Astronomy and Computing 41 (2022) 100648

Contents lists available at ScienceDirect

Astronomy and Computing

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

Full length article

Rosetta: A container-centric science platform for resource-intensive, interactive data analysis

S.A. Russo [a,c,*], S. Bertocco [a], C. Gheller [b], G. Taffoni [a]

[a] INAF (Italian National Institute for Astrophysics) – Astronomical Observatory of Trieste, Italy
[b] INAF (Italian National Institute for Astrophysics) – Institute of Radioastronomy, Bologna, Italy
[c] Department of Mathematics and Geosciences, University of Trieste, Italy

ARTICLE INFO

Article history:
Received 2 May 2022
Accepted 26 August 2022
Available online 14 September 2022

Keywords:
Science platforms
Data analysis
Reproducibility
Software containers
Big data
HPC

ABSTRACT

Rosetta is a science platform for resource-intensive, interactive data analysis which runs user tasks as software containers. It is built on top of a novel architecture based on framing user tasks as microservices – independent and self-contained units – which allows to fully support custom and user-defined software packages, libraries and environments. These include complete remote desktop and GUI applications, besides common analysis environments as the Jupyter Notebooks. Rosetta relies on Open Container Initiative containers, which allow for safe, effective and reproducible code execution; can use a number of container engines and runtimes; and seamlessly supports several workload management systems, thus enabling containerized workloads on a wide range of computing resources. Although developed in the astronomy and astrophysics space, Rosetta can virtually support any science and technology domain where resource-intensive, interactive data analysis is required.

© 2022 Published by Elsevier B.V.

1. Introduction

Data volumes are rapidly increasing in several research fields, as in bioinformatics, particle physics, earth sciences, and more. Next generation sequencing technologies, new particle detectors, recent advances in remote sensing techniques and higher resolutions in general, on both the instrumental and the simulation side, are constantly setting new challenges for data storage, processing and analysis.

Astrophysics is no different, and the upcoming generation of surveys and scientific instruments as the Square Kilometer Array (SKA) (Dewdney et al., 2009), the Cherenkov Telescope Array (CTA) (Acharya et al., 2013), the Extremely Large Telescope (ELT) (De Zeeuw et al., 2014), the James Webb Space telescope (Gardner et al., 2006), the Euclid satellite (Laureijs et al., 2012) and the eROSITA All-Sky Survey (Merloni et al., 2012) will pile up on this trend, bringing the data volumes in the exabyte-scale. Moreover, numerical simulations, a theoretical counterpart capable of reproducing the formation and evolution of the cosmic structures of the Universe, must reach both larger volumes and higher resolutions to cope with the large amount of data produced by current and upcoming surveys. State of the art cosmological N-body hydrodynamic codes (as OpenGADGET, GADGET4 Springel et al., 2020 and RAMSES Bleuler and Teyssier, 2014) can generate up to 20 petabytes of data out of a single simulation run, which are required to be further post-processed and compared with observational data (Springel et al., 2018; Ragagnin et al., 2017; Taffoni et al., 2019; Habib et al., 2016).

The size and complexity of these new experiments (both observational and numerical) require therefore considerable storage and computing resources for their data to be processed and analyzed, and possibly to adopt new approaches and architectures. High Performance Computing (HPC) systems including Graphical Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs), together with the so called "bring computing close to the data" paradigm are thus becoming key players in obtaining new scientific results (Asch et al., 2018), not only by reducing the time-to-solution, but also by becoming the sole approach capable of processing datasets of the expected size and complexity.

In particular, even the last steps of the data analysis processes, which could be usually performed on researchers' workstations and laptops, are getting too resource-intensive and are progressively required to be offloaded to such systems as well.

Although capable of satisfying the necessary computing and storage requirements, these systems are usually hosted in remote, dedicated computing centers and often managed with queue systems, in order to dynamically share their resources across different users and to optimize the workload and the throughput. This can strongly complicate the user interaction, requiring remote connections for shell and graphical access (as SSH and X

* Corresponding author at: INAF (Italian National Institute for Astrophysics) – Astronomical Observatory of Trieste, Italy.
E-mail address: stefano.russo@inaf.it (S.A. Russo).

https://doi.org/10.1016/j.ascom.2022.100648
2213-1337/© 2022 Published by Elsevier B.V.

protocol forwarding), careful data transfer and management, and scheduler-based access to computing resources which strongly limits interactive access to the system. Bringing along the software required for the analysis can be even more challenging, and without proper setup (in particular with respect to its dependencies) it can not only fail to start or even compile, but also severe reproducibility issues can arise (Bhandari Neupane et al., 2019).

To address these challenges, we see an increasing effort in developing the so called *science platforms* (Taffoni et al., 2020; Desai et al., 2019; Cui et al., 2020; Taghizadeh-Popp et al., 2020). A science platform (SP) is an environment designed to offer users a smoother experience when interacting with remote computing and storage resources, in order to mitigate some of the issues outlined above.

In science and, more specifically, in astronomy, a number of SPs have been designed and developed over the past years.

CERN SWAN (Piparo et al., 2018) represents CERN's effort to build towards the science platform paradigm. SWAN is a service for interactive, web-based data analysis which makes Jupyter Notebooks widely available on CERN computing infrastructure together with a Dropbox-like solution for data management. However, as of today, this solution does not provide support for applications other than the Jupyter Notebooks and a built-in shell terminal, does not allow using custom or graphical software environments and requires heavy system-level integration in order to be used on top of existent computing resources.

ESA Datalabs (Arviset et al., 2021) is a science platform specific to astronomy and astrophysics. Similarly to CERN SWAN, it allows users to work on ESA's computing infrastructure using interactive computing environments as Jupyter Lab and Octave (or to choose from pre-packaged applications as TOPCAT). Datalabs is mainly focused on enabling users to gain direct access to ESA's datasets, it does not support using custom software environments, and it is not an open source project.

The Large Synoptic Survey Telescope (LSST) developed a similar science platform (Jurić et al., 2017), based on a set of integrated web applications and services through which the scientific community will be able to "access, visualize, subset and analyze LSST data". The platform vision document does not mention applications other than the Jupyter Notebooks, nor support for custom or graphical software environments, and refers to its own computing architecture.

There are also a number of initiatives entirely focussing on supporting Jupyter Notebooks on cloud and HPC infrastructures (such as Nicklas et al., 2018; Mendez et al., 2019; Milligan, 2018; Castronova et al., 2018), which might fall in our SP definition to some extent, and in particular in Astronomy and Astrophysics it is worth to mention SciServer (Taghizadeh-Popp et al., 2020), Jovial (Araya et al., 2018) and CADC Arcade (Major et al., 2019).

Lastly, it has to be noted that the private sector is moving fast with respect to resource-intensive and interactive data analysis, mainly driven by the recent advances in artificial intelligence and machine learning. In this context, we want to cite Google Colab (Bisong, 2019) and Kaggle Notebooks (Kaggle, 2018), which are built around heavily customized versions of the Jupyter Notebooks, and Azure Machine Learning (Chappell, 2015), which provides a nearly full-optional SP specifically targeted at machine learning workflows.

While on one hand all of the above mentioned SPs do make it easier to access and use remote computing resources, on the other, since they are mainly focused on web-based and integrated analysis environments built on top Jupyter Notebooks or similar software, they also introduce two main drawbacks:

1. users are restricted in using pre-defined software packages, libraries and environments, which besides constraining their work can also lead to reproducibility issues, and
2. graphical software environments as remote desktops and GUI applications are supported only to a limited extent, if not completely unsupported.

