Astronomy and Computing 52 (2025) 100959

Contents lists available at ScienceDirect

Astronomy and Computing

journal homepage: www.elsevier.com/locate/ascom

Full length article

Africanus IV. The Stimela2 framework: Scalable and repeatable workflows, from local to cloud compute

O.M. Smirnov [a,b,c,*], S. Makhathini [d,a,e], J.S. Kenyon [a], H.L. Bester [b,a], S.J. Perkins [b], A.J.T. Ramaila [b,a], B.V. Hugo [b,a]

[a] Centre for Radio Astronomy Techniques & Technologies (RATT), Department of Physics and Electronics, Rhodes University, Makhanda, EC, South Africa
[b] South African Radio Astronomy Observatory (SARAO), Cape Town, WC, South Africa
[c] Institute for Radioastronomy, National Institute of Astrophysics (INAF IRA), Bologna, Italy
[d] School of Physics, University of the Witwatersrand, Johannesburg, Gauteng, South Africa
[e] Observatory of Cagliari, National Institute of Astrophysics (INAF OAC), Cagliari, Italy

ARTICLE INFO

Keywords:
Standards – techniques
Interferometric – Computer systems organization
Pipeline computing – Software and its engineering
Data flow architectures – Software and its engineering
Cloud computing – Software and its engineering
Interoperability

ABSTRACT

Stimela2 is a new-generation framework for developing data reduction workflows. It is designed for radio astronomy data but can be adapted for other data processing applications. Stimela2 aims at the middle ground between ease of development, human readability, and enabling robust, scalable and repeatable workflows. Stimela2 defines a YAML-based domain specific language (DSL), which represents workflows by linear, concise and intuitive YAML-format *recipes*. Atomic data reduction tasks (binary executables, Python functions and code, and CASA tasks) are described by YAML-format *cab definitions* detailing each task's *schema* (inputs and outputs). The Stimela2 DSL provides a rich syntax for chaining tasks together, and encourages a high degree of modularity: recipes may be nested into other recipes, and configuration is cleanly separated from recipe logic. Tasks can be executed natively or in isolated environments using containerization technologies such as Apptainer. The container images are open-source and maintained through a companion package called cult-cargo. This enables the development of system-agnostic and repeatable workflows. Stimela2 facilitates the deployment of scalable, distributed workflows by interfacing with the Slurm scheduler and the Kubernetes API. The latter allows workflows to be readily deployed in the cloud. Previous papers in this series used Stimela2 as the underlying technology to run workflows on the AWS cloud.

This paper presents an overview of Stimela2's design, architecture and use in the radio astronomy context.

1. Introduction

Radio astronomy data reduction has been anecdotally described as "death by a million papercuts". The past decade has exacerbated this, with several new radio facilities coming online, each with its own instrumental quirks and data challenges, and with rapid progress in algorithmic and software developments. Radio astronomy packages are becoming increasingly arcane – for example, packages such as WSClean (Offringa et al., 2014) and DDFacet (Tasse et al., 2018) have many dozens of parameters – with relatively few "black belt" experts possessing the expertise to exploit them optimally. On the other hand, building "black box" data reduction pipelines aimed at the relatively unqualified user has, historically, proven challenging (see discussion of this problem by Molenaar, 2021). There are some success stories: the ALMA and VLA pipelines provided by CASA serve the needs of most users for standard observations. The word "standard", however, is an important qualifier: the sensitivity, high spatial resolution and field-of-view of new instruments leads to new and more subtle calibration and imaging problems that require non-standard, often novel, software tools. Now, while such tools (e.g., WSClean, DDFacet, pfb-imaging, QuartiCal) are available and continue being developed, incoperating them into pipelines requires expert knowledge of both the tools and the pipelines, making it prohibitively difficult for most astronomers. The OxKAT pipeline (Heywood, 2020) integrates a number of these novel tools into a single workflow, and ships as a single Docker image with all software pre-installed. The CARACal pipeline (Józsa et al., 2022) attempts to address this by using the first-generation Stimela[1]

* Corresponding author at: Centre for Radio Astronomy Techniques & Technologies (RATT), Department of Physics and Electronics, Rhodes University, Makhanda, EC, South Africa.
E-mail address: o.smirnov@ru.ac.za (O.M. Smirnov).

[1] *Stimela* is the isiZulu word for a [railroad] train.

https://doi.org/10.1016/j.ascom.2025.100959
Received 13 December 2024; Accepted 11 March 2025
Available online 26 March 2025

package (Makhathini, 2018) as the backend, and providing a YAML-format user configuration. CARACal is somewhat more transparent and includes a rich set of legacy and novel tools, but the process of adding new tools is verbose and cumbersome due to limitations of the Stimela backend (discussed below). While both of these pipelines have done a great deal to empower data reduction by non-expert MeerKAT users, they do not readily lend themselves to casual user modification. Under such conditions, it remains the case that most high-fidelity data reductions are the result of handcrafted data reduction by black belts (a process also referred to in the community as "hero mode" data reduction). This also contributes to a reproducibility crisis – not unique to radio astronomy – in that while the raw observational data for any particular science result may be publicly available, the data reduction leading to it may not be readily repeatable, let alone reproducible. Minor changes in parameter settings and package versions can often lead to appreciable changes in the output images.

This creates an important problem: how do we simultaneously manage software complexity, make diverse (and rapidly evolving) software packages work together, create fully repeatable data reduction recipes,[2] while allowing black belt users, who in the normal course of their job develop new recipes, to share these with the community in an accessible way? Finally, an issue that ends up touching upon all of these questions, is, as we will see later in this work, how (and if) can we use cloud computing resources to run such recipes?

Cloud computing in astronomy. Cloud computing has become a mainstream technology, and it is a common choice for businesses to deploy their data processing on commercial clouds, such as those provided by Amazon (AWS), Google (GCE) and Microsoft (Azure). Simple economy-of-scale arguments suggest that big commercial providers must be able to supply computing resources at lower unit cost than in-house compute, for all but the biggest organizations — the caveat is, does a given compute workflow and software stack map onto the cloud compute model efficiently? In astronomy, the flagship example is provided by the Rubin Observatory, which is deploying its entire Science Data Platform[3] on the Google cloud (O'Mullane et al., 2023); see also Berriman (2023) for a discussion. At 100+ Petabytes in scale, the Rubin SDP project is in the same class as the SKA and its precursors. Despite a concerted effort by AWS and the SKA Observatory (dating back to 2015) to promote cloud development in radio astronomy,[4] our community has been relatively slow to exploit this technology, although a few project-specific demonstrations and deployments have been reported (Toomey et al., 2017; Sabater et al., 2017; Byrne and Jacobs, 2021; Dodson et al., 2016, 2022).

This reluctance could be due to both sociological reasons and technological bottlenecks. Of the latter, we can identify at least two that are relevant to this paper series. Firstly, our most established data formats, in particular the Measurement Set (MSv2) and its underlying Casacore Table Data System (CTDS: van Diepen, 2015) require a POSIX filesystem, which, in a cloud environment, necessitates the use of the more expensive *block store* (as opposed to cheaper *object store* options such as S3). The standard MSv2 storage manager (SM) format, as well as the CTDS API, is also ill-suited for parallel I/O, as discussed earlier in this paper series by Perkins et al. (2025). The latter can be somewhat alleviated via alternative SMs: for example Wang et al. (2020a) demonstrated massively parallel I/O using an ADIOS2-based SM. Regretfully, this solution has not been adopted by observatories at large, with most legacy MSv2 data still stored using the built-in SMs. CTDS storage costs are also exacerbated by legacy tools that commonly at least double or triple the on-disk data volumes (viz., CASA's practice of creating model and corrected data columns.) Secondly, our data reduction workflows, besides being long and complex, tend to be "thick-thin", in the sense that some steps may be highly parallelizable (e.g. gridding), while others are strictly serial (e.g. classic CLEAN minor cycle); some have a large RAM footprint, while others are very economical. This makes it particularly challenging to utilize cloud resources in a cost-efficient way, since one is charged an hourly rate for a given virtual machine instance, regardless of actual CPU or RAM utilization. This paper series discusses our solution for both of these bottlenecks.

Containerization. The original Stimela package aimed to address repeatability and ease-of-installation concerns by implementing a *containerized* workflow framework. Stimela provides a set of curated container images for most of the popular radio astronomy packages (these images being regularly updated with each release of Stimela). Each image is complemented by a JSON-format *schema* formally describing the inputs and outputs of the corresponding package. Together, the image and the schema is known as a *cab* definition. These cabs can be chained together via a Python API to produce *recipes*, the latter being a sequence of steps, with each step invoking a particular cab with a set of parameters. Stimela takes care of parameter validation (attempting to catch as many errors up front as possible), and then executes the recipe, by instantiating the images as Docker (or Apptainer/Singularity, or Podman) containers and passing them the appropriate parameters. Stimela addressed some of the above concerns: wrapping the packages into container images takes away most of the software installation complexity, and allows the recipe to run virtually anywhere (only Python, and Docker or Podman or Apptainer is needed on the host system) in a repeatable manner. Stimela has had success as the underlying engine for the VermeerKAT[5] and CARACal pipelines. We have also observed a lot of casual use in the community, although primarily to run individual cabs in one-off mode (thus using Stimela as, essentially, a software distribution tool), but there has been no meaningful recipe development outside the core team. We have identified three major shortcomings in the original design: (i) recipe execution is strictly serial, which prevents cabs without interdependence from running in parallel. Any more complex flow control (or parallelization) is deferred to the calling Python code, however (ii) the Python-based API has proven ill-suited as a standardized recipe-building framework. In particular, long and complicated workflows tend to devolve to, essentially, regular Python scripts, and become hard to read — which goes against the core aim of simplifying recipe-building. Finally, (iii) while invoking native binary applications from within a recipe can be done via standard Python modules, this effectively bypasses Stimela entirely. This limits rapid development and experimentation in a research environment.

Stimela2 attempts to address these shortcomings through a number of new features:

- A concise, rich, and human-friendly YAML-based recipe syntax, replacing the Python API;
- High degree of portability: libraries of recipes and custom cab definitions can be distributed as YAML files;
- A rich formula and string substitution syntax for specifying parameter values;
- Support for scatter–gather constructs within a recipe;
- Cabs are no longer restricted to container images and Python functions: native binaries and virtual environments can be wrapped into cabs, and intermixed within a single workflow;
- Support for distributing workflows over slurm[6] and Kubernetes[7] clusters;
- Built-in performance metrics and profiling.

[2] *Recipes*, in the fully general sense of procedures and best practices. The term *recipe* will take on a more specific technical meaning later in this work.

[3] https://data.lsst.cloud

[4] https://aws.amazon.com/blogs/aws/new-astrocompute-in-the-cloud-grants-program/

[5] https://github.com/ratt-ru/vermeerkat

[6] https://slurm.schedmd.com/

[7] https://kubernetes.io/

Effectively, Stimela2 replaces the original version's Python-based API with a fully-fledged *domain specific language* (DSL) based on YAML. While this has the obvious drawback of presenting a learning curve to the user (noting that the original Python API in itself presented a steep learning curve), we believe this is outweighed by an important advantage, namely that the workflow and cab interaction logic becomes explicitly encoded in the DSL. Firstly, this makes recipes more concise and human-readable, with recipe structure less obscured by extra syntax. On the other hand, this gives Stimela2 itself better knowledge of the execution logic (and any inherent top-level parallelism), which it is then able to take advantage of by (a) deploying workflow steps in parallel when possible, (b) keeping track of data products and rerunning steps only when necessary (i.e. simplifying partial repetition of workflows).

Design aims. First and foremost, Stimela2 aims to serve the "research software" niche, introduced earlier in the paper series (Perkins et al., 2025). In our experience, this niche has become increasingly important due to (a) the pace of algorithmic development over the past decades, and (b) the data volumes produced by SKA precursor telescopes. Experimenting with, and validating, the emerging software packages on data from new telescopes requires the rapid deployment of many prototype workflows, with contributions from both software developers and black belt astronomers, who may be casual programmers at best. At the same time, successful research software can find itself pressed into production in lieu of alternatives, with the LOFAR MSSS survey's adoption of DDFacet/killMS (Tasse et al., 2018) being a prime example. Likewise, experimental workflows often need to be deployed at scale.

Stimela2 therefore aims for the middle ground between accessibility to the casual astronomer, and full-on production scalability. Usability is a cornerstone of the design, and our fundamental design decision is representing workflows by script-like linear recipes, on the presumption that these are more easily accessible to the casual astronomer-programmer (than e.g. a Python API or an explicit graph language), with advanced constructs such as scatter–gather enabling scalability. The Stimela2 *cab definition* scheme allows existing packages to be wrapped into "black boxes" (and ultimately containerized) non-invasively and with minimal effort, without involving the package developers. At the same time, recipe and cab definitions are sufficiently formalized that Stimela2 has sufficient information to be able to scale them up on e.g. a Kubernetes or slurm cluster. The goal of Stimela2 is to provide a workflow framework that is (a) easy to develop in, (b) deploys locally in a development environment with minimum fuss, (c) supports repeatability across different computing environments and is (d) reasonably performant and scalable when deployed in an HPC environment. As we shall see later in this work, meeting these goals enables (e) the deployment of Stimela2 workflows in a cloud environment.

Related work. The Common Workflow Language[8] (CWL), which originated in the biomedical and bioinformatics field, has many similarities with Stimela2, particularly containerization, and the use of a YAML-based DSL. During the original redesign, we intended for Stimela2 to be fully based on CWL, however, a number of limitations in the CWL specification proved difficult to overcome (first and foremost, poor support for read-write access to Measurement Sets), and we eventually found it impossible to meet the aims given above with a CWL-based solution. CWL has evolved since then, and may have become "friendlier" to radio astronomy, and efforts to adopt it continue. Notably, the LOFAR project is pursuing a reimplementation of the Prefactor pipeline using CWL.[9] Tellingly, the Prefactor-CWL project is a formal software development effort, with a team of developers (Loose, priv. comm.) pursuing the implementation of one big production pipeline, which does not suggest that this is a solution suitable to the casual astronomer-programmer.

