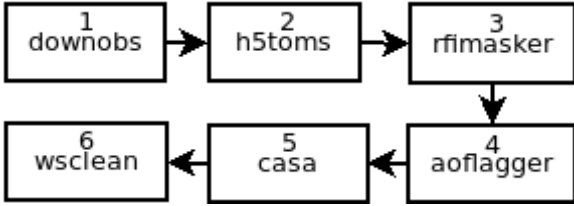
5.1. VerMeerKAT

Figure 1: Flow diagram of the VerMeerKAT data reduction pipeline.

To illustrate the mechanics of chaining container together we explain a real world application here, the VerMeerKAT pipeline.

VerMeerKAT is a semi-automated data reduction pipeline for the first phase of deployment of the MeerKAT telescope[14][3][8]. All steps in this pipeline are shown in Figure 1. It is a closed source project used internally at SKA South Africa and is based on a set of bash scripts. Using bash for this is not ideal; it is hard to make a portable pipeline, not trivial to recover and continue from errors, reuse intermediate results. Parallelization is possible, but this needs to be manually and explicitly defined in the scripts.

For this paper we made a Kliko version of this pipeline[15]. Using Kliko for composing this pipelines has some key advantages; i) easy installation and deployment of the software ii) optional caching of intermediate data products;

ii) implicit parallelization of tasks independent steps; and iii) progress visualization and reporting using a tool like Luigi.

The VerMeerKAT pipeline, (see Figure 1), starts by querying the MeerKAT data archive for a given set of observations (step 1), along with the meta-data for that observation. Next step is to convert the downloaded data from the hdf5 format to a MeasurementSet (MS)[9] (step 2), since most radio astronomy tools only support this format. Once the data is in the MS format, it is then taken through a series of manual and automated tools that excise data points that are contaminated by radio frequency interference [15, 13] (step 3, 4 and 5). The data are then calibrated [7, 18] and imaged [4] (step 6).

For the creation of the VerMeerKAT Kliko containers, we make use of the packages from KERN[12]. KERN is a bi-annually released set of radio astronomical software packages. This suite contains most of the tools that a radio astronomer needs to process radio telescope data. These packages are precompiled binaries in the Debian format and contain all the metadata required for installing the package like dependencies and conflicts. KERN is only supported on Ubuntu 16.04 at the moment of writing, but that is no problem inside a Docker container.

Listing 7 is an example Dockerfile for the wsclean[14] Kliko container in VerMeerKAT. When this file is built it will install the KERN package of wsclean inside the container and bundle the container with the Kliko definition and parser script. The Docker files for the other steps are very similar.

```
FROM kernsuite/base:1
RUN docker-apt-install wsclean
ADD kliko.yml /
ADD kliko /
```

Listing 7: Dockerfile for a KERN package

Listing 8 is an example Kliko task definition. This example will use the rfimasker Kliko containers. It depends on the H5tomsTask Kliko task. When this task is invoked using Luigi, Luigi will do the dependency resolution, check if the required tasks have run and if not, run them. The progress can be visualized with the Luigi interface (Figure 2). All other steps in the workflow are very similar to this example.

```
from kliko.luigi_util import KlikoTask

class RfiMaskerTask(KlikoTask):
    @classmethod
    def image_name(cls):
        return "vermeerkat/rfimasker:0.1"

    def requires(self):
        return H5tomsTask()
```

Listing 8: An example KlikoTask
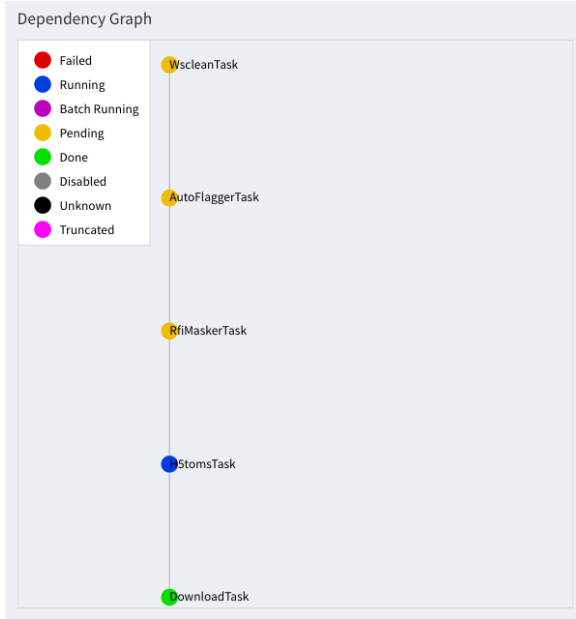
[12] https://airflow.apache.org
[13] https://github.com/spotify/luigi
[14] http://www.ska.ac.za/gallery/meerkat/
[15] https://github.com/gijzelaerr/vermeerkat-kliko