Moreover, the deployment options for most of the SPs developed today rely on technologies originating from the IT industry (e.g. Kubernetes) and require deep integration at system-level, which is often hard to achieve in the framework of HPC clusters and data-intensive system. This is not only because of technological factors and legacy aspects, but also because of a generalized pushback for exogenous technologies from some parts of the HPC community (Crafts, 1990; Gorda, 2021; CERFACS COOP-Algo Team, 2021; Dursi, 2021).

In this paper we present a science platform which aims at overcoming these limitations: *Rosetta*. Built on top of a novel architecture based on framing user tasks as microservices – independent and self-contained units – Rosetta allows to fully support custom software packages, libraries and environments, including remote desktops and GUI applications, besides standard web-based analysis environments as the Jupyter Notebooks. Its user tasks are implemented as software containers (SUSE, 2022), which allow for safe, effective and reproducible code execution (Boettiger, 2015), and that in turn allows users to add and use their own software containers on the platform.

Rosetta is also designed with real-world deployment scenarios in mind, and thus to easily integrate with existing computing and storage resources including HPC clusters and data-intensive systems, even when they do not natively support containerization.

Although astronomy remains its mainstay (Rosetta has been developed in the framework of the EU funded project ESCAPE[1]), Rosetta can virtually support any science and technology domain.

This paper is organized as follows. In Sections 2–4, we discuss the architecture of the Rosetta platform, its implementation and the security aspects. This is followed, in Section 5, by an overview of the platform from a user prospective. Next, we present the deployment and usage scenario in a real production environment and a few use cases we are supporting (Section 6), leaving the last section to conclusions and future work.

2. Architecture

Rosetta's architecture is entirely designed to provide simplified access to remote and possibly dynamically allocated computing and storage resources, without restricting the users to a set of pre-defined software packages, libraries and environments. It unfolds in two main components: the *platform architecture* and the *task orchestration architecture*.

The platform architecture follows a standard approach where a set of services implement the various functionalities, and it is schematized in Fig. 1. These comprise a web application service for the main application logic and the web-based UI, a database service for storing internal data and a proxy service for securing the connections. The web application service functionalities can be further grouped in modules which are responsible for managing the software containers, interacting with the computing and storage resources, orchestrating the user tasks, handling the user authentication and so on.

In particular:

- *Software* functionalities allow to track the software containers available on the platform, their settings and container registries[2];

[1] ESCAPE aims to address the open science challenges shared by SKA, CTA, KM3Net, EST, ELT, HL-LHC, FAIR as well as other pan-European research infrastructures as CERN, ESO, JIVE in astronomy and particle physics.

[2] A container registry is a place where container images are stored, which can be public or private, and deployed both on premises or in the Cloud. Many container registries can co-exist at the same time.

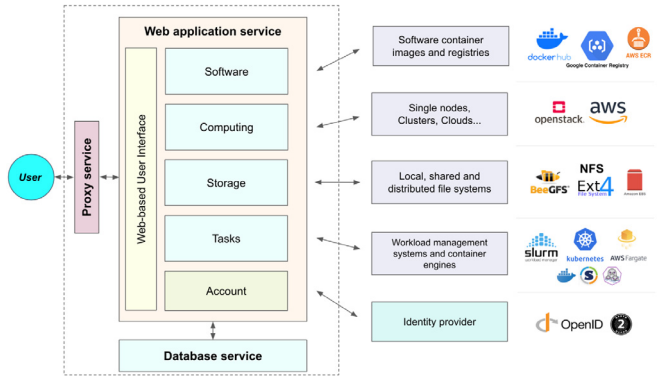


Fig. 1. Rosetta main architecture. The first level of abstraction consist in the proxy, database and web application services. The web application service is further break down into its main components (software, computing, storage, tasks and account) together with their real world counterparts, some examples of which are given in the right part of the figure.

- *Computing* functionalities allow to interact with both standalone and clustered computing resources, hosted either on premises (e.g. via Openstack) or on cloud systems (e.g. on Amazon AWS);
- *Storage* functionalities allow browsing and operating on local and shared file system (as Ext4, NFS, BeeGFS);
- *Task* functionalities allow submitting and stopping tasks as well as viewing their logs, by interacting with the computing resources workload management systems (WMSs) as Slurm and Kubernetes and/or their container engines (e.g. Docker, Singularity, Podman);
- *Account* functionalities provide user account and profile management features including user registration, login and logout, supporting both local and external authentication (e.g. OpenID Connect, Shibbolet).

Rosetta's task orchestration architecture follows instead a novel, microservice-oriented architecture (Russo et al., 2021) based on software containers. Microservices (Newman, 2015) are independent, self-contained and self-consistent units that perform a given task, which can range from a simple functionality (e.g. serving a file to download) to complex computer programs (e.g. classifying images using a neural network). They are interacted with using a specific interface, usually a REST API over HTTP, which is exposed on a given port. Microservices fit naturally in the containerization approach, where each microservice runs in its own container, isolated from the underlying operating system, network, and storage layers. User tasks in Rosetta are thus always executed as software containers, and treated as microservices. Rosetta can therefore stay agnostic with respect to the task interface, some examples of which include a Jupyter Notebook server, a web-based remote desktop or a virtual network computing (VNC) server, but also a secure shell (SSH) server with X protocol forwarding is a perfectly viable choice.

One of the main features of this approach, where user tasks are completely decoupled from the platform, is to make it possible for the users to add their own software containers. There is indeed no difference between "platform" and "user" containers, as long as they behave as a microservice. Rosetta users can thus upload their own software containers on a container registry, add them in the platform by setting up a few parameters (as the container image and the interface port), and then use them for their tasks.

In order to make use of this architecture for user tasks orchestration, Rosetta needs to be able to submit to the computing resources a container for execution, and to know how to reach it (i.e. on which IP address). These functionalities are standard and

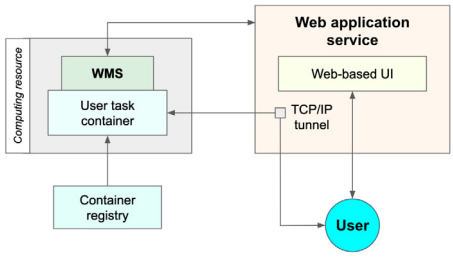


Fig. 2. Rosetta user task orchestration using the computing resource's WMS and a direct connection to the task interface through a TCP/IP tunnel.

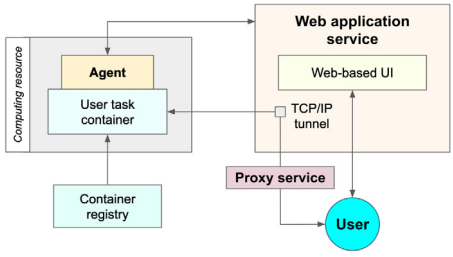


Fig. 3. Rosetta user task orchestration using the agent and the proxy service on top of a TCP/IP tunnel for connecting to the task interface.

built-in in most modern container orchestrators (e.g Kubernetes), however as mentioned in the introduction Rosetta has been designed to also support computing resources not natively supporting containerized workloads (e.g. HPC clusters and data-intensive systems). On these computing resources, also depending on the WMS and container engine used, some key features might not be available, as full container-host filesystem isolation, network virtualization and TCP/IP traffic routing between the containers. To work around these missing features, Rosetta relies on an *agent*, which is a small software component in charge of helping to manage the task container life cycle. Its main features comprises setting up the environment for the container execution, managing dynamic port allocation, reporting the host IP address to the platform, and running the container itself. The agent internal logic is described more in detail in Section 3.4.

When a container is started, its interface has to be made accessible by the user. This is achieved first by making the interface port reachable on the internal network between the computing resource and Rosetta, and then by exposing it to the outside world through Rosetta itself, thus making it accessible by the user. The first step can make use of simple TCP/IP tunnels as well as more sophisticated techniques usually available in modern container orchestrators and WMSs, while the second one can be accomplished either by directly exposing the task interface as-is or by relaying on a proxy service, which also allows to enforce access control and connection encryption.