DALiuGE (Wu et al., 2017; Mei et al., 2022) is a framework that was specifically developed to address the large-scale data processing requirements of SKA1. DALiuGE is explicitly based on *dataflow programming* (DFP). The first paper of this series (Perkins et al., 2025) already introduced DFP in the context of high-performance numeric code – i.e., relatively low-level – implementations. DALiuGE applies DFP at the highest level – that of workflow management – by explicitly representing workflows via directed acyclic graphs (DAGs) that can be assembled via a GUI-based editor. The explicit graph approach is very well-suited to defining highly scalable, parallelized workflows, as demonstrated by Wang et al. (2020b), and is very different from Stimela2's linearly scripted DSL philosophy. It is interesting that the Stimela2 design started from a cornerstone of casual usability (hence the scripting philosophy) and evolved towards scalability (with the addition of the slurm and Kubernetes backends), while DALiuGE started from a base of extreme scalability, and is evolving towards better usability (Wicenec, priv. comm.); this is creating a space of use cases where the two packages overlap, and providing opportunities for potential collaboration and convergence.

Existing stimela2 applications. Previous papers in this series (Kenyon et al., 2025; Bester et al., 2025) demonstrated how to use Stimela2 to deploy and drive scalable workflows, including in a cloud environment. Stimela2-based workflows have also been used as the basis of published science results. In particular, an investigation of the RATT PARROT transient (Smirnov et al., 2024) was published along with a complete set of Stimela2 recipes for the (somewhat non-standard) data reduction procedures used in that work. These recipes seamlessly combine multiple software packages and snippets of Python code, and should be fully repeatable by anybody, given a sufficiently large compute node with only Stimela2 and Apptainer/Singularity installations, or, alternatively, access to an AWS EKS cluster.

Stimela2 is also the underlying technology for pipelines that represent standalone software artefacts of their own. The SolarKAT pipeline (Samboco et al., 2024) is a Stimela2 workflow that subtracts solar RFI from MeerKAT datasets, yielding images of the Sun as a byproduct. The Transient Radio Observations for Newbies (TRON) pipeline uses Stimela2 for high-time candence imaging and transient detection, and has already yielded first scientific results (Smirnov et al., 2025b,a).

About this paper. The aim of this paper is to discuss the Stimela2 design (briefly), and its application to radio astronomy workflows (in depth). Since the YAML-based DSL is both the major innovation of Stimela2, and the main part of the user experience, the bulk of the paper is dedicated to a tour of the DSL's philosophy, features and examples of use. For additional technical detail, the reader is urged to refer to the official documentation page.[10]

2. Main concepts

Recipes. At the most basic level, a workflow in Stimela2 is represented by a *recipe*. A recipe has a set of inputs and outputs (defined by its *schema*), and contains a sequence of steps. Each step either invokes a *cab*, or another recipe (which is then referred to as a sub-recipe), specifying a number of *parameter* values that are validated against the cab's or sub-recipe's schema. The command

```
$ stimela run recipe.yml myrecipe ms=foo.ms
```

tells Stimela2 to load recipe definitions from the file `recipe.yml`, and run a recipe from therein called `myrecipe`, setting the input named `ms` to the value `"foo.ms"` (Listing 1, but see also Appendix A for a real-life example).

[8] https://www.commonwl.org

[9] https://github.com/EOSC-LOFAR/prefactor-cwl

[10] https://stimela.readthedocs.io

```yaml
myrecipe:
 inputs:
  foo:
   dtype: str
   default: "this is the default setting for foo"
  ms:
   dtype: MS
   required: true
 steps:
  img:
   cab: wsclean
   params:
    ms: =recipe.ms
    size: [1024, 1024]
  cal:
   cab: cubical
   params:
    ms: =recipe.ms
```

Listing 1: A very simple and notional `recipe.yml`.

This notional example demonstrates some fundamental concepts: a recipe's schema defines zero or more inputs and outputs, a recipe contains a sequence of arbitrarily-named steps (here, `cal` and `img`), each step invokes a cab (`cubical` and `wsclean`), and at each step we specify the cab's inputs and outputs via a `params` section. Finally, parameters can reference other parameters (here, `=recipe.ms` means "use the value of the `ms` parameter of the parent recipe") in various interesting ways (Section 7.2).

The example recipe also illustrates the fundamental data structure employed throughout the Stimela2 architecture: the ordered dictionary (or *mapping*, in YAML parlance). An ordered dictionary is a set of keys and values; keys are unique; key order is meaningful (`img` comes before `cal`). Most constructs in Stimela2 are represented by such nested ordered dictionaries, which naturally map to the YAML syntax of indented sections. A recipe is a nested ordered dictionary containing keys (a.k.a. sections) such as `inputs`, `outputs` and `steps`; the latter is an ordered dictionary where each step's name is the key, and the value is an ordered dictionary defining the step, and so on all the way down. We will use the term *section* to refer to keys whose values are nested dictionaries, and *field* for keys whose values are primitive types. This terminology intuitively matches the visual layout of the YAML files.

Cabs. Cabs are the atomic tasks available to Stimela2. A cab has a defined set of inputs and outputs given by its *schema*. The *cargo* (i.e. content) of a cab can be a any executable shell command, a Python function, a CASA task, or indeed any snippet of Python code. Depending on the chosen *backend*, cabs can be executed natively on the host system, remotely, and/or as containers. Listing 2 is an example cab definition, corresponding to the `shadems` command. The `inputs` and `outputs` sections define a cab's parameter schema.

Unlike its first-generation precursor, Stimela2 itself does not include any standard radio astronomy cabs. Instead, it is designed to work with user-installable (and/or user-defined) collections of cabs. There is an "official" collection of cab definitions called cult-cargo[11] which is distributed via PyPI. This provides cabs for most popular radio astronomy packages, as well as for individual CASA tasks. It can be loosely thought of as the KERN suite[12] for Stimela2 (at least once it is reasonably complete — the current public release is a subset, but the suite is under active development). A corresponding set of versioned container images is uploaded to the https://quay.io registry, from where Stimela2 can automatically download them on demand (that is, the first time a cab is invoked) if a containerized backend is employed. Note that cult-cargo is only "official" in the sense that it is maintained directly by the Stimela2 team; other users or teams are free to create and distribute their own cab collections, and these can be intermixed in any given workflow (see Appendix A for an example).

In addition to this, the user can define arbitrary custom cabs, by writing YAML similar to Listing 2. The use-and-include features (Section 4.1) allow these definitions to be structured into modular, reusable and shareable libraries of cab definitions.

Schemas. A schema defines the inputs and outputs of a cab (or recipe, as we saw above). These play a crucial role in Stimela2. A schema is defined as a list of named sections, one per parameter, containing a data type specifier (the `dtype` field), plus help strings, policies, and other optional attributes. The type specifier uses the same syntax as the standard Python `typing` module, which allows for a rich variety of data types, including compound constructs such as

```
Union[int, str, Tuple[int, int]]
```

...meaning, in this case, that a parameter may be an integer, a string, or two integers. Stimela2 extends this with a few additional types, such as `File`, `Directory` and `MS`. It is also possible to specify a `choices` field, for a parameter than can only assume a restricted set of values.

Parameter validation. The type information in the schema allows Stimela2 to perform fairly extensive parameter validation. Our encouraged approach is to specify fairly strict schemas in the cabs, so as to catch as many user errors as possible (though developers remain free to defeat this purpose by employing permissive types such as `str` or `Any`). Validation is done in two stages. *Prevalidation* is performed before the recipe is executed; this checks the recipe and all steps for any missing required parameters, incorrect types for known parameters, missing files (for `File` and `MS`-type parameters), etc. This attempts to catch as many errors as possible up front. Additional *runtime validation* is done before and after each step. This catches errors in parameter values that arise at runtime (particularly in cases where the output of one step is used as the input of another step).

Substitutions and evaluations. These are performed on parameter values at runtime. We saw an example in the notional recipe, with `=recipe.ms` being used to refer to the `ms` input of the parent recipe. Any parameter value that starts with an "=" sign is treated as a formula *evaluation*. For example, `=current.x + 1` is evaluated as the value of the parameter `x` of the current step, plus one. The formula language is explained in more detail in Section 7.2. *Substitutions* on strings are invoked using the curly braces construct (inspired by and implemented via Python format strings). Using something like `new-{recipe.image_name}.fits` would result in taking the value of the `image_name` parameter of the recipe, and adding the prefix `new-` and the suffix `.fits`. This is a very powerful mechanism for keeping the "million papercuts" problem under control. A typical workflow will generate dozens of data products, and will have fairly complex parameter interactions between steps. Using substitutions and evaluations makes it much easier to keep track of parameters, and to enforce naming conventions for output files.

2.1. Running recipes and accessing documentation

As seen above, the basic way to run a recipe is to specify a YAML file, a recipe name, and the required parameters, if any:

```
$ stimela run recipe.yml myrecipe ms=foo.ms
```

If the document contains a single recipe only, the recipe name is not necessary. Alternatively, `run -l` runs the *last* recipe in a document.

Running

[11] https://github.com/caracal-pipeline/cult-cargo
[12] https://kernsuite.info

```yaml
name: shadems
image:
 name: shadems
command: shadems
info: 'Rapid Measurement Set plotting with xarray-ms
 and datashader'

defaults:
 norm: auto
 xcanvas: 1280
 ycanvas: 900

policies:
 prefix: '--'
 positional: false
 repeat: list

inputs:
 ms:
  info: Measurement set to plot
  required: true
  dtype: MS
  policies:
   positional: true
 xaxis:
  info: 'X axis to plot. Can be any MS column name,
   also: CHAN, FREQ, CORR, ROW, WAVEL, U, V, W, UV,
   and, for complex columns, keywords such as amp,
   phase, real, imag. You can also specify
   correlations, e.g. DATA:phase:XX. The order of
   specifiers does not matter.'
  dtype: str
 yaxis:
  info: 'Y axis to plot. Must be given the same
   number of times as xaxis.'
  dtype: str
 ...
```

Listing 2: Example of a cab definition.

```
$ stimela doc recipe.yml
```

prints documentation on the recipe and its parameters (using the embedded `info` strings which, hopefully, have been provided by the recipe developer).

The `doc` command can also document cabs. For example, assuming the `cult-cargo` package is installed,

```
$ stimela doc cultcargo::wsclean.yml
```

will print complete documentation on the WSClean cab (note that the "`::`" syntax on the command line is shorthand for "find Python module named `cultcargo` and look for the specified YAML file therein").

Finally, note that both `run` and `doc` will accept multiple YAML documents and *compose* (see next section) them together before looking at the recipe. This makes it straightforward to separate the logical recipe per se from, say, local configuration tweaks, and compose the two with:

```
$ stimela run recipe.yml config.yml myrecipe
```

3. Architecture

A top-level diagram of Stimela2's architecture and interaction with the computing environment is presented in Fig. 1. The basic input to Stimela2 is a YAML recipe and optional YAML configuration files; together with cab definitions included by the recipe, these are composed (Section 4) into a global config namespace which is then passed to the *recipe runner*. The recipe runner is responsible for executing the logic of the recipe, and passing data between steps; it invokes *step runners* to execute each step. If a scatter construct is employed (Section 7.4), these are executed via a process pool, using the Python `concurrent.futures` API. The step runner is responsible for executing one step of the workflow, and can be seen as the "local" representation of a (possibly, remotely executed) step. It is also responsible for collecting console output and error conditions from its workload, and passing them to the *console manager* (implemented using the Python `rich` library) for display and logging. A step runner can either invoke a nested recipe, or execute a cab.

Cabs are executed via *backends* (Section 8). In a local node deployment, the *native* and *Apptainer* backends then directly launch the local executable or Apptainer container. In a SLURM deployment, the SLURM *wrapper* interacts with these backends to wrap the commands and pass them to the SLURM scheduler. In a Kubernetes deployment, the Kubernetes backend calls the K8s API to execute cabs on the configured Kubernetes cluster.

A separate *profiler* component (Section 9.2) interacts with the OS and/or with the K8s API to collect profiling information and pass it to the console and logger.

4. YAML, nested namespaces and composability

The nested dictionary, which we also call the nested namespace (and YAML calls a mapping), is the primary data structure of Stimela2. Pretty much everything in the system is represented by a nested dictionary, including the overall system configuration (a.k.a. the *config namespace*). At the top level, the config namespace has the following structure:

```yaml
cabs:      # cab definitions
 foo:      #  defines cab foo
  # ...
 bar:      #  defines cab bar
  # ...

lib:       # libraries of reusable components
 recipes:  #  library of recipes
  my_recipe:
   # ...
 # other free-form library entries

vars:      # arbitrary user-defined variables
 x: 0
 y: "a string"

opts:      # runtime options
 log:      #  logging options
  # ...

run:       # runtime environment information
 date:     #  date the run started
 time:     #  time the run started
 node:     #  hostname of current node
 env:      #  environment variables from shell
  PATH:
  # ...
```

The config namespace is populated at startup, firstly from (a) the Stimela2 configuration, (b) runtime information (in particular, for the `run` namespace), and finally (c) user-specified configuration and recipe files, which define recipes and include (see below) cab definitions.