Figure 2: Screenshot of Luigi, running the vermeerkat pipeline

Figure 3: Screenshot of RODRIGUES, the result visualizer.

5.2. RODRIGUES

Another project using Kliko is RODRIGUES (RATT Online Deconvolved Radio Image Generation Using Esoteric Software)[16]. RODRIGUES is a web-based Kliko job scheduling tool and it uses the Kliko as a required format for the job. RODRIGUES acts as a "kliko runner". A user of RODRIGUES can log into RODRIGUES and add a new Kliko container. RODRIGUES will open the Kliko container, parse the parameters and expose these parameters to the user using a web form (Figure 4). The user can then fill in the parameters in this form and submit the job into the RODRIGUES container queue. The container will be run on the system configured by the RODRIGUES system administrator. Once the job is finished the results are presented to the user in the same web interface (Figure 3).

RODRIGUES makes it much easier to schedule new jobs with varying parameters, enabling scientists with minimal programming or computing knowledge to run experiments in a clean, visual, structured and reproducible way.

6. Software availability

Kliko and its library are open source, licensed under the GNU Public License 2.0[17]. Kliko is bundled with an extensive test suite which covers 80% of the source code as of the current release `0.6.1`. The Kliko library is written in Python and is compatible with Python 2.7, all Python 3 versions and even PyPy. Development and distribution is done on Github and a third party continuous integration service runs the full test suite on all supported platforms for every commit and every Github pull request.
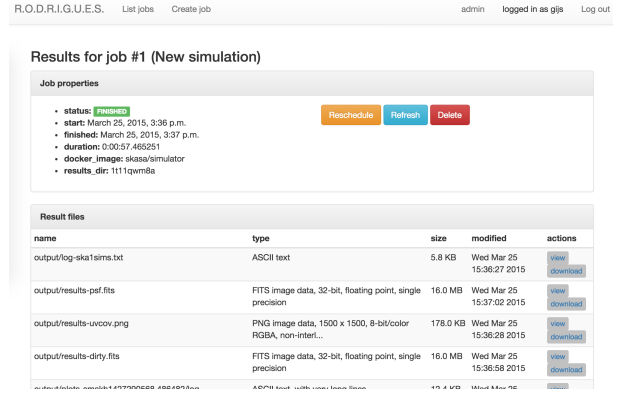

Figure 4: Screenshot of RODRIGUES, parameter form is generated from Kliko definition.

---

[16] https://github.com/ska-sa/rodrigues
[17] https://github.com/gijzelaerr/kliko

7

7. Discussions and Prospects

7.1. Limitations

While developing Kliko, we ran into various issues with Docker which might limit the applicability. It is up to the user to decide if this affects the usefulness of Docker and or Kliko. These issues are listed hereafter for your consideration. That said, the field of containerization is evolving very fast and hopefully most of these issues will be resolved soon or can be worked around.

First of all, being able to run a Docker container on a system is very similar to giving the user administrative access to the machine, that is the user can escalate easily to root privileges [18] [5]. The singularity containerization technology has a more secure design and we are planning to add support for this framework in future versions of Kliko.

Additionally, using GPU acceleration with NVidia hardware is not trivial, since the kernel driver version and library version need to match up, breaking the independence between host and container. There is a workaround available, but this requires a replacement of the Docker daemon with a custom one [19].