Once tasks are executed and their interfaces made accessible, no further operations are required, and the users can be looped in. A diagram of this flow is presented with two examples: the first using a WMS supporting containerized workloads with direct connection to the task interface (Fig. 2), the second using the agent to run the task container and relaying on the proxy for connecting to the task interface (Fig. 3).

3. Implementation

Rosetta is entirely built using open-source technologies, in particular Python and the Django web framework, and released as an open source project.[3] Other technologies include HTML

[3] https://www.ict.inaf.it/gitlab/exact/Rosetta

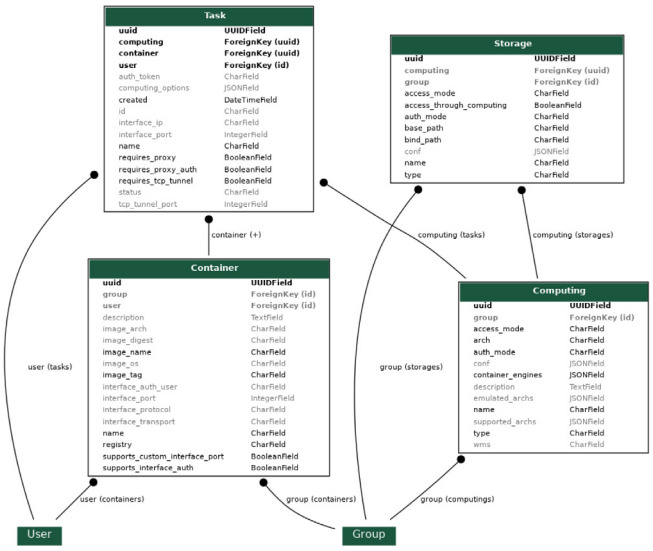


Fig. 4. The Rosetta Django ORM schema, showing the various models and their relationships. Some minor and less relevant models as the user profile, the login tokens and the key pairs have been excluded for the sake of simplicity.

and JavaScript for the UI, Postgres for the database[4] and Apache for the proxy. The platform services (not to be confused with the user tasks software containers) are containerized using the Docker engine and using Docker Compose as the default orchestrator.[5] Besides the web application, database and proxy services, Rosetta includes an optional container registry service, which can be used to store software containers locally, and a test Slurm cluster service for testing and debugging. Rosetta deployment tools provide a set of management scripts to build, bootstrap and operate the platform and a logging system capable of handling both user-generated and system errors, exceptions and stack traces.

The web application functionalities are handled with a combination of Django object–relational mapping (ORM) models and standard Python functions and classes. The ORM schema, which represents how the ORM models are actually stored in the database, is summarized in Fig. 4. In the following subsections we will describe their implementation according to the grouping introduced in Section 2: Software, Computing, Storage, Tasks and Account.

3.1. Software

Software lives in Rosetta only as software containers. Software containers are represented using a Django ORM model which acts as a twin of the "real" container, providing metadata about the container itself. Rosetta relies on Open Container Initiative (OCI) containers, which must be stored on an OCI-compliant container registry.

The `Container` ORM model has a `name` and a `description` fields to represent the container on Rosetta, and a series of attributes to identify its image: the `registry` (to set on which container registry it is hosted), the `image_name` (to locate it on the registry) and the `image_tag` (to set a specific container version).

The `image_arch`, `image_os` and `image_digest` attributes provide instead more fine-grained control in order to uniquely identify the image, and should be used in production environments. A container image is indeed uniquely identified on an OCI registry only if using, besides its name, either a triplet of tag, architecture and OS or an image hash digest (usually generated with SHA-256). This is because on OCI registries, multiple images can be assigned to the same tag, in order to enable multi-OS and multi-architecture support. Moreover, it has also to be noted that while a tag can be re-assigned, a digest is an immutable identifier and ensures reproducibility.

Containers can be registered in Rosetta as platform containers or user containers. A platform container is not associated with a specific user and thus available for all of them, while a user container belongs to and is accessible by a specif user only, according to its `user` attribute. Containers can also be shared within (and made accessible only to) a specific `group`.

An `interface_port` attribute lets Rosetta know on which port the container will expose its interface, and the `interface_protocol` sets the corresponding protocol (e.g. HTTP, SSH, VNC etc.). The `interface_transport` (defaulted to TCP/IP) can be used to cover non-standard scenarios (e.g. if using UDP).

Since as explained in Section 2, the container interfaces are made accessible to the outside world, they need to be secured. For this to happen, Rosetta allows to set up a one-time password or token at task creation-time to be used for accessing the task interface afterwards. Task interfaces can get password-protected in two ways: by implementing a password-based authentication at task-level, or by delegating it to the proxy service. In the first case, the container must be built to support this feature and must be registered on the platform with the extra `supports_interface_auth` attribute set to `True`. Rosetta can then forward the password or token to the container via an environment variable. Instead, if the container makes use of an HTTP-based interface, it can delegate its access control to the proxy service, and just expose a plain, unprotected interface over HTTP. In this case, Rosetta will set up the proxy service in order enforce user authentication when accessing the task interface, and encrypt it using SSL. Delegating the task authentication to the proxy service is the default method for HTTP-based interfaces, since it is far more secure than leaving the authentication to be implemented at task-level, as it will be discussed in Section 4.

In order to support container engines missing port mapping capabilities, Rosetta provides a mechanism to let containers receive instructions on which port to start their interface on. As already mentioned in Section 2, while most industry-standard container engines can autonomously manage TCP port mapping between containers and their host to avoid conflicts with ports already allocated (either by another service, by another container or by another instance of the same container), some of them cannot (e.g. Singularity). In this case, the Rosetta agent can provide a specific port to the container where to make its interface to listen on, which is chosen between the free ephemeral ports of the host and passed to the container via an environment variable. To let Rosetta (and the agent) know that a given container supports this mechanism, its extra attribute `supports_custom_interface_port` must be set to `True` (and the `interface_port` attribute is then discarded).

Rosetta comes with a number of a base containers for GUI applications, generic remote desktops and Jupyter Notebooks which can be easily extended to suit several needs:

- JupyterNotebook, the official Jupyter Notebook container extended to support custom interface ports;
- GUIApplication, a container built to run a single GUI application with no desktop environment;
- MinimalDesktop, a desktop environment based on Fluxbox where more than one application can be run in parallel;

[4] The database service can be replaced by any other database supported by Django.

[5] Other orchestrators can be supported as well, e.g. Kubernetes.

- BasicDesktop, a desktop environment based on Xfce for tasks requiring common desktop features as a file manager and a terminal.

The GUIApplication and Desktop containers make use of KasmVNC, a web-based VNC client built on top of modified versions of TigerVNC and NoVNC which provides seamless clipboard sharing between the remote application or desktop and the user's local desktop environment, as well as supporting dynamic resolution changes in order to always fit the web browser window, that are essential features in the everyday use.

3.2. Computing

Computing resources are divided in two main types: *standalone* and *clusters*. The first ones may or may not have a WMS in place, while the second ones always do. If a computing resource has no WMS, the task execution is synchronous, otherwise the execution is asynchronous and the tasks are queued.

The Django ORM model class used to represent computing resources is named `Computing`, and it includes a `type`, a `name` and a `description` fields for identifying a specific computing resource within Rosetta. A set of attributes describe how to access it and to submit user tasks: the `access_mode` specifies how the computing resource is accessed (i.e. over SSH, using a command line interface (CLI), or a set of APIs); the `auth_mode` specifies how the platform gets authorized on the computing resource; the `wms` specifies the WMS in place (or if there is none) and the `container_engine` specifies which container engines (and runtimes) are available. With respect to the `container_engine`, if the WMS natively supports containerized workloads and there is no need of running tasks using a specific container engine or runtime, then it can be just set to the value "`internal`".