4.1. YAML extensions: use, include, merge and augment

Bringing configuration information together from a large set of sources (YAML files) has necessitated some additional design features

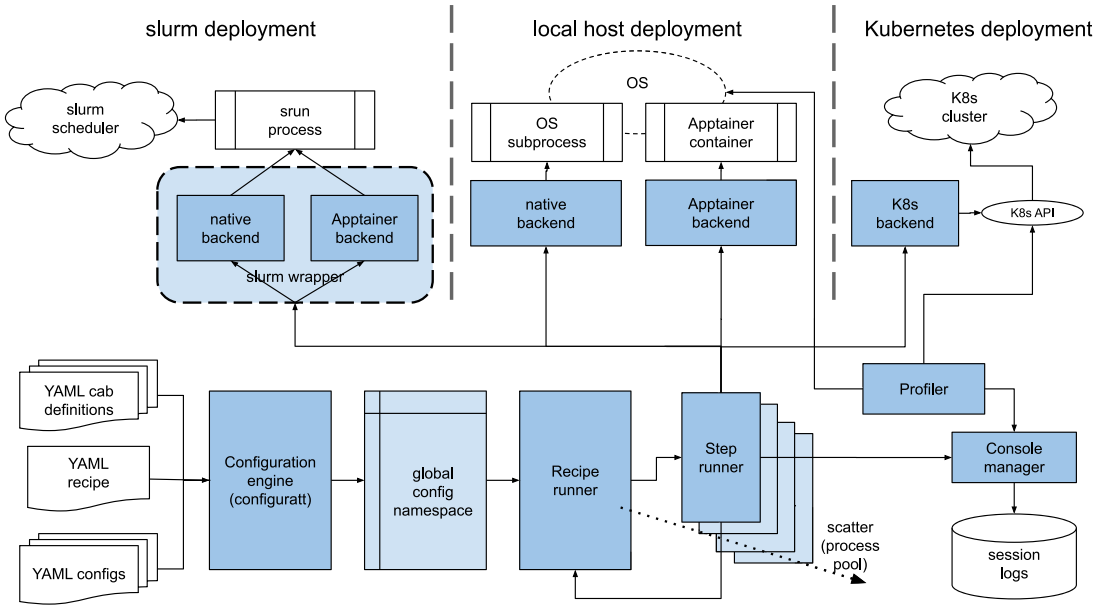

Fig. 1. A schematic diagram of Stimela2's architecture and interaction with the environment in three deployment scenarios. The shaded boxes represent components of Stimela2.

to enable modularity and re-use of YAML. The YAML standard[13] offers some rather rudimentary features (anchors, aliases and merge) in support of this. A package called OmegaConf[14] provides a number of very useful extensions such as variable interpolation, as well as a very convenient API for working with large YAML configurations and mapping them to Python dataclasses. This package was adopted by Stimela2 as the underlying technology. On top of this, Stimela2 defines a module called `configuratt` which implements a number of useful YAML extensions supporting modularity and libraries. In particular, these are the `_include` and `_use` keys.

The core idea behind these constructs is very simple, and familiar to any programmer: they allow for snippets of YAML to be defined in one place, and reused elsewhere, library-style. An `_include` key *merges* the entire content of one or more YAML file(s) into the section where it appears. The `_use` key merges the content of one or more previously defined sections into the section where it appears. All other content in the section then *augments* the merged content — that is, merged-in keys may be modified, and new keys may be introduced. These constructs can be nested (i.e., `_included` documents can contain their own `_includes`) and intermixed (i.e., `_use` can refer to sections defined by previously `_included` content).

Further details are best left to the technical documentation — here we rather provide a real-life illustration. Consider the case of CASA tasks. The cult-cargo package provides cab definitions for many CASA tasks, allowing them to be invoked individually from any step of the recipe. A number of these tasks (particularly, those that operate on visibility data) share a common collection of parameters which are used to specify the input measurement set, and data selection to be performed on that measurement set. Rather than repetitively listing these parameters within each cab's schema (which would be laborious and error-prone), we can define this collection of parameters once in a "library" file called `casa.yml`:

```
lib:
 params:
  casa:
   mssel_inputs:
    # standard arguments for CASA tasks that take
    # a vis argument, and a standard set of data
    # selection options
    ms:
     info: Name of input visibility file
     dtype: MS
     writable: true
     required: true
     nom_de_guerre: vis
    field:
     info: Select field using field id(s) or
      field name(s)
     dtype: str
    spw:
     info: Select spectral window/channels
     dtype: str
    # ...
```

We can then reuse this library content within any cab definition:

```
_include: casa.yml

cabs:
 casa.applycal:
  inputs:
   # pull in standard set of selection parameters
   _use: lib.params.casa.mssel_inputs
   # and now define some task-specific inputs:
   docallib:
    info: Use callib or traditional cal apply
     parameters
    dtype: bool
   # ...
```

The `_include` key may refer to a full path. Relative paths (and bare filenames) are resolved by looking in (a) the current directory, (b) the directory of the document invoking the include, (c) directories specified by the `STIMELA_INCLUDE` environment variable, and (d) some predefined locations such as `/lib/stimela`, `/usr/lib/stimela`, etc. A special form of the path allows one to refer to a Python package, e.g.

```
_include: (cultcargo)wsclean.yml
```

will tell Stimela2 to look for `wsclean.yml` within the installation directory of `cultcargo`.

[13] https://yaml.org/spec/1.2.2
[14] https://github.com/omry/omegaconf

While at most one `_include` key is allowed per section,[15] it is possible to include multiple files by providing a sequence (list) instead of a single path. The `_include` can also be structured if multiple files from the same location need to be pulled in:

```
_include:
 (cultcargo):
  - wsclean.yml
  - cubical.yml
```

The applications of these constructs are plentiful and useful: libraries of recipes, libraries of step templates, reusing and augmenting step definitions within a recipe, etc. Programmers intuitively recoil at repetition, for both aesthetic and practical reasons — it tends to promote hard-to-spot errors. The `_use` and `_include` constructs can serve to avoid repetition and promote modularity within YAML documents. They are the key enabling technology that allows Stimela2 to work with distributable collections of cabs such as cult-cargo.

4.2. Composability

The crucial concept enabled by the above constructs is that of *composability*. Before it can run a workflow, Stimela2 needs to know many things, from the top-level recipe steps, to cab definitions, to the local (or cluster) environment configuration. Although it is perfectly possible to write a recipe file that provides all this information within a single big YAML document, this is hardly the recommended approach, for obvious reasons. Instead, one can leverage the use/include/merge/augment features described above to compose a workflow description from *multiple* YAML files, which are ultimately all merged into the global configuration namespace. For example, a workflow such as that published by Smirnov et al. (2024) is composed of multiple documents:

- The top-level recipe, describing the logic of the workflow (inputs, outputs and steps), provided by the recipe developer;
- Cab definitions for standard tools, provided by the cult-cargo package;
- Cab definitions for custom tools and scripts, provided by the recipe developer;
- Sub-recipes employed within the main recipe, provided by the recipe developer;
- Local compute environment configuration (e.g. Kubernetes cluster setup), based on information provided by the system administrator;
- Optional additional configuration tweaks (e.g. to optimize performance), provided by the end user.

Composability promotes modularity and re-use of cab definitions and sub-recipes, and allows for a clear separation of responsibilities, particularly between the pure logic of the workflow, and the nuts-and-bolts specifics of the local compute environment. Appendix B provides an illustrative example of this.

5. Cabs and cargo

A **cab** defines an atomic task available to Stimela2, with a defined set of inputs and outputs specified by a *schema*. The *cargo* (content) of a cab can come in a variety of *flavours*: an executable command, a function defined within a Python module, an individual CASA task, or even a snippet of inlined Python code.

[15] A section is a YAML mapping, after all: the keys of a mapping must be unique.

5.1. Cab definitions

A cab definition is simply a section of YAML, Fig. 2 being an example. The illustrated cab will run the `shadems` command. If a containerized backend is employed, it will use a container image called `stimela2/shadems`.

The `inputs` section defines the input parameter schema (see Section 6). An optional `defaults` section specifies default settings for parameters (alternatively, these can be embedded in the schema itself). The `policies` section tells Stimela2 how to convert parameter values into command-line arguments (see Section 6.3).

5.2. Cab flavours

The default cab flavour runs the specified executable command (natively, or within a container, depending on the configured backend), passing it arguments according to the specified policies. There are a few additional flavours available.

The `python` flavour will invoke an arbitrary Python function, in which case the `command` key specifies the function to be called as [*package.*]*module.function*. The function's signature (i.e. arguments) must be correctly described by the cab schema. The function's return value can be interpreted as a cab output. Note that the function is invoked via an external Python interpreter (possibly in a separate virtual environment or a container).

The `casa-task` flavour will invoke a CASA task. The `command` key then specifies the task name. The task's signature (i.e. arguments) must be correctly described by the cab schema.

The `python-code` flavour can execute arbitrary Python code directly embedded into the cab definition. The `command` key then specifies the code itself (YAML's use of the "|" character to designate an indented multi-line string is particularly handy here). The cab's inputs are mapped to local variables before the code snippet is executed, and the outputs are collected from local variables after the code snippet completes. This is useful for small tasks and "glue code". For example, the cab below will subtract the mean value from a FITS image, and return the resulting image and mean value as outputs:

```
cabs:
 subtract-mean:
  inputs:
   image:
    dtype: File
    required: true
  outputs:
   output-image:
    dtype: File
    required: true
   mean-value:
    dtype: float
  flavour: python-code
  command: |
   from astropy.io import fits
   ff = fits.open(image)
   mean_value = ff[0].data.mean()
   ff[0].data -= mean_value
   ff.writeto(output_image, overwrite=True)
```

As an aside, the above example illustrates that YAML keys (and therefore cab names, parameter names, step labels, etc.) can contain dash characters (moreover, dashes within step names can be put to serve a useful purpose — see Section 7.2.) Since Python variable names cannot contain dashes, Stimela2 will implicitly map them to underscores when communicating with a `python-code` or `casa-task` cab.

5.3. Console output & wranglers

When running a cab, Stimela2 will intercept its console output and send it to its own logger (and console). There are cases when this output contains information that is required later in the workflow. For such cases, Stimela2 provides a mechanism called *output wranglers*, which can parse and manipulate a cab's textual output in various ways, and capture its content into formal output parameters.

This is best illustrated by an example. Imagine that a workflow needs to know the fraction of visibilities flagged within a given measurement set. A short `python-code` flavour cab can be readily defined, containing code to open the measurement set, compute this fraction, and return it in a cab output, as described above. An alternative is to invoke the CASA `flagdata` task, ask it for a summary, and parse the value out of its output. (A more general version of this use case is exemplified by any external software package that prints some important item of information midway through its run — how can we automatically capture and parse this information?) This can be done via the following cab definition:

```
cabs:
 flagsummary:
  command: flagdata
  flavour: casa-task
  inputs:
   ms:
    dtype: MS
    required: true
    nom_de_guerre: vis
   mode:
    implicit: 'summary'
  outputs:
   percentage:
    dtype: float
  management:
   wranglers:
    'Total.* Counts: .*
     \((?P<percentage>[\d.]+)%\)':
      - PARSE_OUTPUT:percentage:float
      - HIGHLIGHT:bold green
```

The `management.wranglers` section contains a sequence of regular expressions (employing standard Python `re` module syntax), which are matched against each line of the output. A matching line then causes a number of actions to be invoked. In this case, the first action captures the flagged percentage (using a named regex group) and turns it into a `float` value that is returned as the `percentage` output of the cab. The second action highlights the matching line in bold green text, which is simply a cosmetic convenience for the user.

Other useful wrangler actions can mark a cab as having succeeded or failed based on encountering some particular text output. By default, and as per normal Unix practice, Stimela2 determines success or failure via the return code of the underlying process. Certain software (notably, CASA) does **not** return error codes on failure — the wrangler feature allows Stimela2 to detect these error conditions anyway.

As another aside, note the `implicit` definition for the `mode` parameter of this cab. An implicit input is something that is not supplied by the user, but needs to be passed to the underlying cargo anyway. In this case, we pass `mode='summary'` to the `flagdata` task to force it into summary mode. (Likewise, implicit outputs can be used to describe the file-type outputs of a cab over the naming of which the user has no control.)

5.4. Cult-cargo and custom cargo

The cult-cargo package is an (optional) companion to Stimela2. This provides cab definitions for a number of standard radio astronomy packages, as well as for individual CASA tasks. cult-cargo is shipped through PyPI – to use it, one only needs to `pip install` it alongside Stimela2. The cabs within can then be directly included into a recipe, as shown in Section 4.1.

cult-cargo does not install the actual packages per se — only their cab definitions. It does, however, have an associated container image registry on https://quay.io, where a collection of versioned images is maintained. References to these images are provided within the cab definitions. The upshot of this is that the end user need only specify that they want to use a containerized backend such as Apptainer/Singularity (Section 8) to use these images. Everything else is taken care of automatically by Stimela2, making for a true installation-free workflow. On the other hand, developers or power users who want to run local custom installations of some (or all) packages are free to do so, by switching to the native backend. Note that it is also possible to specify backends on a per-cab (or per-step) basis, so a developer needing to experiment on some small part of the pipeline can temporarily switch to a native install for a particular step or cab, while continuing to use standard images everywhere else. This is done by augmenting the cab definition after including it:

```
_include: (cultcargo)wsclean.yml
cabs:
 wsclean:
  backend:
   select: native
  command: /path/to/my/wsclean
```

The cult-cargo registry provides multiple image versions for each package. The default image is usually the latest stable release. However, if one wanted to experiment with a different version of the package for which cult-cargo provides an image, it trivially achieved by augmenting the definition with a specific version label:

```
_include: (cultcargo)wsclean.yml
cabs:
 wsclean:
  image:
   version: 2.10.1-kern7
```

Developers can also provide customized cab definitions, for packages that have not (perhaps yet) made it into cult-cargo. For example, the PARROT recipes (Smirnov et al., 2024) use some custom cabs from the `omstimelation`[16] collection. At time of writing, this is not a PyPI package, but rather just a git repository. To use it, once needs to clone the repository to a location in which Stimela2 knows to search for includes (Section 4.1), and then invoke is as:

```
_include: omstimelation/oms-cabs-cc.yml
```

For custom cabs relying on container images, it is left up to the developers to set up an image registry and make their images available (and to specify their location in the cab definition) in order for their cabs to work with a containerized backend. The cult-cargo approach provides a distribution model that can be readily followed. Note that images do not need to be specified for `python`, `python-code` or `casa-task` flavour cabs, since by default, these will use standard Python and/or CASA images provided by cult-cargo.

6. Schemas

Schema definitions are at the core of Stimela2's parameter definition and validation system. A schema defines the input and output parameters (the signature, in other words) of a cab or a recipe, and (optionally) a set of *policies* that specify how the parameters are passed to the underlying cargo.