A similar issue arises with optimization flags. For example, SIMD instructions can greatly enhance the runtime speed, but not all x86 processors support all SIMD optimization. This will result in crashes of the binary if it is compiled with optimization not supported by the host. Again this breaks the platform independence assumption. A good strategy is to be conservative and compile your binaries for the oldest architecture you intend to support. The good news is that it is easier to support multiple target platforms in the same binary when using modern versions of GCC [20].

Another issue is that it is easy to inherit Docker definitions from other Docker definitions, but it is currently not possible to combine Docker definitions together or to inherit from multiple Docker definitions at the same time (merge). Following the Docker philosophy, Docker containers should have a single responsibility, so this should not be a problem, it does not require mixing of Docker definitions. However, in practice, this is not always possible: sometimes various big libraries need to communicate in the same memory space so a new Docker container with all software needs to be created. The results are far away from minimal small single purpose Docker containers.

For network intensive applications Docker may be less well suited, since the use of network address translation[6]. Also here there is a workaround available by disabling the translation and using the host network stack directly.

7.2. Future Work

During the development and usage of Kliko we became aware of CommonWL[1], a more generic approach to describe applications input/output flow and parameters. We will investigate how we can incorporate CommonWL into Kliko to extend the usability and user base.

At the moment Kliko is designed with other container solutions in mind. Singularity is an alternative that looks to be gaining momentum within the scientific compute (HPC) community since it is more aware and careful of the security implications that come by allowing running containers on a shared infrastructure.

7.2.1. Streaming Kliko

Kliko was born in the field of radio astronomy. Most tools in this field operate on data living on disk. Radio astronomy uses several file formats, for example, casacore Measurement Sets, FITS images and, in some cases HDF5. Using the file system is an easy to understand and technically stable approach, but problems arise when the size of the dataset grows. Emerging telescope arrays such as MeerKAT, followed by SKA phase 1, will result in an exponential growth in data rates. Repeatedly reading and writing data from to slowest medium in a computer – the disk – is not going to scale and a new strategy is needed. Directly streaming data between processing tasks will be required. Although this is already being done [21] in some pipelines, there is no field-wide accepted standard that fits all needs. Our plan of action is to investigate existing solutions being used, investigate industry standards [22], optionally create a sub-specification and open source reference implementation with support libraries for the most used public languages[23].

8. Conclusions

Kliko is a Docker-based container specification. It is used to create abstract descriptions of the input and output of existing software resulting in Kliko containers. These Kliko containers can be used to encapsulate a single job or can be chained together in a pipeline. In the future, we probably will adopt the CWL standard to extend the interoperability with other existing workflow tools. Kliko is written in Python, open source and available free to use.

9. Acknowledgments

---

[18]https://github.com/docker/docker/issues/6324
[19]https://github.com/NVIDIA/nvidia-docker
[20]https://lwn.net/Articles/691932

---

[21]https://github.com/ska-sa/spead2
[22]https://github.com/google/protobuf
[23]http://arxiv.org/pdf/1507.03989.pdf

10. References

References

[1] Peter Amstutz, Michael R. Crusoe, Neboja Tijani, Brad Chapman, John Chilton, Michael Heuer, Andrey Kartashov, Dan Leehr, Herv Mnager, Maya Nedeljkovich, Matt Scales, Stian Soiland-Reyes, and Luka Stojanovic. Common Workflow Language, v1.0. 7 2016.

[2] Carl Boettiger. An introduction to docker for reproducible research, with examples from the R environment. *CoRR*, abs/1410.0846, 2014.

[3] RS Booth, WJG De Blok, JL Jonas, and B Fanaroff. Meerkat key project science, specifications, and proposals. *arXiv preprint arXiv:0910.2935*, 2009.

[4] D. S. Briggs, F. R. Schwab, and R. A. Sramek. Imaging. In G. B. Taylor, C. L. Carilli, and R. A. Perley, editors, *Synthesis Imaging in Radio Astronomy II*, volume 180 of *Astronomical Society of the Pacific Conference Series*, page 127, 1999.

[5] Thanh Bui. Analysis of docker security, January 2015.

[6] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172. IEEE, March 2015.