Some example combinations of these attributes are reported in Table 1, where each row corresponds to a physical computing resource. The first row represents a classic HPC cluster using Slurm as WMS and Singularity as container engine, and requiring an accredited cluster user to submit tasks over SSH using the Slurm command line interface. The second row represents the same cluster but supporting, besides Singularity, also the Docker engine with both runC and Kata runtimes, in order to allow Rosetta (or its users) to chose the best one for a given task. The third row represents yet the same cluster but accessed over Slurm REST APIs using JSON web tokens (JWT) for authentication. The fourth and fifth rows represent instead standalone computing resources, using the Docker container engine, and accessed using SSH as a standard user for the fourth and the Docker REST APIs with a platform certificate for the fifth. The sixth, seventh and eight rows all use computing resources managed with Kubernetes, and in the eight row the container runtimes available within Kubernetes are explicitly stated. The last row is instead an example using Fargate, a hosted container execution service from Amazon Web Services (AWS) built on top of their Elastic Container Service (ECS), and accessed using its proprietary APIs.

When deploying Rosetta on a commercial cloud infrastructure as AWS or Google Cloud Platform (GCP), there are two options. The first one is to treat such infrastructures as transparent, and simply use standard (i.e. not proprietary) access modes as SSH, Slurm, or Kubernetes. In this case there is no difference between using Rosetta with computing resources deployed on premises or on such commercial cloud systems. The second option is to instead integrate at a deeper level, using AWS or GCP proprietary APIs and/or clients to automatically start new virtual machines upon request, or to use some of their native scheduling systems, as the last example of Table 1.

The implementation work to support all of the combinations of access and authentication modes, container engines and WMSs is still ongoing, as we privileged SSH and Slurm since they fit well in the application scenarios we encountered so far. However, we wanted to lie down a general framework in order to easily expand the platform in future.

The `Computing` model describes the computing resource architectures as well, and in particular the `arch` attribute defines the native architecture (e.g. amd64, arm64/v8), the `supported_archs` attribute lists extra supported architectures (e.g. 386 on amd64 architectures) and the `emulated_archs` attribute lists the architectures that can be emulated.

Computing resources can be also assigned to a specific group of users, using the `group` attribute which, if set, restricts access to the group members only, and the `conf` attribute can be used to store some computing resource-specific configurations (e.g. the host of the computing resource). Lastly, the `Computing` ORM model implements an additional `manager` property which provides common functionalities for accessing and operating on the real computing resource, as submitting and stopping tasks, viewing their logs, and executing generic commands. This property is implemented as a Python function which upon invocation instantiates and returns an object sub-classing the `ComputingManager` class, based on the computing resource `type`, `access_mode`, `auth_mode` and `wms` attributes.

Computing resources which are accessed using SSH can be accessed both as a standard user (using its account on the computing resource) or using a superuser (e.g. a "platform" user), depending on the deployment requirements.

In order to access using a standard user on the computing resource, Rosetta generates a dedicated private/public key pair, the public key of which is required to be added on the computing resource account by the user. To instead access using a "platform" superuser (and thus using the same user for orchestrating all of the user tasks), a dedicated account and key pairs are required to be set up both on the computing resource and within Rosetta.

Accessing computing resources using SSH requires no integration with the existent infrastructure at all, provided that standard SSH is available and a container engine is installed. For this reason, it perfectly fits our requirement of operating on HPC clusters and data-intensive systems where more complex integrations are hard to achieve.

3.3. Storage

Storage functionalities provide a way of defining, mounting and browsing data storages. A `Storage` is defined by a set of attributes, which include a `name`, a `type`, an `auth_mode` and an `access_mode`. If a storage is attached to a computing resource, then the `computing` attribute can be set. In this case, if the storage and the computing resource share the same access mode, the `access_through_computing` option can be ticked so that Rosetta can just use the computing resource one. The `group` attribute, if set, specifies the set of users authorized to access the storage. The `base_path` attribute sets the internal path to the storage, and supports using two variables: the `\$USER`, which is substituted with the Rosetta internal user name, and the `\$SSH_USER`, which is substituted with the SSH username (if the access method is based on SSH). The `bind_path` sets instead where the storage is made accessible within the software containers. If a data storage is attached to a computing resource and its `bind_path` is set, it will be then made accessible from all of the containers running on that computing resource, under the location specified by the `bind_path`.

For example, a storage mounted on the `/data` mount point of an SSH-based computing resource (and represented in Rosetta using `generic_posix` as type and `SSH+CLI` as access method) could have a `base_path` set to `/data/users/\$USER` and a

Table 1
Examples of various combinations of computing resource attributes. In order to schedule containerized workloads on a given computing resource, Rosetta needs to know how to access it (`access_mode`), how to get authorized (`auth_mode`), if and what WMS to use (`wms`), and which container engines are available (`container_engines`), possibly with their runtimes.

	`Access_mode`	`Auth_mode`	`Wms`	`Container_engines`
Computing resource #1	SSH+CLI	user keys	Slurm	Singularity
Computing resource #2	SSH+CLI	user keys	Slurm	Docker[runC,Kata],Singularity
Computing resource #3	API	JWT	Slurm	Docker,Singularity
Computing resource #4	SSH+CLI	user keys	none	Docker
Computing resource #5	API	platform cert.	none	Docker
Computing resource #6	CLI	platform cert.	Kubernetes	internal
Computing resource #7	SSH+CLI	platform keys	Kubernetes	internal
Computing resource #8	API	platform cert.	Kubernetes	internal[runC,Kata]
Computing resource #9	API	platform cert.	Fargate	internal

`bind_path` set to `/storages/user_data`, in order to separate data belonging to different users at orchestration-level.

At the moment only POSIX file systems are supported, which must be mounted on the various computing resources and that are in turn exposed inside the containers using the standard binding mechanism offered by most container engines. Any filesystem that can be mounted as such (e.g using FUSE) is therefore automatically supported, as CephFS or Amazon S3.

We envision adding support for other storage types in future releases, as for example object storages, but in this case accessing the storage APIs is up to the application running inside the container, and Rosetta can only act as a file manager. How to provide access in a standardized way to non-POSIX file systems within containers is indeed still an open theme.

Storage functionalities also include a set of APIs to provide support for the file manager embedded in the Rosetta web-based UI, which is built on top of the Rich File Manager[6] open source project. These APIs implement common functionalities (as get, put, dir, rename etc.) to perform file management operations, the internal logic of which depends on the storage type, making it easy to expand them in the future.

3.4. Tasks

Tasks are represented using an ORM model and a set of states (*queued*, *running* or *stopped*). Tasks running on computing resources without a WMS are directly created in the running state, while when a WMS is in place they are created in the queued state and set as running only when they get executed. States are stored in the `state` attribute of the `Task` model, which also includes a `name` and the links with the software container and the computing resource executing the task, plus its options (the `container`, `computing` and `computing_options` attributes, respectively). A set of other attributes as the `interface_ip`, `interface_port`, `tcp_tunnel_port` and `auth_token` let Rosetta know how to instantiate the connection to the task (i.e. for setting up the tunnel and/or configuring the proxy service).

Once a task starts on a computing resource, its IP address and port are saved in the corresponding `Task` fields, and the task is marked as running. If the task was queued, an email is sent to the user with a link to the task, which is particularly useful to let users immediately know when their tasks are ready, thus preventing to waste computing time on shared systems. Task functionalities also include opening the TCP/IP tunnel to the task interface port and/or configuring the HTTP proxy service in order to provide access to the task interface.