[16] https://github.com/o-smirnov/omstimelation

Some examples of schemas were already presented above, see e.g. Section 4.1. Fundamentally, each cab or recipe definition contains an `inputs` and/or an `outputs` section, with named subsections defining individual parameters. These may be further nested if one wants to impose some kind of organizational hierarchy onto the parameters. For example, this schema

```
inputs:
 data:
  ms:
   dtype: MS
   required: true
   info: Measurement set to process
  column:
   dtype: str
   default: "DATA"
   info: column name
outputs:
 image:
  dtype: File
  required: true
  info: output image
```

...would specify two inputs, `data.ms` and `data.column`, and one *named file* output, `image`. (A named file output refers to the common situation where the user needs to specify the name of an output file or directory. Thus the file*name* is, in some sense, an input — however, the file itself is an output product, and is treated as such by Stimela2. An alternative scenario is an *implicit* file output, where the user has no control over the filename, as it is determined by the underlying tool itself. This can be specified in the schema via an `implicit` keyword supplying the filename.)

The `dtype` entry specifies the type of the parameter, using the type annotation language provided by the `typing` package of the Python Standard Library. Besides the built-in primitive types (`bool`, `str`, `int`, `float` and the like), Stimela2 defines a few additional primitives (`File`, `MS`, `Directory`). These can be combined with the annotation language to define rich types such as `List[File]`, or even something like

```
dtype: Union[str, Tuple[str, float]]
```

which is useful for e.g. the imaging weight parameter, specifying something that can be either a single string ("uniform"), or a string and a number ("robust 0").

Other schema options allow one to specify a parameter that can only assume a fixed set of values (`choices`), to set a default value, to mark a parameter as `required`, and to remap the name of the parameter when passing it to the underlying cargo (`nom_de_guerre`).

6.1. Shorthand schemas

For those in a rush, Stimela2 also supports a shorthand schema syntax reminiscent of Python function signature annotations. The schema above can be expressed in shorthand as:

```
inputs:
 data: MS * "Measurement set to process"
 column: str = "DATA" "column name"
outputs:
 image: File * "output image"
```

A shorthand schema contains the type, an optional default, an optional "*" character indicating the parameter is required, and an optional trailing documentation string. To specify any other options, one must resort to a full schema definition as per the previous section. Shorthand and full schemas may be intermixed.

6.2. Parameter validation

The point of having well-defined schemas is to impose some robustness on the recipe definitions, and to catch errors as early as possible, rather than letting workflows fail midway through due to, say, missing parameters. Stimela2 accomplishes this in two stages. A *prevalidation* process is performed up front, before the recipe is executed. This checks that all required parameters for all steps are defined, that the supplied types (of known parameters) are correct, that file-type inputs to the recipe are present (unless marked as `must_exist: false` in the schema), etc. The scope of pre-validation is naturally restricted — after all, any non-trivial pipeline will have steps that depend on the outputs of preceding steps, which cannot be known upfront. Further *runtime validation* is therefore performed before and after executing each step. This is a more thorough check ensuring that all the inputs to the step (which, by this point, are fully defined) match the schema. A similar check is done on the outputs of the step.

6.3. Parameter policies

Where the schema describes a (sub)recipe, or a cab with a Python function (or a CASA task) as its cargo, the information in the schema is necessary and sufficient for Stimela2 to invoke the cargo and pass arguments to it. If the cargo wraps a command-line tool, there is typically a one-to-one correspondence (modulo nom-de-guerre's) between the entries in the schema and the command-line parameters of the tool. However, different tools employ different command-line syntax conventions (a single dash prefixing options? A double dash? Are some arguments positional rather than optional? Does the tool rather use *`key=value`* for its arguments?) The optional `policies` section of the schema is used to describe these argument-passing conventions. A default overall set of policies can be defined at cab level, and then redefined and tweaked at the level of individual parameters as needed. The variety of settings provided by the `policies` section is sufficiently rich to represent most command-line conventions found in the wild. We leave details of this to the technical documentation.

6.4. Dynamic schemas

Some tools, QuartiCal (Kenyon et al., 2025), and its predecessor CubiCal (Kenyon et al., 2018) being notable examples, have mutable command-line schemas, in the sense that the values of some command-line arguments can affect the structure of following command-line arguments. For these two tools in particular, there is a "Jones chain" argument that specifies a list of Jones matrices to be included in the solution (e.g. G, K), in response to which the command-line parser starts recognizing extra arguments for specific settings related to these Jones terms (e.g. `--g-type`, `k-type`, etc.) Stimela2 supports this by providing a *dynamic schema* mechanism. A cab definition can include a callable Python function which takes in initial argument settings, and returns an updated schema for the cab.

6.5. Using schemas as a CLI builder

For any third-party command-line tool, most of the information in the cab schema (i.e. argument names and types, help strings) directly mirrors that already provided to the tool's command-line parser. When wrapping a third-party package in a cab, this leads to an unavoidable duplication of effort (with all the attendant potential for inconsistencies) – after all, the package developer has already implemented their own command-line interface (CLI) parser, and this CLI needs to be described to Stimela2. Note, however, that the schema itself provides all the information that would be needed to construct a CLI in the first place. For newly-developed packages, this provides a substantial labour-saving opportunity. Stimela2 includes a utility function that can

convert a schema into a CLI using the `click`[17] package. Given a separate YAML file defining the cab schema, this is as simple as:

```
#!/usr/bin/env python
import click
from scabha.schema_utils import clickify_parameters

@click.command()
@clickify_parameters("hello_schema.yml")
def hello(count, name):
  """Simple program that greets NAME for a
    total of COUNT times."""
  for x in range(count):
    print(f"Hello {name}!")

if __name__ == '__main__':
  hello()
```

For newly-developed code, this mechanism ensures that all inputs and outputs need only be defined by the developer once, in a single place — and provides both a CLI and Stimela2 integration with no additional effort, while ensuring that these are mutually consistent by construction. The QuartiCal and pfb-imaging packages discussed earlier in this series (Kenyon et al., 2025; Bester et al., 2025), for example, make extensive use of this functionality.

7. Recipe structure

A recipe is defined in a YAML document that Stimela2 merges into its config namespace (Section 4) on startup. As such, it can augment the content of any of the standard config sections, i.e. `opts` to modify runtime options, `cabs` to provide new cab definitions or to modify existing cabs, etc. Any top-level section in this document that does not match a config section is implicitly treated as a recipe body (and subsequently moved to `lib.recipes`.[18])

The recipe body consists of the following elements:

- Optional `name` and `info` fields giving a recipe name and description;
- `inputs` and `outputs` sections containing schemas for the recipe's input and output parameters;
- An optional `defaults` section specifying default parameter values (these can also be embedded directly in the schemas);
- An optional `aliases` section (Section 7.3);
- An optional `assign` section specifying variable assignments;
- An optional `for_loop` section which specifies that the recipe is a loop or a scatter–gather construct (Section 7.4);
- An optional `backend` section to tweak recipe-specific backend settings (Section 8);
- A `steps` section specifying the sequence of steps.

7.1. Step definitions

Each step of the recipe is defined by a subsection under `steps`. The key of each subsection is the step label, while the body of the subsection contains:

- An optional `info` field documenting the step.
- The cargo: this is either a `cab` field specifying a cab, or a `recipe` field specifying a sub-recipe. Normally, these are specified by name, but it is also possible to make this field into a subsection that directly embeds a full cab definition or sub-recipe definition.
- A `params` section specifying the step parameters.
- An optional `assign` section.
- An optional list of `tags` associated with the step (Section 7.5).
- Optional `skip` and `skip_if_outputs` fields (Section 7.5).
- An optional `backend` section to tweak step-specific backend settings (Section 8);

The steps are executed in sequence. Before each step is run, Stimela2 performs evaluation and substitution on the parameters, matching them against the schema (the inputs) of the cargo. Any missing or invalid parameters cause execution to stop with an error. Likewise, after a step is run, its outputs are matched against the schema.

7.2. Substitutions, evaluations, namespaces

As briefly seen in Section 2, parameters of steps can refer to other parameters via constructs such as ``=recipe.ms'' and ``{recipe.ms}''. These are examples of Stimela2's more general substitution and evaluation mechanism, which is invoked when evaluating step parameters (as well as in a few other contexts, such as backend settings and logging options, see below.)

The mechanism has two basic rules: (a) a string that starts with an "=" character is parsed using the formula syntax, and (b) all string values are subject to {}-substitution. Thus, the above two examples achieve identical results in very different ways.

Namespace lookup. In both cases, the underlying element (the variable in a formula, or the value being substituted in), `recipe.ms`, is an instance of a *namespace lookup*. Some of the namespaces available for evaluation and substitution include:

`recipe` contains the parameters (inputs and outputs) and variables (see below) of the current recipe.

`current` contains the parameters of the current (i.e. "this") step in the recipe.

`previous` contains the parameters of the directly preceding step.

`root` contains the parameters of the top-level (root) recipe.

`steps.foo` contains the parameters of (a necessarily preceding) step called "foo". Note that when doing namespace lookups, Stimela2 recognizes the wildcard character "*" and interprets it in the sense of *last matching element*. Thus, `=steps.make-image-*.output-image` would refer to the `output-image` parameter of the closest *preceding* step (preceding, because at any given step, only the preceding steps are present in the `steps` namespace) whose label matches the `make-image-*` pattern.

`info` contains information about the current step. For example `info.label` is the step label, and `info.suffix` is the *suffix* of the label, if defined (that is, if the label contains dash characters, e.g. "`image-1`", then the part following the last dash is the suffix, in this case "`1`").

`config` provides access to Stimela2's entire config namespace (see Section 4). For example, `config.run.env.HOME` provides the value of the `HOME` environment variable.

Formula language. Any parameter value starting with "=" is parsed as a formula (unless it starts with "==", which evaluates to single equals sign, and treats the rest of the string as a literal). The formula language is patterned on basic Python syntax, and recognizes the following elements:

- atomic elements, i.e. namespace lookups, as well as numeric and string constants.

[17] https://click.palletsprojects.com/

[18] Alternatively, recipes may be defined under `lib.recipes` directly, but this adds two extra levels of indentation to the document.

- standard Python arithmetic and logical operators.
- the UNSET keyword. If the formula evaluates to UNSET, the corresponding parameter is deleted from the set of parameters passed to the step.
- a number of built-in functions. Please refer to the technical documentation for a complete list of available functions.

An example of combining namespace lookup with formulas would be the following:

```
steps:
  convert-to-fits:
    image: =recipe.image
    output-fits: =STRIPEXT(current.image) + ".fits"
```

This sets the "image" parameter of the step to that of the recipe, and forms an "output-fits" filename by replacing the input filename's extension by `.fits`.

Curly brace substitutions. The {}-substitution syntax is equivalent to Python format strings, since it uses that very mechanism under the hood. The braces must contain a valid namespace lookup, with an optional `:format` suffix to control the precise formatting of the string.

Although both achieve the same means, there is a subtle difference between `{recipe.ms}` and `=recipe.ms`. The former does a namespace lookup and then formats the value as a string; the latter does a namespace lookup and returns the value directly. For string-type values, the result is the same, whereas for numeric-type values, the resulting data type will be different. (However, since Stimela2 knows to convert strings into numbers where a numeric parameter is defined in the schema, the actual outcome may end up being effectively the same anyway.)

Variable assignments. In addition to parameters, a recipe may define arbitrary variables (loosely speaking, the distinction is similar to that between the arguments and local variables of a function.) These become available for namespace lookup within the `recipe` namespace. Variables are primarily useful for setting up parameter substitutions and enforcing things such as filename conventions. They are defined via the `assign` and `assign_based_on` sections. We refer the reader to the technical documentation for details.

7.3. Aliases

Often, a recipe parameter is directly passed to a step as is. In the example above, this is achieved via a `=recipe.ms` evaluation. *Aliases* provide a way to formalize this relationship. Instead of specifying an evaluation (or substitution) at the step level, one could remove the definition of the recipe's `ms` input, and add an `aliases` section instead:

```
  aliases:
    ms:
    - image-1.ms
```

This tells Stimela2 that the recipe has an input (or output) named `ms` with the exact same schema as the `ms` input of step `image-1`, and that the value of `ms` should be passed to that step directly. The `ms: =recipe.ms` setting at step level is then obviated. The difference between an alias and an evaluation or substitution is that an alias is a more formal, hard link — it implicitly declares a matching schema at recipe level, and the input parameter can be checked up front before the recipe is run (whereas `=recipe.ms` is only evaluated before the step itself is run, so no errors can be caught until then.) An alias can link a recipe parameter to multiple steps:

```
  aliases:
    ms:
    - image-1.ms
    - calibrate-1.input-ms.path
    - image-2.ms
```

The above can also be specified more economically via wildcards:

```
aliases:
  ms:
  - image-*.ms
  - calibrate-*.input_ms.path
```

or by referring to all instances of a particular cab:

```
aliases:
  ms:
  - (wsclean).ms
  - (quartical).input_ms.path
```

In certain situations, aliases are generated automatically. For example, if the recipe contains an `image` step referring to a cab with a mandatory `ms` parameter, but a value for `ms` is not provided within the step body, Stimela2 will automatically generate a recipe-level alias named `image.ms`, which then becomes a required recipe parameter. More subtly, any *optional* step parameters that are not explicitly set in the step are also auto-aliased to recipe-level parameters (using the same naming scheme). Stimela2 considers such recipe parameters "obscure" or "hidden" (depending on whether the parameter has a default), since the recipe can be operated without the user needing to know anything about them. However, these parameters are still accessible from the command line (and can also be set when the recipe is invoked as a sub-recipe), should the user need to tweak them. The `stimela doc` command does not list obscure or hidden parameters by default, but provides an option for it (see `stimela doc --help`).

7.4. For-loops

A recipe can be turned into a loop by providing a `for_loop` section:

```
for_loop:
  var: counter
  over: [0,1,2]  # or equivalently, =RANGE(3)
```

This declares a recipe-level `counter` variable, and iterates the recipe three times, for the three different values of `counter`. Instead of providing a list of values directly, the `over` field can also refer to another recipe variable or input:

```
my-recipe:
  inputs:
    ms-list:
      dtype: List[MS]
  for_loop:
    var: ms
    over: ms-list
    scatter: 4
```

The `scatter: 4` setting tells Stimela2 to run up to four iterations of the loop in parallel (use -1 to run all iterations in parallel). If a distributed backend such as Kubernetes or Slurm is configured (Section 8), this will distribute jobs across the cluster (and if not, all four jobs will run on the local machine — it is up to the user to ensure that this has enough resources to support the parallelism.)