[7] J. P. Hamaker. Understanding radio polarimetry. *A&A*, 456(1):395–404, 2006.

[8] Justin L Jonas. Meerkatthe south african array with composite dishes and wide-band single pixel feeds. *Proceedings of the IEEE*, 97(8):1522–1530, 2009.

[9] AJ Kemball and MH Wieringa. Measurementset definition version 2.0. *URL: http://casa. nrao. edu/Memos/229. html*, 2000.

[10] Gregory M. Kurtzer. Singularity 2.1.2 - Linux application and environment containers for science, August 2016.

[11] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[12] Gijs Molenaar. Kern - a bi-annually released set of radio astronomical software packages (in preperation), 2017.

[13] A. R. Offringa, J. J. van de Gronde, and J. B. T. M. Roerdink. A morphological algorithm for improved radio-frequency interference detection. *A&A*, 539, March 2012.

[14] AR Offringa, Benjamin McKinley, Natasha Hurley-Walker, FH Briggs, RB Wayth, DL Kaplan, ME Bell, Lu Feng, AR Neben, JD Hughes, et al. Wsclean: an implementation of a fast, generic wide-field imager for radio astronomy. *Monthly Notices of the Royal Astronomical Society*, 444(1):606–619, 2014.

[15] Jayanti Prasad and Jayaram Chengalur. Flagcal: a flagging and calibration package for radio interferometric data. *Experimental Astronomy*, 33(1):157–171, 2012.

[16] Daniel Price and Andrew Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *LISA '04: Eighteenth Systems Administration Conference*, pages 241–254, 2004.

[17] Rami Rosen. Resource management: Linux kernel namespaces and cgroups. *Haifux, May*, 186, 2013.

[18] Smirnov, O. M. Revisiting the radio interferometer measurement equation. ii. calibration and direction-dependent effects. *A&A*, 527:A107, 2011.

Appendix A. The Kliko validation specification

```
schema;fields:
 type: map
 mapping:
   name:
     type: str
     required: True
   type:
     type: str
     required: True
     enum: >
       ['choice', 'char', 'float', 'file', 'bool', 'int']
   initial:
     type: any
     required: False
   max_length:
     type: int
     required: False
   choices:
     type: map
     required: False
     mapping:
       regex;(.*):
         type: str
   label:
     type: str
     required: False
   help_text:
     type: str
     required: False
   required:
     type: bool
     required: False
type: map
mapping:
 schema_version:
   type: int
 author:
   type: str
 name:
   type: str
 description:
   type: str
 container:
   type: str
   pattern: .+/.+
 email:
   type: str
   pattern: .+@.+
 url:
   type: str
   pattern: >
     https?:\/\/(www\.)?[-a-zA-Z0-9@:
     %._\+~#=]{2,256}\.[a-z]{2,6}\b([-a-zA-Z0-9@:%_\+.~#?&//=]*)
 io:
   type: str
   required: True
   enum: ['split', 'join']
 sections:
   type: seq
   matching: "any"
   sequence:
     - type: map
       mapping:
         name:
           type: str
           required: True
         description:
           type: str
           required: True
         fields:
           type: seq
           required: True
           sequence:
             - include: fields
```

Listing 9: The Kliko definition version 2

Appendix B. An example kliko.yml file

```
schema_version: 2
name: Kliko test image
description: for testing purposes only
container: kliko/klikotest
author: Gijs Molenaar
email: gijsmolenaar@gmail.com
url: http://github.com/gijzelaerr/kliko
io: split

sections:
 -
   name: section1
   description: The first section
   fields:
      -
        name: choice
        label: choice field
        type: choice
        initial: second
        required: True
        choices:
          first: option 1
          second: option 2
      -
        name: char
        label: char field
        help_text: maximum of 10 chars
        type: char
        max_length: 10
        initial: empty
        required: True
      -
        name: float
        label: float field
        type: float
        initial: 0.0
        required: False
 -
   name: section2
   description: The final section
   fields:
      -
        name: file
        label: file field
        help_text: a helpful text
        type: file
        required: True
      -
        name: int
        label: int field
        type: int
        required: True
```

Listing 10: Example `/kliko.yml`