One of the main components of the task management functionalities is the agent, which as introduced in Section 2 allows to seamlessly support both WMSs not natively supporting containerized workloads and container engines missing some key features. In other words, it makes all of the computing resources behave in the same way from a Rosetta prospective. The agent is implemented as a Python script which is served by the Rosetta web application and that can run both as a superuser and as a standard, unprivileged user. When it is required, Rosetta delivers a bootstrap script on the computing resource which pulls and executes the agent code. As soon as it gets executed, the agent calls back the Rosetta web application and communicates the IP address of its host. If the agent landed on a computing resource using a container engine missing the dynamic port mapping feature, then it also searches for an available ephemeral TCP/IP port and communicates it to the web application as well. Lastly, the agent sets up the environment for the user task container, and starts it.

3.5. Account

Account and profile functionalities provide support for both local and external authentication services (e.g. Open ID connect). The accounts linking between local and external identities is based on the user email address, which is the standard approach in this kind of services.

Local and external authentication can co-exist at the same time, provided that if a user originally singed up using an external authentication service it will be then always required to log-in using that service. If allowing to register as local users or to entirely rely on external authentication is up to the administrators, and can be configured in the web application service.

Rosetta provides both user-based and group-based authorization, so that computing and storage resources, as well as software containers, can be made available to specific users or subsets of users only.

The user profile also supports some user-based configuration parameters for accessing the computing resources (e.g. the computing resource username if using an SSH-based access mode with user keys). Other minor functionalities, as password recovery, login tokens and time zone settings are provided as well.

4. Security

Security of computing systems and web applications is a wide chapter and an extensive discussion on the topic is beyond the scope of this article, however we wanted to mention the main issues and how we took them into account.

The first layer of security in Rosetta consists in using software containers for the user tasks. The base executable unit in Rosetta is indeed the container itself, meaning that users have no control outside of their containers at all: once a container is sent for execution and Rosetta handles all the orchestration, the user is dropped inside it and cannot escape.

For this reason, even if a container gets compromised, all the other ones as well as the underlying host system do not get affected. However, this statement is true in the measure of

[6] https://github.com/psolom/RichFilemanager

which the container engine can guarantee isolation and prevent privilege escalation. The Docker engine has an intrinsic issue with this respect, as it makes use of a daemon running with superuser privileges. Podman, a nearly drop-in replacement for Docker, runs instead in user-space and prevents this kind of issues by design, as well as Singularity. Other container engines as gVisor and Kata push security even further, providing respectively kernel and hardware virtualization.

Moreover, when Rosetta is integrated on computing resources using SSH-based access, the administrators can opt for revoking direct SSH user access on them, leaving Rosetta – and its containerized tasks – the only access point, thus greatly improving overall security.

With respect to potential malicious software, the first line of defense usually takes place in the container registry. Docker Hub, for example, has a built-in security scanning system, and there are a number of free and open source scanners that can be used for on-premise container registries as Klar/Clair.[7]

Scanning for malicious software can also be done when executing task containers,[8] but not all container engines support this feature. Allowing only containers coming from registries which run security scanning, or to implement these checks along the building pipeline could be the best approach to protect against malicious software in container images (Brady et al., 2020).

For what concerns software packages that can be installed at runtime inside the containers, Rosetta does not do any checking as it would be technically very hard if not even impossible. This is a common issue when giving users the freedom to download and execute code, including on commercial platforms as Google Colab and Kaggle. Even restricting user permissions would not prevent such issue, given that these packages can be always just downloaded and executed from a different location (e.g. a temporary folder). Having users to download and execute malicious software by mistake is therefore something very hard to prevent, and that has no simple mitigation approach unless relying on classic antivirus software which should run inside the containers.

As introduced in Section 2, since Rosetta user task interfaces are made accessible to the outside world, they are required to be secured, both in terms of access control and connection encryption. With this respect, it is necessary to make a distinction between HTTP-based and generic task interfaces. HTTP-based task interfaces can rely on the authentication and SSL encryption provided by the proxy service, and can therefore just use a plain HTTP protocol. Generic task interfaces (e.g. a VNC or X server) are instead required to be secured at task-level, and it is responsibility of the task container to enforce it. As explained in Section 3.1, access control setup is in this case achieved by forwarding to the task a one-time password set by the user at task creation-time, which is then to be used by the container interface to authenticate the user. Encryption has to be set up at task-level too, and can be provided in first instance using self-signed certificates, or implementing more complex solutions as dynamic certificates provisioning.

An important detail in the task security context is that Rosetta makes a strong distinction between standard and power users, through a status switch in their profile. By default, only the latter can set up custom software containers using generic task interface protocols other than the HTTP, since handling security at task level (which is always required in this case) is error-prone and must be treated carefully. Standard users can therefore add and use custom software containers for their tasks on the platform only if using an HTTP-based interface, which is in turn forced to be secured by the proxy service.

For what concerns the tunnel from the web application service to the tasks, this is protocol-agnostic (above the TCP/IP transport layer) and is either accomplished by a direct connection on a private and dedicated network (e.g. if using Kubernets) or using an SSH-based TCP/IP tunnel using users' public/private keys, as explained in Section 2, and thus assumed safe.

In terms of web security, we considered potential security risks originating from cross-site request forgery (CSRF), cross-origin resource sharing (CORS), cross-site scripting (XSS), and similar attacks. The same origin policy (SOP) of modern web browsers is already a strong mitigation for these attacks, and all the platform web pages and APIs (with a few exceptions for internal functionalities) uses Django's built-in CSRF token protection mechanism. However, the SOP policy has limitations (Schwenk et al., 2017; Chen et al., 2018), in particular in our scenario where users can run custom (and possibly malicious) JavaScript code from within the platform, either using the Jupyter Notebooks or by other means (e.g. by setting up a task serving a web page).

We therefore focused on isolating user tasks from the rest of the platform even on the web browser side. Using the same domain for both the platform and the user tasks (e.g. https://rosetta.platform/tasks/1 is indeed definitely not a viable solution as it does not allow to enforce the SOP policy at all). Also using dedicated subdomains (e.g. https://task1.rosetta.platform) has several issues, in particular involving the use of cookies (Zalewski, 2012, 2009; Squarcina et al., 2021).

The secure-by-design, safe solution is to serve user tasks from a *separate* domain (e.g. rosetta-tasks.platform). Then, each task can have its own subdomain (as https://task1.rosetta-tasks.platform) and stay separated form the main platform domain. However, handling and securing subdomains like this requires wildcard DNS services and SSL certificates, which for many institutional domains are not available (JupyterHub, 2016), including ours. For this reason, in Rosetta we opted for an intermediate solution: we serve user tasks from a separate domain (e.g. rosetta-tasks.platform) assigning each of them to a different port, under the same SSL certificate. In this way, the URL to reach the task number 1 at https://rosetta-tasks.platform:7001 can be secured by the same SSL certificate covering the URL for task number 2 at https://rosetta-tasks.platform:7002, but are treated as different origins by web browsers. SSL certificates are indeed port-agnostic, while the SOP (which basically involves the triplet protocol, host and port for defining the origin) it is not, thus enabling web browsers to enforce it between the task 1 and 2, and in general securing all of the users tasks against each others. While this approach might lead to some issues with institutional firewalls blocking external port access beyond the standard 80 and 443 ports, we found it to be the right compromise in our environment. Moreover, switching to serving each task from its own subdomain is just a matter of a quick change in the Rosetta proxy service configuration.

5. User experience

From a user prospective, Rosetta presents itself as a web application with a web-based user interface (UI) that is shown upon user login in Fig. 5. The UI, following the architecture presented in Section 2, is organized in five main areas: the *Software* section, where to browse for the software containers available on the platform or to add custom ones; the *Computing* section, where to view the available computing resources; the *Storage* section, which provides a file manager for the various data storages; the *Tasks* dashboard, where to manage and interact with the user tasks, including connecting with them and viewing their logs; and the *Account* pages, where to configure or modify user credentials and access keys.