Note that Stimela2's composability features (Section 4.2) make it trivial to turn a non-looping recipe into a looping one. Let us say we have a `recipe.yml` document defining a (non-looping) recipe called `my-recipe`, with an `ms` input. By placing the YAML snippet above into a separate document called `loop-recipe.yml`, and running

```
$ stimela run recipe.yml loop-recipe.yml
  'ms-list==GLOB('*.ms')"
```

we augment our recipe into a for-loop over all matching measurement sets.

7.5. Skips and tags

The optional `skip` field of a step can be used to tell Stimela2 to skip the step. A simple `skip: true` setting is what is called a force-skip — normally, it is the equivalent of commenting the step out (note, however, that the `-s` option to `stimela run` can be used to override the skip and execute the step anyway.) More interestingly, one can declare a *conditional* skip based on a formula evaluation. For example,

```
skip: =EXISTS(current.output-file)
```

will tell Stimela2 to skip the step if its `output-file` parameter refers to an existing file. This is such a common use case in workflows with expensive steps that Stimela2 supports an explicit `skip_if_outputs` field:

`skip_if_outputs: exist` will cause a step to be skipped if all its file- and directory-type outputs exist.

`skip_if_outputs: fresh` will cause a step to be skipped if all its file- and directory-type outputs are "fresh", in the sense that their modification times are newer than or equal to that of the most recent file- or directory-type input. Those familiar with Unix-style Makefiles will readily recognize this logic.

The optional `tags` field serves a related purpose, and can be used to group related steps of a long workflow into subsets that can be selected or deselected for execution *en masse*.[19] In a nutshell, invoking

```
$ stimela run recipe.yml -t foo
```

will run only the steps that have a `foo` (and/or `always` tag — the latter having this special meaning), and skip all others. A second special tag is `never`. Steps tagged with `never` are normally skipped, unless they have another tag that has been explicitly specified with `-t`.

8. Backends

Stimela2's *backend* components are responsible for actually scheduling and executing the jobs (cabs). At time of writing, the following backends are supported:

Native. This backend runs the command (or Python interpreter, or CASA) natively on the host system. Optionally, a separate virtual environment may be specified for any given cab or step. The native backend provides the most flexibility in a development or experimental environment, but the onus is entirely on the user to make sure that all the underlying packages are installed and available on the host. Consequently, this offers the least reproducibility — running the recipe on another system requires that exactly the same versions of all software packages be installed.

Singularity/Apptainer. This backend runs the command (or Python interpreter, or CASA) using the Apptainer[20] containerization engine. The image from which the container is instantiated can be downloaded automatically from a container registry (such as `quay.io`, where the standard cult-cargo images are maintained, see Section 5.4) and built on the spot, or may be provided by the user as a local SIF (Singularity image format) file. At present, Apptainer is the container engine of choice for HPC environments.[21]

The Apptainer backend allows for true zero-install, fully repeatable workflows. The host system only requires an installation of Stimela2 and Singularity/Apptainer, while all required images are downloaded on-demand (but can also be pre-downloaded by the user). The host system must, of course, have sufficient resources (primarily, RAM and disk) to accommodate all the steps of the workflow.

Kkubernetes. This backend runs the command (or Python interpreter, or CASA) on a Kubernetes cluster in a *pod*, using a container associated with a registry. From the user's point of view, the Kubernetes backend is also zero-install — there is, however, considerable onus on the system administrator to configure and provide access to a Kubernetes cluster. Kubernetes has emerged as the technology of choice for scalable workflows, particularly cloud-based ones. The results shown earlier in this paper series use the Stimela2 Kubernetes backend to run workflows on the AWS platform. This backend will be described in some detail below.

Slurm. This backend (or strictly speaking, backend wrapper) can schedule jobs remotely via the slurm scheduler, using the native or Apptainer backends to actually run the jobs on the compute nodes.

Specifying backends. Backends are specified via a `backend` section in four possible locations: (a) the top-level `opts` config section, (b) the recipe definition, (c) the cab definition, and (d) a particular step definition. At each step, Stimela2 will compose the settings from the four locations in the order given here, and apply the resulting settings. In a routine end-user workflow, only the top-level backend (a) is specified — this can be as simple as adding

```
opts:
 backend:
  select: singularity
```

to the recipe file so as to run the entire workflow using Apptainer/Singularity. Options (b) through (d) are meant for development workflows, as well as tuning resource allocation under slurm or Kubernetes. In principle, they even allow a workflow to mix-and-match backends. For example, one could quickly tweak the recipe to switch from a standard WSClean cult-cargo image to a locally-built binary. This allows for rapid experimentation, but certainly does not promote reproducibility.

8.1. The slurm backend wrapper

The slurm wrapper must be combined with the native or Apptainer/Singularity backend. It can be enabled as follows:

```
opts:
 backend:
  select: singularity
  slurm:
   enable: true
```

In a slurm deployment, Stimela2 itself is executed on the head (login) node, since it is a relatively lightweight process requiring very little resources. It then wraps job invocations in slurm's `srun` command, which causes them to be executed on compute nodes, and captures the jobs' console output via its logging mechanism, while waiting for jobs to complete. This approach naturally dovetails with the scatter feature of loop recipes (Section 7.4), and makes it straightforward to deploy parallel workflows across the cluster.

The `srun` command has a wide array of options to control resource allocation (e.g. CPUs, RAM) and node placement. Instead of actively

[19] The tag selection logic is inspired by and modelled on Ansible playbooks (https://www.ansible.com).

[20] https://apptainer.org. Apptainer was formerly branded as Singularity, and Stimela2's current documentation, as well as the configuration syntax, still uses the old name in some contexts. Older, Singularity-branded versions of the engine (version 2.6+) are currently still supported. The old name will likely be deprecated in future versions of Stimela2.

[21] Docker and Podman provide similar functionality, and backends for these engines can probably be added to Stimela2 in the future, given sufficient demand.

managing these, Stimela2 takes a hands-off approach, providing a transparent mechanism for tuning `srun` on a per-cab and per-step basis. An optional `srun_opts` subsection under `backends.slurm` is mapped to `srun` options — that is, every *`key: value`* entry in that subsection is converted into *`--key value`* arguments on the `srun` command line. Since backend options are composed hierarchically (see above), this allows for a fine degree of tuning, e.g.:

```
opts:
 backend:
  select: singularity
  slurm:
   enable: true

cabs:
 wsclean:
  backend:
   slurm:
    srun_opts:
     cpus-per-task: 32
     mem: 128G

recipe:
 steps:
  foo:
   backend:
    slurm:
     srun_opts:
      cpus-per-task: 1
      mem: 16G
  bar:
   backend:
    slurm:
     srun_opts:
      cpus-per-task: 4
      mem: 32G
```

Stimela2's composability features (Section 4.2) allow for this tuning information to be placed into a separate YAML document that can be combined with the recipe itself at runtime.

8.2. The Kubernetes backend

Kubernetes, often denoted as K8s, is a highly extensible, open-source container orchestration platform designed to facilitate the deployment, scaling, and management of distributed, containerized applications. Emerging from Google, based on its experience of running containers at scale, Kubernetes has rapidly become the de facto standard in container orchestration, achieving widespread adoption in the cloud-native computing community. Kubernetes is a cross-platform standard, and provides a unified and abstracted interface to a "cluster" resource that is largely independent of the underlying implementation, the latter taking many forms and scales, e.g.:

- Elastic Kubernetes Service (EKS) provided by AWS;
- Google Kubernetes Engine (GKE) provided by Google Cloud;
- Azure Kubernetes Service;
- On-premises cluster solutions such as Rancher[22] and the MicroK8s[23] engine;
- A private "virtual cluster" on the local machine, provided by the Kubernetes In Docker (kind)[24] package.

The Stimela2 Kubernetes backend is implemented in terms of the standard K8s Python API. While the recipe itself is parsed and managed by the local Stimela2 installation, each actual step of the recipe is executed as a *pod* on a (potentially, remote) K8s cluster — while console output and logfiles are captured locally. This means that both the logical structure of the recipe, as well as the end-user look-and-feel, remain the same for both local and cloud-based[25] workflows (modulo, of course, the data itself being made available on the relevant platform.)

The Kubernetes backend tends to require a lot more configuration (see Appendix B for an example), with input from relevant system administrators. Fundamentally, this is because the K8s paradigm tends to deal in highly abstracted resources, managed via *operators*, which in turn manage the lifecycle of these resources. For example, instead of a single (local or networked) POSIX-like filesystem (as would generally be the case in a local or slurm environment), K8s deals in *volumes* and *persistent volume claims*, which are mapped onto underlying cluster resources. Stimela2 does not offer any features to simplify K8s cluster administration — the cluster must be preconfigured, and this requires specialist administrator knowledge. However, once this is in place, Stimela2's composability features (Section 4.2) allow for a clean separation of roles and responsibilities:

- The cluster administrator sets up and maintains the K8s environment, as well as access rights to the cluster. Ultimately, the users are instructed on how to configure a K8s *context* in which their jobs will be executed, and how to obtain cluster access credentials.
- The context is specified in the `opts.backend.kube` section (in fact, the context may be omitted, if the user has selected a default context in their K8s configuration file).
- Additional information from the administrator is used to configure recipe-specific resources, such as volumes and node allocations, in the `opts.backends.kube` section.
- All this configuration can be defined in a separate YAML document, which exists independently of the recipe.

In principle, a clean separation can be achieved, such that the difference between running a slurm and a K8s workflow can be as little as

```
$ stimela run recipe.yml slurm-config.yml
```

versus

```
$ stimela run recipe.yml kube-config.yml
```

8.3. Resource allocation

When running a recipe via a local backend, Stimela2 simply executes the steps one-by-one (or in parallel, if a loop scatter construct is employed), and defers to the operating system to enforce resource limits (via disk quotas, cgroups, etc.) However, an optional `backends.rlimits` subsection can be configured to set per-process limits in the Stimela2 session. This supports all the standard `RLIMIT_` symbols defined by the PSL `resource` module.[26]

When it comes to resource *requests* (as opposed to limits), e.g. the number of CPU cores to use, different packages have different conventions for specifying such. It is generally left up to the recipe developer to propagate these options to the packages appropriately. A good practice, for example, is to define a recipe-level input:

[22] https://rancher.com
[23] https://microk8s.io
[24] https://kind.sigs.k8s.io

[25] At time of writing, we have only tested this with the AWS EKS cloud implementation. However, the backend – by design – knows nothing about AWS per se, with only the standard K8s API is employed throughout. We can thus reasonably expect that other cloud architectures can be supported with minimal to no effort. The onus is entirely on the cluster administrator to configure a K8s cluster on the appopriate platform.
[26] https://docs.python.org/3/library/resource.html

```
inputs:
 ncpu:
  dtype: int
  info: max number of CPUs to use
  default: =config.run.ncpu-physical
```

...and pass this to cabs consistently. Note that the `config.run` namespace provides information on the number of physical and logical cores on the local system.

In a SLURM or KUBERNETES environment, resource management becomes somewhat more elaborate. STIMELA2 does not attempt to control this directly. Instead, the philosophy is to expose the relevant options via backend settings (mirroring the options of the underlying cluster interface as much as possible, rather than trying to invent an extra abstraction layer), and let the user tweak these on a per-cab and per-step basis using hierarchical composition. We saw an example of this with the `srun_opts` section of the SLURM backend wrapper above. In the KUBERNETES environment, the standard K8s API provides the concept of CPU and memory *requests* and *limits*, which can be controlled via the `backend.kube.job_pod.cpu` and `backend.kube.job_pod.memory` subsections. To give another example, platform-specific implementations of K8s provide fine-grained control over where a pod is scheduled (for example, on what kind of AWS EC2 instance) via a custom `nodeSelector` section in the pod spec. STIMELA2 simply exposes this section under `backend.kube.job_pod`, for the user to tune (see Appendix B).

8.4. Custom resource management

One exception to STIMELA2's hands-off resource policy is the management of DASK jobs on K8s, controlled via the DASK-KUBERNETES[27] operator. The K8s API supports the concept of *custom resources* (CRs), defined by *custom resource definitions* (CRDs). DASK-KUBERNETES uses this API to define a DASKJOB CRD, hierarchically composed of DASKCLUSTER and DASKWORKERGROUP CRDs, which include definitions for *job runner*, *scheduler* and *worker* pods. Upon creation of a DASKJOB CR, the operator creates all the dependent CRs in a hierarchical fashion, bringing up the pods. The operator then monitors the execution of the job, retrying the three pod types on failure. Upon successful completion of the job, or after a limited number of failures, the operator destroys the pods. Explicit deletion of the DASKJOB results in the operator deleting all related resources.

To support DASK-aware application such as QUARTICAL, STIMELA2 provides a `backends.kube.dask_cluster` section, where the components of the DASKJOB are defined. As with all other backend settings, this can be tweaked on a per-cab or even per-step basis. The content of the section is passed on to the DASK-KUBERNETES API. On completion (or failure) of the DASKJOB, STIMELA2 deletes it, ensuring that all related CRs are released.

A second exception is temporary disk storage, required by some applications (consider, e.g., WSCLEAN and its `-temp-dir` option). With local and SLURM backends, this is simply a temporary directory within the filesystem, so no particular management is required. In the K8s environment, a volume and a persistent volume claim must be configured, and must be cleaned up properly to avoid hogging a (potentially costly) resource. STIMELA2 provides control over this via the `backends.kube.volumes` section; temporary volumes can be configured to persist for the duration of the recipe (`lifecycle: session`) or even just the step (`lifecycle: step`).

9. Diagnosing complex workflows

As anyone familiar with developing pipelines (or even simple processing scripts) will know, two of the most common questions that arise in the process are (a) what went wrong? and (b) what took so long? STIMELA2 provides a couple of mechanisms to help provide [at least first-order] answers.