[7] https://github.com/optiopay/klar

[8] https://docs.docker.com/engine/scan/

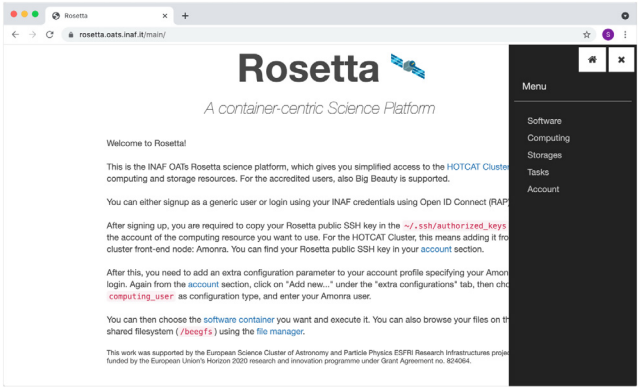


Fig. 5. The Rosetta science platform main page and menu, which provides access to its core functionalities.

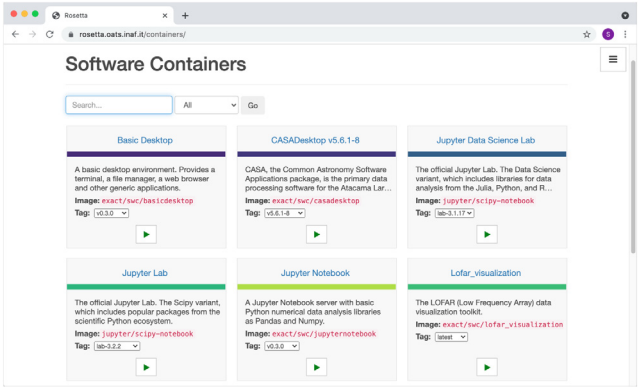


Fig. 6. Software containers list. For each entry, a brief description is provided, together with the container image name and a menu from which to select a specific version. The "play" button will start a new task with the given software.

To run a typical analysis task, the user first accesses the Software section (Fig. 6) in order to choose (or add) the desired software container. If adding a new software container, the user has to set its registry, image name, tag, the container interface port and protocol, plus some optional advanced attributes (Fig. 7). The new container will then be listed together with the other ones so that the can be chosen for execution.

Once the software container is chosen, the user hits the "play" button to create a new task. The platform will then ask the user on which computing resource to run the task, and to set a task name. A one-time password token is also generated, which is usually automatically handled by Rosetta and not required to be entered manually when connecting to the task (Fig. 8). For some computing resources, extra options as the queue or partition name, CPU cores and memory requirements can be set as well. The task is then created and submitted.

As soon as the task is starting up on the computing resource, a "connect" button in the task dashboard becomes active. At this point, the user can connect to the task with just one click: Rosetta will automatically handle all the tunneling required to reach the task on the computing resource where it is running, and drop the user inside it (Figs. 9 and 10).

Users can transfer files to and from the data storages (and thus the tasks) using the built-in file manager (Fig. 11), which is an effective solution for light datasets, analysis scripts, plots and results. Larger datasets are instead supposed to be already located on a storage, either because the data repository is located on the storage itself (in a logic of bringing the computing close to the data) or because they have been previously staged using an external procedures.

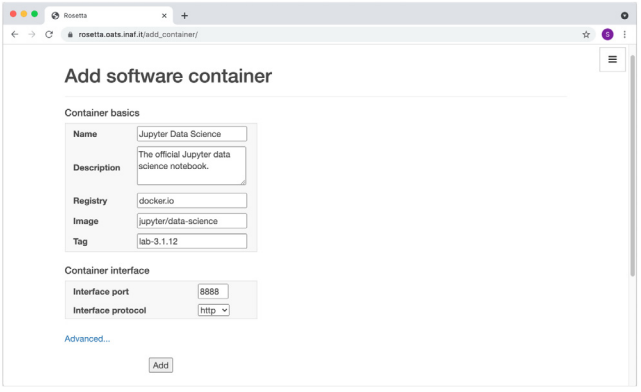


Fig. 7. Adding a software container. Besides a name and a description, key fields are the container registry, image and tag, plus the interface port and protocol.

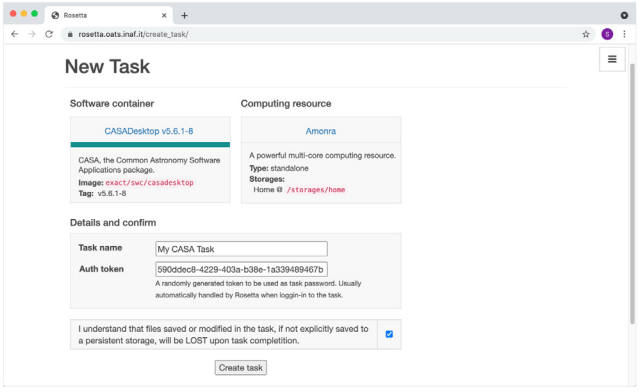


Fig. 8. Last step of new task creation, after selecting the software container and a computing resource: the interface asks the user to enter a task name and possibly other task parameters as the required number of CPUs, memory, or queue name.

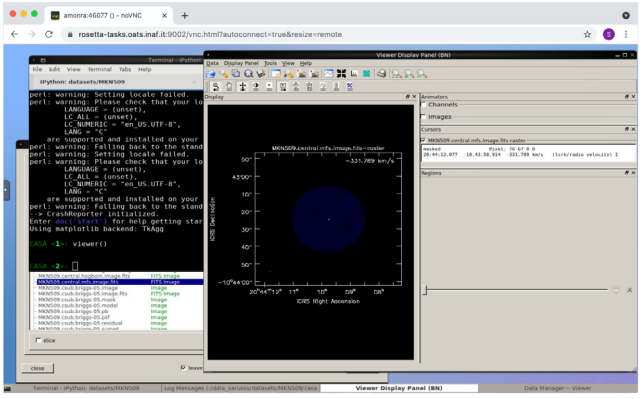


Fig. 9. A Rosetta user task running a GUI application from the CASA suite, in a remote desktop environment. The remote desktop server is web-based, and supports dynamic resolution changes and seamless clipboards sharing with the client, allowing for a smooth user experience.

6. Deployment and use cases

Rosetta is deployed in production at the computing center of INAF - Osservatorio Astronomico di Trieste (Bertocco et al., 2020), using an external, aggregated authentication system named RAP (Tinarelli et al., 2020) and serving a set of different users with different software requirements.

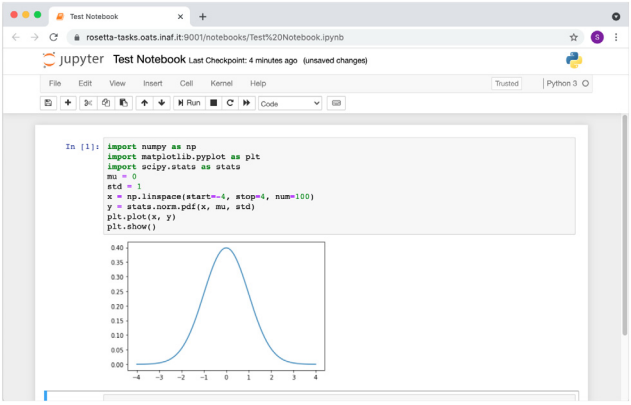


Fig. 10. A Rosetta user tasks running a Jupyter Notebook, displaying a plot using Numpy and Matplotlib. The authentication for the Notebook server is handled by the Rosetta proxy service, which also secures the connection over SSL.

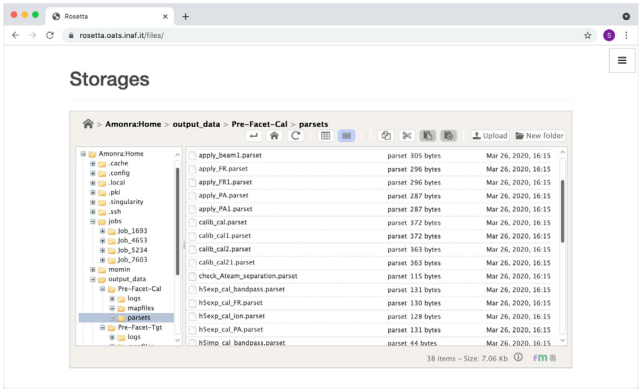


Fig. 11. The Rosetta built-in file manager, which allows for browsing data storages and to upload or download data files. While not suitable for large datasets, it is an effective tool for lighter ones as well as analysis scripts, plots and results.

To support our user community, we offer a pre-defined portfolio of containerized applications that span from generic data analysis and exploration tools (as iPython, R and Julia) to specific Astronomy and Astrophysics codes. These include common astronomical data reduction software and pipelines as IRAF, CASA, DS9, Astropy, but also Cosmological simulation visualization and analysis tools, and project-specific applications and codes. All of them are listed in the Software section of Rosetta and are accessible from the users' web browsers by running a task instance.

In the following we discuss more in detail four different use cases among the various projects we support: *the LOFAR pipelines*, *the SKA data challenges*, *the Astrocook quasar spectral analysis software*, and *the HPC FPGA bitstream design*.

6.1. The LOFAR pipelines

The software collection for the LOFAR community consists in a set of tools and pipelines used to process LOFAR data, as the Prefactor and DDFacet data reduction codes (Tasse et al., 2018), for which we created a set of software containers.

A typical run of the LOFAR data processing pipelines holds for several days, and requires significant computing resources (in terms of RAM, CPUs and Storage) to process terabytes of data ($\sim$ 15TB). Several checks are necessary during a pipeline run to verify the status of the data processing and the convergence of the results.

In this context, we are using Rosetta to run the pipelines within a software container that provides both the pipelines themselves and visual tools to check the status of the processing phase. Rosetta tasks run on an HPC cluster managed using the Slurm WMS, which allocates a set of resources in terms of RAM and CPUs as requested by the scientists in the task creation phase. These tasks compete with other standard Slurm jobs running on the cluster, thus ensuring an optimized allocation of the available resources among all users.

Scientist running the pipelines in this mode are not required to interact with the Slurm WMS or to manually deploy any software on the cluster, instead they can just rely on Rosetta and update the containers with new software if necessary.

The container source codes are available online as part of the LOFAR Italian collaboration[9] and once built are registered to an INAF private container registry in order to account for both public and private codes as required by the different LOFAR Key Projects collaborations.

6.2. The SKA data challenges

INAF participated in the SKA Data Challenges[10] as infrastructure provider. The purpose of these challenges is to allow the scientific community to get familiar with the data that SKA will produce, and to optimize their analyses for extracting scientific results from them.

The participants in the second SKA Data Challenge analyzed a simulated dataset of 1 TB in size, in order to find and characterize the neutral hydrogen content of galaxies across a sky area of 20 square degrees. To process and visualize such a large dataset, it was necessary to use at least 512 GB of RAM, and INAF offered a computing infrastructure where such resources were available.

We used Rosetta to provide simplified access to this computing infrastructure (an HPC cluster managed using the Slurm WMS) and, as for the LOFAR pipelines use case, we provided a software container that provided all of the tools and applications necessary to complete the challenge (as CASA, CARTA, WSClean, Astropy and Sofia) in a desktop environment.

Most notably, users were able to ask for specific computing resource requirements when starting their analysis tasks (512 GB of RAM, in this case), and the cluster parallel file system used to store the dataset provided high I/O performance ($>$ 4 GB/s) and plenty of disk space, so that users could focus on the scientific aspects of the challenge and not worry about orchestration and performance issues.

6.3. The astrocook quasar spectral analysis software

Astrocook (Cupani et al., 2020) is a quasar spectral analysis software built with the aim of providing many built-in recipes to process a spectrum. While this software is not necessarily resource-intensive in general, it can require quite relevant computing power in order to apply the various recipes.

Astroccok comes as a GUI application with some common and less common Python dependencies which are sometimes hard to install (as Astropy, StatsModels and wxPython) and it is a great example about how to use Rosetta in order to provide one-click access to a GUI application which might require some extra computing power.

Fig. 12 shows Astrocook running in a Rosetta task on a mid-sized, standalone computing resource, and accessed using the web-based remote desktop interface.

[9] https://www.ict.inaf.it/gitlab/lofarit/containers
[10] https://sdc2.astronomers.skatelescope.org/sdc2-challenge

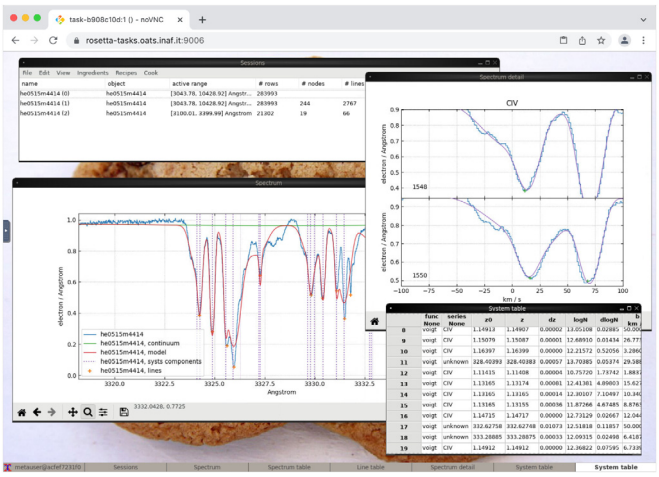


Fig. 12. The Astrocook quasar spectral analysis software running in a Rosetta task on a mid-sized computing resource. The "Spectrum" and "Spectrum detail" windows are the main components of Astrocook, while the "Sessions" and "System table" windows recap the analysis steps and parameters.

6.4. The HPC FPGA bitstream design

Field Programmable Gate Arrays (FPGAs) can be used as accelerators in the context of physics simulations and scientific computing and they have been adopted as a low-energy acceleration devices for exascale testbeds.

One of these testbeds is ExaNeSt's (European Exascale System Interconnect and Storage) prototype (Katevenis et al., 2018), a liquid-cooled cluster composed of proprietary Quad-FPGA daughterboard computing nodes, interconnected with a custom network and equipped with a BeeGFS parallel filesystem.

To use this cluster it is necessary to re-engineer codes and algorithms (Taffoni et al., 2019, 2020; Goz et al., 2020): the substantial programming efforts required to program FPGAs using the standard approach based on Hardware Description Languages (HDLs), together with its subsequent weak code portability have long been the main challenges in using FPGA-enabled HPC clusters as the ExaNeSt's prototype.

However, thanks to the High Level Synthesis (HLS) approach, FPGAs can be programmed using high level languages, thus highly reducing the programming effort and greatly improving portability. HLS tools use high level input languages as C, C++, OpenCL and SystemC which, after a process involving intermediate analysis, logic synthesis and algorithmic optimization, are translated into FPGA-compatible code as the so called "bitstream" files.

This last step in particular requires a considerable amount of resources: 128 GB of RAM, extensive multi-threading support and 100 GB of hard disk space are the requirements for creating the bitstream files for the above mentioned FPGA-enabled HPC cluster. Moreover, from a user prospective, the design of an FPGA bitstream requires the interactive use of several GUI applications (as nearly all the HLS tools) and to let the software work for several hours.