[27] https://kubernetes.dask.org/

9.1. Logfile management

For a complex workflow with many steps, a full output log, while usually extensive and chaotic, can yet be an invaluable debugging aid. STIMELA2 tries to aid this further. When running a recipe, it will intercept each cab's console (stdout and stderr) output, and send a copy to its log (as well as to its own console). It then provides a number of features to organize log files in a sensible and human-friendly way; these are invoked by assigning to the `opts.log` section of the config namespace. This can be done directly within a recipe's YAML code. Here's an example of generically useful logger settings (employing the substitutions described in Section 7.2):

```
opts:
 log:
  dir: logs/log-{config.run.datetime}
  name: log-{info.fqname}
  nest: 2
  symlink: log
```

This snippet will configure STIMELA2 to do the following:

- The logfiles for each run are stored under a unique log subdirectory named `./logs/log-`*`YYYYMMDD-HHMMSS`*`/`, where the timestamp refers to the date/time the (outer) recipe was started.
- Each step of the recipe is logged into its own separate logfile, based on its fully-qualified name (i.e., `log-`*`recipe_name`*`.`*`step_label`*`.txt`).
- The output of nested sub-recipes, if any, is logged as part of the enclosing step (this is implied by nesting level 2). Increasing the nesting level will cause nested sub-recipes to generate their own uniquely-named logfiles.
- The symlink `logs/log` is updated to point to the most recent log subdirectory.

9.2. Profiling

STIMELA2 includes some basic profiling functionality. When using a local (native or Apptainer/Singularity) backend, it collects and reports the following set of performance metrics:

- Elapsed time
- CPU use percentage
- RAM use
- System load
- Read/write operations (Gb per second)
- Read/write operations total.

These metrics are broken down by step (hierarchically, if sub-recipes are employed), as well as by peak and average, and reported to the console at the end of a run, as well as saved to a YAML file for future analysis. This allows for quick identification of both particularly slow steps, as well as resource-hogging ones.

For more detailed profiling of individual steps, it is a simple matter to modify the cab definition so as to invoke an external profiler, for example:

```
cabs:
 wsclean:
  command: valgrind --tool=callgrind wsclean
```

With the SLURM backend wrapper, STIMELA2 can (currently) only measure elapsed time.

With the K8s backend, STIMELA2 uses the K8s metrics API to collect metrics from running pods. However, since the detailed metrics provided by this API are highly platform-dependent, STIMELA2 is currently restricted to only the basic set of standard metrics:

- CPU core usage

- RAM usage
- Number of running pods
- Elapsed time (for which no API is required, of course).

The ultimate profiling metric (when running on the cloud) would be dollars per run (or dollars per MeerKAT image). In principle, Amazon's EKS engine, at least, seems to provide the necessary custom APIs to make collecting this information feasible. We have deliberately avoided incorporating any vendor-specific APIs into the Kubernetes backend, but a possible avenue for future development is adding a "plugin" capability to support such custom metrics.

10. Conclusions, discussion and future work

We believe Stimela2 has largely achieved its aim of occupying the middle ground between ease of use (linear scripting with a rich syntax), scalable workflows (provided by the slurm and Kubernetes backends), and practical repeatability (provided by containerization). It has also enabled us to run workflows in the cloud (as demonstrated by the use of AWS for some of the benchmarks in this paper series.) It is already used as the underlying platform for standalone projects (Samboco et al., 2024; Smirnov et al., 2025b). Stimela2 has enabled extremely non-standard data reductions (Smirnov et al., 2024), with the latter work providing an important example of publishing a science paper along with repeatable recipes. We would welcome and support wider adoption of Stimela2 in the community. Stimela2 is fully open source, and has a stable public release available via PyPI.

This paper series presents a new software ecosystem that is already reasonably feature-complete: it is now almost feasible to run an entire MeerKAT data reduction, from raw visibilities to final images, within a Stimela2 workflow, using exclusively Dask-MS-based packages (implying that we can dispense with a CTDS-backed measurement set and use a more efficient, parallel-I/O-enabled storage backend, throughout). At the same time, Stimela2 remains fully compatible with legacy tools, the PARROT recipes (Smirnov et al., 2024) being a case in point – these readily fall back on CASA and WSClean when needed. This development raises a number of interesting issues meriting further discussion.

10.1. Whither cloud?

Our new software ecosystem removes two of the technical bottlenecks to adopting cloud solutions in radio astronomy. Firstly, the Dask-MS data access layer allows us to replace the traditional CTDS storage backend with modern Zarr- or Apache-Arrow-based backends that can use S3 object stores, the most economical cloud storage solution which also scales linearly via requests on multiple nodes. This also opens the door to exploring interesting price/performance optimizations, since cloud providers such as AWS offer hierarchical object store solutions with progressively pricier/cheaper faster/slower storage.

Two related points should be made here:

- One does not need to be all-in on Dask-MS to start exploiting the cloud. We also use Stimela2 to run cloud workflows that intermix legacy packages such as CASA and WSClean with our new-generation tools. To support access via legacy packages, the data then necessarily has to reside in CTDS-backed measurement sets on block storage. This will continue to remain an option going forward, albeit a less cost-effective and performant one.
- The flexibility provided by the QuartiCal and pfb-imaging packages can further reduce storage costs (of selfcal-style workflows), by (a) obviating the need for separate model and corrected data columns (in a throw-back to the traditional AIPS approach, a QuartiCal to pfb-imaging selfcal loop need only store the raw data, plus per-antenna gain solutions), and (b) providing support for baseline-dependent averaging.

Secondly, Stimela2, and its Kubernetes backend in particular, shows a way to resolve the "thick-thin" problem of traditional workflows. Pods are scheduled with predefined CPU and RAM requirements. The K8s autoscaler is then able to bring the required number (and type) of virtual machine instances up and down on demand, with "thin" steps allocated to small and cheap instances, and large expensive instances only spun up when "thick" steps can make effective use of them.

We do not claim to have solved all the problems of cloud computing for radio astronomy here, and neither are we (yet) in a position to claim that the cloud is the more (or less) cost-effective solution for our needs — although the example of Rubin Observatory looms large. In particular, we have completely ignored the issue of data ingress and egress. What we can claim is that (a) our new software ecosystem does enable more economical workflows (in a cloud context) than the legacy software stack, and that (b) it does open the door to quite detailed price/performance evaluations and optimizations in the future. (As an aside, we further note that cost analysis on the cloud is a fairly simple and transparent process – it is much more intricate for on-premises compute, where power, labour costs, etc. need to be factored in.)

Finally, in a research software context, the cloud does offer an absolutely unique opportunity for algorithm evaluation. At any given point in time, any on-premises cluster (or HPC centre) is locked into a particular set of hardware configurations and architectures. If one wanted to evaluate an algorithm's performance on a different architecture (how does my solution scale to 100 thin nodes? Can I make efficient use of 16 GPUs at the same time?), these resources may simply not be available locally, or even at national HPC centres. Cloud providers, on the other hand, offer a rich variety of architectures for short-term rental. This comes with a price tag, of course — but having an option with a price tag is better than having no option at all. We therefore believe that cloud computing will have a large role to play in our community going forward, regardless of its adoption by major radio telescopes.

As an anecdotal data point to illustrate this, the total cost for this paper series (and associated software packages), in terms of AWS cloud charges, came to approximately $25,000 over two years. This included a lot of software development and testing, many abortive, misguided and simply ill-conceived experiments, and the final, often massively scaled, benchmark runs presented in the previous papers. This is comparable to the price of a single high-end compute node, without power and administration costs. While a compute node has a useful lifetime of well over two years, even a high-end node would not have accommodated some of the scaling experiments presented in this series. As another data point, we recently conducted a four-day data reduction workshop (Africalim 2024[28]) with $\sim$ 50 participants, introducing them to the Africanus software stack. Virtually all the data tutorials were run on an AWS EKS cluster, with participants spinning up multiple instances of workflows at the same time, without experiencing any resource contention. The total cloud charges for the workshop came in at under $1,000.

10.2. Wither reproducibility?

Reproducibility of scientific results remains a thorny issue in many fields, In radio astronomy, it is particularly dire. Given the complexity of our software stacks, even simply *repeating* a workflow on a different system may be difficult. This is exacerbated by the varying level of data reduction detail given in the literature; authors will often refer to calibrating data using "standard CASA practices", or mention some subset of WSClean parameters without specifying the rest of them, or the version used. Full reproducibility is an asymptotically elusive goal. Numerical algorithms are subject to round-off errors, may behave differently on different architectures, and even give subtly different results

[28] https://github.com/africalim/resources

when parallel computation results in the order of operations not being guaranteed. *Total replicability*, as defined by Pritchard and Wicenec (2024), may always remain hostage to the robustness of algorithm implementations. Even then, such replicability does not ensure true scientific reproducibility. In the era of increasingly sensitive instruments and subtle detections, verifying that a feature in the data product is not of "algorithmic" origin can only be done by recovering it via multiple alternative workflows, the Event Horizon Telescope results providing an example of this approach (Event Horizon Telescope Collaboration et al., 2019a,b).

There is then no magic software bullet for this, but at least ensuring repeatability seems like worthwhile goal. Stimela2's combination of fully containerized workflows, as well as the maintenance of versioned images on cult-cargo's `quay.io` registry, means that any given recipe should be able to be run repeatably across a range of architectures (natively, on slurm, or even Kubernetes), modulo availability of CPU and RAM resources. However, this still needs to be proven in practice. In terms of the reproducibility tenets (Pritchard and Wicenec, 2024), Stimela2 is designed to provide *scientifically replicatable* workflows.

There is also a sociological aspect to this issue. Stimela2 quite deliberately makes development workflows easy to implement: test images and locally-built binaries may be swapped in by changing one line in a YAML file. This makes it tempting to violate repeatability, in the name of quicker experimental turnaround — the onus remains firmly on the user to provide the self-discipline when it comes to publishing results that have been obtained with repeatable workflows.

In order to encourage such self-discipline, it will be worthwhile to introduce the concept of *certifiable* versus *non-certifiable* workflows. A certifiable Stimela2 workflow would be one that operates fully in containerized mode, while *exclusively* using versioned images from public registries. Any other workflow would then be non-certifiable, i.e. not guaranteed to be repeatable. The current version of Stimela2 takes some steps in this direction, by automatically generating a dependencies file listing all images and software versions (including its own) employed in the workflow. With a little more development effort, and adopting some of the ideas of Pritchard and Wicenec (2024), this can become the basis of a formal certification mechanism. Container images already provide signature hashes. These can be combined with a hash of the full YAML configuration, and perhaps even a hash of the input data, to produce a *certified workflow signature*. Anyone attempting to repeat the workflow can then, as a minimum, be guaranteed that they are starting from bit-perfect copies of the input data and software packages.

10.3. A public recipe competition

As an important signpost in this direction, we intend to shortly release a set of end-to-end Stimela2 recipes that take some (publicly available) challenging MeerKAT datasets all the way from raw visibilities to final images. The raw data will be hosted on AWS via its Open Data program[29] (and is, in any case, also available from the SARAO archive.) In the first instance, this will allow the community to test our repeatability claims.

More importantly, this will offer an interesting opportunity to *provably* compare calibration and imaging algorithms. Radio interferometry literature is full of any number of attempted "imaging competitions", as well as many papers illustrating the superior imaging performance of the authors' novel algorithm, via a comparison with best-effort images made with a competing package. When presented in paper form, it is inevitable that the impact of such comparisons is limited and often contentious. Ideally, what we would like to see instead are public releases of modified recipes where, for example, the imaging step has been swapped out for the authors' novel package. If such recipes are made publicly available, and can be run repeatably, then algorithm comparisons become far more meaningful and quantitative. We therefore intend to promote our end-to-end recipes as the start of an open *recipe* (as opposed to *image*) competition.

CRediT authorship contribution statement

O.M. Smirnov: Writing – review & editing, Writing – original draft, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization. **S. Makhathini:** Writing – original draft, Software, Methodology, Conceptualization. **J.S. Kenyon:** Writing – review & editing, Validation, Software, Investigation. **H.L. Bester:** Writing – review & editing, Writing – original draft, Validation, Software, Investigation. **S.J. Perkins:** Writing – review & editing, Validation, Software, Methodology, Investigation, Conceptualization. **A.J.T. Ramaila:** Writing – review & editing, Validation, Software. **B.V. Hugo:** Writing – review & editing, Validation, Investigation.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Oleg Smirnov reports financial support was provided by National Research Foundation. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

Funding

OMS's and JSK's research is supported by the South African Research Chairs Initiative of the Department of Science, Technology and Innovation and National Research Foundation (grant No. 81737).

SM's research is supported by a *SARAO-Funded University Research Groups* grant of the South African Radio Astronomy Observatory and the National Research Foundation (grant No. 97792). The MeerKAT telescope is operated by the South African Radio Astronomy Observatory, which is a facility of the National Research Foundation, an agency of the Department of Science, Technology and Innovation.

We would like to thank Ross Davies, Ilze van Tonder, Christopher Voges, Peter Fosseus, Nawaal Adams, Dietrich Keller and their colleagues at Silicon Overdrive, as well as Agnat Max Makgoale (Amazon Web Services) for invaluable technical and logistical assistance over the course of this work.

[29] https://aws.amazon.com/opendata

Appendix A. Basic recipe walk-through: simulating a MeerKAT observation

This example shows a recipe that will create a simulated measurement set containing visibility data products, based on a source catalogue. This simulation follows the following steps:

1. Configuration of the telescope and observing parameters, such as:
   - Observing frequency, channelization and the number of channels.
   - Observation start time, integration time and duration.
   - Phase centre.
2. Definition of the sky to be observed, and simulation of visibility data.
3. Imaging of the simulated data.

Now, let us write the corresponding STIMELA2 recipe. This simulation will use the TELSIM and SKYSIM tools from the SIMMS package,[30] as well as the WSCLEAN cab from the standard CULT-CARGO suite. This assumes that SIMMS and CULT-CARGO have been installed (e.g. via pip).

```
_include:
 - (simms.parser_config)simms-cabs.yaml
 - (cultcargo)wsclean.yaml
```

The TELSIM tool creates an observation template (or empty MS) given an array telescope configuration and observing parameters, and SKYSIM simulates a sky model into an MS. Both these tools are loaded in the `_include` section from the `simms-cabs.yaml` configuration file that comes with the `simms` package. We will also use the WSCLEAN package, for which we pull in a cab definition from the standard CULT-CARGO suite.

Next, we set the runtime options for the recipe. Here, we set the recipe to use the host's native software environment (i.e. native backend). This assumes that all the software (including the WSCLEAN binary) is installed locally. We also direct all log files into a directory `logdir` which will nest the logs for each run in a subdirectory labelled with the run's timestamp (Section 9.1). Logs from the latest run can be accessed via a shortcut (symlink) named `logs`:

```
opts:
 backend:
  select: native
 log:
  dir: logdir/logs-{config.run.datetime}
  nest: 2
  symlink: logs
```

Now that the recipe dependencies and runtime settings are done, let us write the recipe per se. We start with defining the recipe inputs (recall that `dtype` is `str` by default):

```
simulator-recipe:
 info: "Basic simulation example"
 inputs:
  prefix:
   default: example-simulation
  msname:
   implicit: "{current.prefix}.ms"
  telescope:
   default: meerkat
  direction:
   default: J2000,0deg,-30deg
  dtime:
   dtype: float
   default: 10
   info: "correlator dump time, in seconds"
  ntimes:
   dtype: int
   default: 180
   info: "number of correlator dumps"
  freq0:
   dtype: Union[float,str]
   default: 1.4GHz
   info: "reference frequency"
  dfreq:
   dtype: Union[float,str]
   default: 1MHz
   info: "channel width"
```

[30] https://github.com/wits-cfa/simms

```
  nchan:
   dtype: int
   default: 5
   info: "number of channels"
  skymodel:
   dtype: File
   required: yes
  sefd:
   dtype: float
   default: 420
   info: "telescope SEFD"
  npix:
   dtype: int
   default: 4096
   info: "image size in pixels"
  pix-size:
   default: 1.2asec
   info: "pixel size"
```

These inputs can be changed at runtime. For example, to run the simulation for the VLA telescope's C configuration instead of the default MeerKAT telescope, the command line would be

```
$ stimela run telsim.yaml telescope=vla-c
```

Now, we add the simulation outputs. These are (a) the simulated MS and (b) the imaging products. For the latter, we use aliases (Section 7.3) to directly propagate out the outputs of the imaging step:

```
 outputs:
  ms:
   dtype: MS
   implicit: =current.msname
  image:
   aliases: [image.restored.mfs]
  dirty-image:
   aliases: [image.dirty.mfs]
  resid-image:
   aliases: [image.residual.mfs]
  psf-image:
   aliases: [image.psf.mfs]
```

Finally, we add the three steps described at the beginning of this section. The first step, `makems`, uses TELSIM to create an empty MS using the defined observation settings. The settings defined in the `recipe.inputs` section can be accessed as recipe attributes, as will be shown in this step. The output of this step is the empty MS.

```
 steps:
  makems:
   cab: telsim
   params:
    ms: =recipe.msname
    telescope: =recipe.telescope
    direction: =recipe.direction
    dtime: =recipe.dtime
    ntimes: =recipe.ntimes
    startfreq: =recipe.freq0
    dfreq: =recipe.dfreq
    nchan: =recipe.nchan
```

The second step, `addsky`, uses SKYSIM to simulate the sky model into the MS. This step updates the `DATA` column of the MS with the simulated visibilities, and does not create any new files.

```
  addsky:
   cab: skysim
   params:
    ms: =steps.makems.ms
    catalogue: =recipe.skymodel
    sefd: =recipe.sefd
    column: DATA
```

The last step, `image`, uses WSCLEAN to make an image of the simulated observation.

```
  image:
   cab: wsclean
   params:
    ms: =steps.makems.ms
    prefix: =recipe.prefix
    column: =steps.addsky.column
    size: =recipe.npix
    weight: briggs 0
    niter: 10000
    scale: =recipe.pix-size
    mgain: 0.85
    auto-mask: 5
    auto-threshold: 2
```

This last step yields a number of FITS files, some of which are propagated to the recipe's outputs via the aliases defined above. The full recipe can now be invoked as

```
$ stimela run telsim.yaml
```

As noted above, any non-default recipe inputs can be added to the command line.

Appendix B. Configuring a mixed workflow including Kubernetes

The YAML code below provides a practical example of advanced Stimela2 usage, illustrating several key points.

- This code is not a standalone recipe, but is rather an augmentation of the base recipe presented in Appendix B of Bester et al. (2025). It is invoked together with the base recipe, as e.g.:

```
$ stimela run pfbimage.yaml kubeconfig.yaml image obs=esohi basedir=s3://rarg-test-binface/ESO137 \
    log-directory=/mnt/data/pfb-test/logs
```

- The purpose of the augmentation is to test and benchmark the base recipe. See comments in YAML code for more details.
- The augmentation results in most of the recipe being executed on AWS, but the first step is configured to execute locally using the Singularity/Apptainer backend.
- The augmentation also adds two additional steps at the end of the base recipe, also executed locally. Cab definitions for these steps are provided below.
- The code also contains a complete example of configuring the Kubernetes backend for an AWS EKS cluster.

Altogether, this illustrates how (rather complicated) runtime deployment logic and performance tweaks can be kept completely separate from the logical structure of the recipe.

```
opts:
 backend:
  select: kube
  # set up global kubernetes config
  kube:
   enable: true
   dir: /mnt/data/pfb-test
   namespace: rarg-test-compute
   volumes:
    rarg-test-compute-efs-pvc:
     mount: /mnt/data

   user:
    uid: 1000
    gid: 1000

   # Some predefined pod specs to be utilised in the recipe
   predefined_pod_specs:
    admin:
     nodeSelector:
      rarg/node-class: admin
    scheduler:
     nodeSelector:
      rarg/node-class: compute
      rarg/instance-type: m6i.large
      rarg/capacity-type: ON_DEMAND
    medium:
     nodeSelector:
```

```
        rarg/node-class: compute
        rarg/instance-type: c6in.8xlarge
        rarg/capacity-type: ON_DEMAND
      large:
       nodeSelector:
        rarg/node-class: compute
        rarg/instance-type: c6in.24xlarge
        rarg/capacity-type: ON_DEMAND

    # common environment variables for worker nodes
    env:
     NUMBA_CACHE_DIR: /mnt/data/pfb-test
     CONFIGURATT_CACHE_DIR: /mnt/data/pfb-test
     JAX_ENABLE_X64: 'True'
     JAX_PLATFORMS: 'cpu'
     LD_LIBRARY_PATH: '/usr/local/lib'

image:
 info:
  This file augments the stimela config in the main imaging recipe.
  It demonstrates the use of multiple backends, including the kubernetes
  backend which enables processing some of the steps on AWS instances.
 steps:
  init:
   info:
    Reads data from local storage and writes the averaged
    Stokes visibility products, potentially to an S3 bucket
    associated with an AWS account.
   backend:
    select: singularity
    singularity:
     rebuild: true
     bind_dirs:
      # These need to be mounted inside the singularity image
      # Substitute $USER for bester here
      /home/bester/numba_cache: rw
      /home/bester/.aws: rw
     env:
      NUMBA_CACHE_DIR: /home/bester/numba_cache

  grid:
   info:
    Sets up a Dask cluster that distributes gridding tasks per band.
   backend:
    select: kube
    kube:
     enable: true
     # The main application runs on the job_pod
     job_pod:
      type: medium
      memory:
       limit: "54Gi"
      cpu:
       # request more than half the available cores
       # to utilize the full instance per pod
       request: 20
     # Spawns Dask cluster with 6 worker nodes
     dask_cluster:
      enable: true
      num_workers: 6
      name: pfb-test-cluster
      threads_per_worker: 1
      worker_pod:
       type: medium
       memory:
        limit: "55Gi"
       cpu:
        request: 20
     # Environment variables can be adjusted per step
     env:
      NUMBA_NUM_THREADS: '32'
```

```
    params:
     fits-mfs: false
     fits-cubes: false
     nworkers: 6
     nthreads: 32


  sara:
    info:
     The main deconvolution application runs on a single large node.
    backend:
     select: kube
     kube:
      enable: true
      job_pod:
       type: large
       memory:
        limit: "170Gi"
       cpu:
        request: 50
      dask_cluster:
       enable: false
    params:
     # FITS files could be written to EFS but not S3.
     # They are not required for the test we perform below.
     fits-mfs: false
     fits-cubes: false
     nworkers: 1
     nthreads: 96

  pull_model:
    info:
     Transfer component model from S3 to local storage for testing
    cab: s3tolocal
    # binary command is executed on the native backend
    backend:
     select: native
    params:
     source: s3://rarg-test-data/hi_combined_bda_I_main_model.mds
     dest: /home/bester/projects/ESO137/from_aws/hi_combined_bda_I_main_model.mds

  compare_models:
    info:
     Ensure that the model produced on AWS infrastructure is compatible
     with the model produced locally up to deconvolution tolerance.
    cab: compmodels
    # python script executed on the native backend in a
    # virtual environment that has pfb-imaging installed
    backend:
     select: native
     native:
      virtual_env: ~/.venv/pfb
    params:
     model_local: /scratch/bester/hi_combined_bda_I_main_model.mds
     model_remote: =previous.dest
     epsilon: =recipe.steps.sara.tol

cabs:
 s3tolocal:
  name: s3tolocal
  info:
   Runs a binary command to transfer data from AWS S3 to local storage.
  command: aws s3 cp --recursive
  policies:
   positional: true
  inputs:
   source:
    info: The location to fetch the model from
    dtype: URI
    required: true
```

```
  outputs:
   dest:
    info: The location to place the model
    required: true
    dtype: Directory
    mkdir: true

 compmodels:
  name: compmodels
  info:
   Runs a python script to test whether two component model are
   compatible up to a tolerance specified by epsilon.
  flavour: python-code
  command: |
   import numpy as np
   import xarray as xr
   from pfb.utils.misc import model_from_mds

   model1 = model_from_mds(model_local)
   model2 = model_from_mds(model_remote)
   model1 -= model2  # diff
   model1 += 1.0    # for relative tolerance
   try:
    assert np.allclose(model1, 1.0, atol=epsilon, rtol=epsilon)
   except:
    raise ValueError("Models don't match")

   print('Success!')

  inputs:
   model_local:
    info: Model obtained by executing locally
    required: true
    dtype: str
   model_remote:
    info: Model obtained by executing remotely
    required: true
    dtype: str
   epsilon:
    info: Precision with which to compare models
    default: 1e-4
    dtype: float
```

Data availability

Public MeerKAT archive data was used.

References

Berriman, G.B., 2023. How can astronomers make the best use of cloud platforms? In: American Astronomical Society Meeting Abstracts, Vol. 241. p. 312.04.

Bester, H.L., Kenyon, J.S., Repetti, A., Perkins, S.J., Smirnov, O.M., Blecher, T., Mhiri, Y., Roth, J., Heywood, I., Wiaux, Y., Hugo, B.V., 2025. Africanus III. pfb-imaging – a flexible radio interferometric imaging suite. Astron. Comput. submitted for publication arXiv:2412.10073.

Byrne, R., Jacobs, D., 2021. Development of a high throughput cloud-based data pipeline for 21 cm cosmology. Astron. Comput. 34, 100447. doi:10.1016/j.ascom.2021.100447.

Dodson, R., Momjian, E., Pisano, D.J., Luber, N., Blue Bird, J., Rozgonyi, K., Smith, E.T., van Gorkom, J.H., Lucero, D., Hess, K.M., Yun, M., Rhee, J., van der Hulst, J.M., Vinsen, K., Meyer, M., Fernandez, X., Gim, H.B., Popping, A., Wilcots, E., 2022. CHILES. VII. Deep Imaging for the CHILES Project, an SKA Prototype. AJ 163 (2), 59. doi:10.3847/1538-3881/ac3e65, arXiv:2112.06488.

Dodson, R., Vinsen, K., Wu, C., Popping, A., Meyer, M., Wicenec, A., Quinn, P., van Gorkom, J., Momjian, E., 2016. Imaging SKA-scale data in three different computing environments. Astron. Comput. 14, 8–22. doi:10.1016/j.ascom.2015.10.007, arXiv:1511.00401.

Event Horizon Telescope Collaboration, Akiyama, K., Alberdi, A., Alef, W., Asada, K., Azulay, R., Baczko, A.-K., Ball, D., Baloković, M., Barrett, J., Bintley, D., Blackburn, L., Boland, W., Bouman, K.L., Bower, G.C., Bremer, M., Brinkerink, C.D., Brissenden, R., Britzen, S., Broderick, A.E., Broguiere, D., Bronzwaer, T., Byun, D.-Y., Carlstrom, J.E., Chael, A., Chan, C.-k., Chatterjee, S., Chatterjee, K., Chen, M.-T., Chen, Y., Cho, I., Christian, P., Conway, J.E., Cordes, J.M., Crew, G.B., Cui, Y., Davelaar, J., De Laurentis, M., Deane, R., Dempsey, J., Desvignes, G., Dexter, J., Doeleman, S.S., Eatough, R.P., Falcke, H., Fish, V.L., Fomalont, E., Fraga-Encinas, R., Friberg, P., Fromm, C.M., Gómez, J.L., Galison, P., Gammie, C.F., García, R., Gentaz, O., Georgiev, B., Goddi, C., Gold, R., Gu, M., Gurwell, M., Hada, K., Hecht, M.H., Hesper, R., Ho, L.C., Ho, P., Honma, M., Huang, C.-W.L., Huang, L., Hughes, D.H., Ikeda, S., Inoue, M., Issaoun, S., James, D.J., Jannuzi, B.T., Janssen, M., Jeter, B., Jiang, W., Johnson, M.D., Jorstad, S., Jung, T., Karami, M., Karuppusamy, R., Kawashima, T., Keating, G.K., Kettenis, M., Kim, J.-Y., Kim, J., Kim, J., Kino, M., Koay, J.Y., Koch, P.M., Koyama, S., Kramer, M., Kramer, C., Krichbaum, T.P., Kuo, C.-Y., Lauer, T.R., Lee, S.-S., Li, Y.-R., Li, Z., Lindqvist, M., Liu, K., Liuzzo, E., Lo, W.-P., Lobanov, A.P., Loinard, L., Lonsdale, C., Lu, R.-S., MacDonald, N.R., Mao, J., Markoff, S., Marrone, D.P., Marscher, A.P., Martí-Vidal, I., Matsushita, S., Matthews, L.D., Medeiros, L., Menten, K.M., Mizuno, Y., Mizuno, I., Moran, J.M., Moriyama, K., Moscibrodzka, M., Müller, C., Nagai, H., Nagar, N.M., Nakamura, M., Narayan, R., Narayanan, G., Natarajan, I., Neri, R., Ni, C., Noutsos, A., Okino, H., Olivares, H., Ortiz-León, G.N., Oyama, T., Özel, F., Palumbo, D.C.M., Patel, N., Pen, U.-L., Pesce, D.W., Piétu, V., Plambeck, R., PopStefanija, A., Porth, O., Prather, B., Preciado-López, J.A., Psaltis, D., Pu, H.-Y., Ramakrishnan, V., Rao, R., Rawlings, M.G., Raymond, A.W., Rezzolla, L., Ripperda, B., Roelofs, F., Rogers, A., Ros, E., Rose, M., Roshanineshat, A., Rottmann, H., Roy, A.L., Ruszczyk, C., Ryan, B.R., Rygl, K.L.J., Sánchez, S., Sánchez-Arguelles, D., Sasada, M., Savolainen, T., Schloerb, F.P., Schuster, K.-F., Shao, L., Shen, Z., Small, D., Sohn, B.W., SooHoo, J., Tazaki, F., Tiede, P., Tilanus, R.P.J., Titus, M., Toma, K., Torne, P., Trent, T., Trippe, S., Tsuda, S.,

van Bemmel, I., van Langevelde, H.J., van Rossum, D.R., Wagner, J., Wardle, J., Weintroub, J., Wex, N., Wharton, R., Wielgus, M., Wong, G.N., Wu, Q., Young, A., Young, K., Younsi, Z., 2019a. First M87 event horizon telescope results. III. Data processing and calibration. ApJL 875 (1), L3. doi:10.3847/2041-8213/ab0c57, arXiv:1906.11240.

Event Horizon Telescope Collaboration, Akiyama, K., Alberdi, A., Alef, W., Asada, K., Azulay, R., Baczko, A.-K., Ball, D., Baloković, M., Barrett, J., Bintley, D., Blackburn, L., Boland, W., Bouman, K.L., Bower, G.C., Bremer, M., Brinkerink, C.D., Brissenden, R., Britzen, S., Broderick, A.E., Broguiere, D., Bronzwaer, T., Byun, D.-Y., Carlstrom, J.E., Chael, A., Chan, C.-k., Chatterjee, S., Chatterjee, K., Chen, M.-T., Chen, Y., Cho, I., Christian, P., Conway, J.E., Cordes, J.M., Crew, G.B., Cui, Y., Davelaar, J., De Laurentis, M., Deane, R., Dempsey, J., Desvignes, G., Dexter, J., Doeleman, S.S., Eatough, R.P., Falcke, H., Fish, V.L., Fomalont, E., Fraga-Encinas, R., Freeman, W.T., Friberg, P., Fromm, C.M., Gómez, J.L., Galison, P., Gammie, C.F., García, R., Gentaz, O., Georgiev, B., Goddi, C., Gold, R., Gu, M., Gurwell, M., Hada, K., Hecht, M.H., Hesper, R., Ho, L.C., Ho, P., Honma, M., Huang, C.-W.L., Huang, L., Hughes, D.H., Ikeda, S., Inoue, M., Issaoun, S., James, D.J., Jannuzi, B.T., Janssen, M., Jeter, B., Jiang, W., Johnson, M.D., Jorstad, S., Jung, T., Karami, M., Karuppusamy, R., Kawashima, T., Keating, G.K., Kettenis, M., Kim, J.-Y., Kim, J., Kim, J., Kino, M., Koay, J.Y., Koch, P.M., Koyama, S., Kramer, M., Kramer, C., Krichbaum, T.P., Kuo, C.-Y., Lauer, T.R., Lee, S.-S., Li, Y.-R., Li, Z., Lindqvist, M., Liu, K., Liuzzo, E., Lo, W.-P., Lobanov, A.P., Loinard, L., Lonsdale, C., Lu, R.-S., MacDonald, N.R., Mao, J., Markoff, S., Marrone, D.P., Marscher, A.P., Martí-Vidal, I., Matsushita, S., Matthews, L.D., Medeiros, L., Menten, K.M., Mizuno, Y., Mizuno, I., Moran, J.M., Moriyama, K., Moscibrodzka, M., Müller, C., Nagai, H., Nagar, N.M., Nakamura, M., Narayan, R., Narayanan, G., Natarajan, I., Neri, R., Ni, C., Noutsos, A., Okino, H., Olivares, H., Oyama, T., Özel, F., Palumbo, D.C.M., Patel, N., Pen, U.-L., Pesce, D.W., Piétu, V., Plambeck, R., PopStefanija, A., Porth, O., Prather, B., Preciado-López, J.A., Psaltis, D., Pu, H.-Y., Ramakrishnan, V., Rao, R., Rawlings, M.G., Raymond, A.W., Rezzolla, L., Ripperda, B., Roelofs, F., Rogers, A., Ros, E., Rose, M., Roshanineshat, A., Rottmann, H., Roy, A.L., Ruszczyk, C., Ryan, B.R., Rygl, K.L.J., Sánchez, S., Sánchez-Arguelles, D., Sasada, M., Savolainen, T., Schloerb, F.P., Schuster, K.-F., Shao, L., Shen, Z., Small, D., Sohn, B.W., SooHoo, J., Tazaki, F., Tiede, P., Tilanus, R.P.J., Titus, M., Toma, K., Torne, P., Trent, T., Trippe, S., Tsuda, S., van Bemmel, I., van Langevelde, H.J., van Rossum, D.R., Wagner, J., Wardle, J., Weintroub, J., Wex, N., Wharton, R., Wielgus, M., Wong, G.N., Wu, Q., Young, A., Young, K., Younsi, Z., 2019b. First M87 event horizon telescope results. IV. Imaging the central supermassive black hole. ApJL 875 (1), L4. doi:10.3847/2041-8213/ab0e85, arXiv:1906.11241.

Heywood, I., 2020. oxkat: Semi-automated imaging of MeerKAT observations. Astrophysics Source Code Library, record ascl:2009.003.

Józsa, G.I.G., Andati, L.A.L., de Blok, W.J.G., Hugo, B.V., Kleiner, D., Kamphuis, P., Molnár, D.C., Makhathini, S., Maccagni, F.M., Perkins, S.J., Ramaila, A., Ramatsoku, M., Serra, P., Smirnov, O.M., Thorat, K., White, S.V., 2022. CARACal - The Containerized Automated Radio Astronomy Calibration Pipeline. In: Ruiz, J.E., Pierfedereci, F., Teuben, P. (Eds.), Astronomical Society of the Pacific Conference Series, Vol. 532. 532, p. 447.

Kenyon, J.S., Perkins, S.J., Bester, H.L., Smirnov, O.M., Russeeawon, C., Hugo, B.V., 2025. Africanus II. QuartiCal: calibrating radio interferometer data at scale using Numba and Dask. Astron. Comput. submitted for publication, arXiv:2412.10072.

Kenyon, J.S., Smirnov, O.M., Grobler, T.L., Perkins, S.J., 2018. CUBICAL - fast radio interferometric calibration suite exploiting complex optimization. MNRAS 478 (2), 2399–2415. doi:10.1093/mnras/sty1221, arXiv:1805.03410.

Makhathini, S., 2018. Advanced Radio Interferometric Simulation and Data Reduction Techniques (Ph.D. thesis). Rhodes University; Faculty of Science, Physics and Electronics.

Mei, Y., Wei, S., Wang, F., Wu, C., Tobar, R., Shaikh, M., Deng, H., Dai, W., Liang, B., Wicenec, A., 2022. An empirical evaluation on the applicability of the DALiuGE execution framework. Astron. Comput. 38, 100541. doi:10.1016/j.ascom.2021.100541, arXiv:2112.13088.

Molenaar, G.J., 2021. Design Patterns and Software Techniques for Large-Scale, Open and Reproducible Data Reduction (Ph.D. thesis). Rhodes University; Faculty of Science, Physics and Electronics.

Offringa, A.R., McKinley, B., Hurley-Walker, N., Briggs, F.H., Wayth, R.B., Kaplan, D.L., Bell, M.E., Feng, L., Neben, A.R., Hughes, J.D., Rhee, J., Murphy, T., Bhat, N.D.R., Bernardi, G., Bowman, J.D., Cappallo, R.J., Corey, B.E., Deshpande, A.A., Emrich, D., Ewall-Wice, A., Gaensler, B.M., Goeke, R., Greenhill, L.J., Hazelton, B.J., Hindson, L., Johnston-Hollitt, M., Jacobs, D.C., Kasper, J.C., Kratzenberg, E., Lenc, E., Lonsdale, C.J., Lynch, M.J., McWhirter, S.R., Mitchell, D.A., Morales, M.F., Morgan, E., Kudryavtseva, N., Oberoi, D., Ord, S.M., Pindor, B., Procopio, P., Prabu, T., Riding, J., Roshi, D.A., Shankar, N.U., Srivani, K.S., Subrahmanyan, R., Tingay, S.J., Waterson, M., Webster, R.L., Whitney, A.R., Williams, A., Williams, C.L., 2014. WSCLEAN: an implementation of a fast, generic wide-field imager for radio astronomy. MNRAS 444 (1), 606–619. doi:10.1093/mnras/stu1368, arXiv:1407.1943.

O'Mullane, W., Guy, L., Dubois-Felsmann, G., Economou, F., AlSayyad, Y., Graham, M., Slater, C., 2023. Vera C. Rubin Observatory: Open Science to the core. In: American Astronomical Society Meeting Abstracts, Vol. 242. p. 116.04.

Perkins, S.J., Kenyon, J.S., Andati, L.A.L., Bester, H.L., Smirnov, O.M., Hugo, B.V., 2025. Africanus I. Scalable, distributed and efficient radio data processing with Dask-MS and Codex Africanus. Astron. Comput. submitted for publication arXiv:2412.12052.

Pritchard, N.J., Wicenec, A., 2024. Formal definition and implementation of reproducibility tenets for computational workflows. arXiv:2406.01146.

Sabater, J., Sánchez-Expósito, S., Best, P., Garrido, J., Verdes-Montenegro, L., Lezzi, D., 2017. Calibration of LOFAR data on the cloud. Astron. Comput. 19, 75–89. doi:10.1016/j.ascom.2017.04.001.

Samboco, V.G., Heywood, I., Smirnov, O., 2024. SolarKAT: Solar imaging pipeline for MeerKAT. Astrophysics Source Code Library, record ascl:2401.013.

Smirnov, O.M., Golden, A., Myburgh, T., Ngcebetsha, B., Tasse, C., Heywood, I., Ramaila, A.J.T., Thompson, M.A., Kenyon, J.S., Perkins, S.J., Dawson, J., Bester, H.L., Bright, J.S., Oozeer, N., Samboco, V.G.G., Sihlangu, I., Choza, C., 2025a. Mining the time axis with TRON - II. MeerKAT detects a stellar radio flare from a distant RS CVn candidate. MNRAS 538 (1), L89–L93. doi:10.1093/mnrasl/slaf015, arXiv:2501.09489.

Smirnov, O.M., Heywood, I., Geyer, M., Myburgh, T., Tasse, C., Kenyon, J.S., Perkins, S.J., Dawson, J., Bester, H.L., Bright, J.S., Ngcebetsha, B., Oozeer, N., Samboco, V.G.G., Sihlangu, I., Choza, C., Siemion, A.P.V., 2025b. Mining the time axis with TRON - I. Millisecond pulsars in Omega Centauri, Terzan 5, and 47 Tucanae detected through MeerKAT interferometric imaging. MNRAS 538 (1), L62–L68. doi:10.1093/mnrasl/slaf009, arXiv:2501.09488.

Smirnov, O.M., Stappers, B.W., Tasse, C., Bester, H.L., Bignall, H., Walker, M.A., Caleb, M., Rajwade, K.M., Buchner, S., Woudt, P., Ivchenko, M., Roth, L., Noordam, J.E., Camilo, F., 2024. The RATT PARROT: serendipitous discovery of a peculiarly scintillating pulsar in MeerKAT imaging observations of the Great Saturn - Jupiter Conjunction of 2020. I. Dynamic imaging and data analysis. MNRAS 528 (4), 6517–6537. doi:10.1093/mnras/stae303.

Tasse, C., Hugo, B., Mirmont, M., Smirnov, O., Atemkeng, M., Bester, L., Hardcastle, M.J., Lakhoo, R., Perkins, S., Shimwell, T., 2018. Faceting for direction-dependent spectral deconvolution. Astron. Astrophys. 611, A87. doi:10.1051/0004-6361/201731474, arXiv:1712.02078.

Toomey, L., Benn, D., Chapman, J., Dai, S., Dempsey, J., Hobbs, G., Russell, C., Wang, C., Wang, J., Zic, J., 2017. Processing Public Pulsar Astronomy Data in the Amazon Cloud. Technical Report, CSIRO.

van Diepen, G., 2015. Casacore table data system and its use in the MeasurementSet. Astron. Comput. 2, doi:10.1016/j.ascom.2015.06.002.

Wang, R., Tobar, R., Dolensky, M., An, T., Wicenec, A., Wu, C., Dulwich, F., Podhorszki, N., Anantharaj, V., Suchyta, E., Lao, B., Klasky, S., 2020a. Processing full-scale square kilometre array data on the summit supercomputer. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12. doi:10.1109/SC41405.2020.00006.

Wang, R., Tobar, R., Dolensky, M., An, T., Wicenec, A., Wu, C., Dulwich, F., Podhorszki, N., Anantharaj, V., Suchyta, E., Lao, B., Klasky, S., 2020b. Processing full-scale square kilometre array data on the summit supercomputer. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12. doi:10.1109/SC41405.2020.00006.

Wu, C., Tobar, R., Vinsen, K., Wicenec, A., Pallot, D., Lao, B., Wang, R., An, T., Boulton, M., Cooper, I., Dodson, R., Dolensky, M., Mei, Y., Wang, F., 2017. DALiuGE: A graph execution framework for harnessing the astronomical data deluge. Astron. Comput. 20, 1–15. doi:10.1016/j.ascom.2017.03.007, arXiv:1702.07617.