Rosetta was adopted as the primary tool for programming INAF's FPGA cluster prototype, and suited very well the use case. Thanks to enabling access to persistent, web-based remote desktops with the required computing and storage resources, users were indeed capable of using HLS tools from their standard computing equipment, and to let them work for as many hours as needed, even if disconnecting and reconnecting the day after.

7. Discussion

In designing and implementing Rosetta we faced two main challenges: supporting custom software packages, libraries and environments, and integrating with computing resources not natively supporting containerized workloads.

We addressed the first challenge by developing a novel architecture based on framing user tasks as microservices. This allowed Rosetta to fully support custom software packages, libraries and environments (including GUI applications and remote desktops) and together with software containers allowed to ensure safe, consistent and reproducible code execution across different computing resources.

With respect to the second challenge, it has first to be noted that HPC clusters and data-intensive systems still rely on Linux users for a number of reasons, including accounting purposes and local permission management. This means that most of the containerization solutions born in the IT industry, which assume to operate as a superuser, are in general not suitable. For this reason, the Singularity container engine was built to operate exclusively at user-level, and quickly become the standard in the HPC space.

However, Singularity is not designed to provide full isolation between the host system and the containers, and by default directories as the home folder, `/tmp`, `/proc`, `/sys`, and `/dev` are all shared with the host, environment variables are exported as they are set on host, the PID namespace is not created from scratch, and the network and sockets are as well shared with the host. Also, the temporary file system provided by Singularity in order to make the container file system writable (which is required for some software) is a relatively weak solution, since it is stored in memory and often with a default size of 16MB, thus very easy to fill up.

We therefore had to address all these issues before being able to use Singularity as a container engine from Rosetta. In particular, we used a combination of command line flags (`-cleanenv`, `-containall`, `-pid`) and ad-hoc runtime sandboxing for the key directories which require write access (as the user home), orchestrated by the agent. This step was key for the success of our approach and proved to remove nearly all the issues related to running Singularity containers on different computing systems.

Similarly, we had to work around a series of features lacking in WMSs not natively supporting containerized workloads (as Slurm), including container life cycle management itself, network virtualization and TCP/IP traffic routing between the containers, all solved using the agent as explained in the previous sections.

Once we were able to ensure a standardized behavior of container engines and WMSs, we were able to make task execution uniform across different kinds of computing resources, providing the very same user experience. In this sense, Rosetta can be considered as an umbrella for a variety of computing resources, and can act as a sort of bridge in the transition towards software containers.

8. Conclusions and future work

We presented Rosetta, a science platform for resource-intensive, interactive data analysis which runs user tasks as software containers. Its main characteristic lies in providing simplified access to remote computing and storage resources without restricting users to a set of pre-defined software packages, libraries and environments.

To achieve this goal, we developed a novel architecture based on framing user tasks as microservices – independent and self-contained units – which we implemented as software containers. This approach allowed us to fully support custom software packages, libraries and environments, including remote desktops and

GUI applications besides standard web-based solutions as the Jupyter Notebooks. Moreover, adopting software containers allowed for safe, effective and reproducible code execution, and enabled us to let our users to add and use their own software containers on the platform.

We also took real-world deployment scenarios in mind, and designed Rosetta to easily integrate with existent computing resources, even where they lacked native support for containerized workloads. This proved to be particularly helpful for integrating with HPC clusters and data-intensive systems.

We successfully tested Rosetta for a number of use cases, including the LOFAR data reduction pipelines at INAF computing centers in the context of the ESCAPE project which funded this work, the SKA data challenges, and other minor use cases of our user community.

The benefits of seamlessly offloading data analysis tasks to a sort of "virtual workstation", hosted on a computing system capable of providing CPUs, RAM and storage resources as per requests were immediately clear, removing constrains and speeding up the various activities.

Although astronomy and astrophysics remains its mainstay, Rosetta can virtually support any science and technology domain requiring resource-intensive, interactive data analysis, and it is currently being tested and evaluated in other institutions.

Future work include adding support for distributed workloads (e.g. MPI, Ray) and for computing resources with mixed architectures, developing a command line interface, integrating with data staging solutions and continuing the implementation efforts for integrating with current and new WMSs (e.g. Torque, Openshift, Rancher, Nomad, and more).

CRediT authorship contribution statement

S.A. Russo: Conception and design of study, Acquisition of data, Analysis and/or interpretation of data, Writing – original draft. **S. Bertocco:** Writing – review & editing. **C. Gheller:** Writing – review & editing. **G. Taffoni:** Conception and design of study, Writing – original draft.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the European Science Cluster of Astronomy and Particle Physics ESFRI Research Infrastructures project, funded by the European Union's Horizon 2020 research and innovation programme under Grant Agreement no. 824064. We also acknowledge the computing center of INAF- Osservatorio Astronomico di Trieste, (Bertocco et al., 2020; Taffoni et al., 2020), for the availability of computing resources and support. All authors read and approved the final manuscript.

References

Acharya, B., Actis, M., Aghajani, T., Agnetta, G., Aguilar, J., Aharonian, F., Ajello, M., Akhperjanian, A., Alcubierre, M., Aleksić, J., et al., 2013. Introducing the CTA concept. Astropart. Phys. 43, 3–18. doi:10.1016/j.astropartphys.2013.01.007.

Araya, M., Osorio, M., Díaz, M., Ponce, C., Villanueva, M., Valenzuela, C., Solar, M., 2018. JOVIAL: Notebook-based astronomical data analysis in the cloud. Astron. Comput. 25, 110–117.

Arviset, C., Navarro, V., Basso, D., Alvarez, R., Basso, D., del Rio, S., Diego, M., Lopez-Caniego, M., Lousa Marques, A., Marinic, F., et al., 2021. ESA DATALABS: Towards a Collaborative E-Science Platform for ESA. LPI Contributions 2549, 7014.

Asch, M., Moore, T., Badia, R., Beck, M., Beckman, P., Bidot, T., Bodin, F., Cappello, F., Choudhary, A., de Supinski, B., et al., 2018. Big data and extreme-scale computing: Pathways to convergence-toward a shaping strategy for a future software and data ecosystem for scientific inquiry. Int. J. High Perform. Comput. Appl. 32 (4), 435–479.

Bertocco, S., Goz, D., Tornatore, L., Ragagnin, A., Maggio, G., Gasparo, F., Vuerli, C., Taffoni, G., Molinaro, M., 2020. INAF trieste astronomical observatory information technology framework. In: Astronomical Data Analysis Software and Systems XXIX. In: Astronomical Society of the Pacific Conference Series, vol. 527, p. 303.

Bhandari Neupane, J., Neupane, R.P., Luo, Y., Yoshida, W.Y., Sun, R., Williams, P.G., 2019. Characterization of leptazolines A–D, polar oxazolines from the cyanobacterium leptolyngbya sp., reveals a glitch with the Willoughby–Hoye scripts for calculating NMR chemical shifts. Org. Lett. 21 (20), 8449–8453.

Bisong, E., 2019. Google colaboratory. In: Building Machine Learning and Deep Learning Models on Google Cloud Platform. Springer, pp. 59–64.

Bleuler, A., Teyssier, R., 2014. Towards a more realistic sink particle algorithm for the RAMSES CODE. Mon. Not. R. Astron. Soc. 445 (4), 4015–4036. doi:10.1093/mnras/stu2005, arXiv:1409.6528.

Boettiger, C., 2015. An introduction to Docker for reproducible research. Oper. Syst. Rev. 49 (1), 71–79.

Brady, K., Moon, S., Nguyen, T., Coffman, J., 2020. Docker container security in cloud computing. In: 2020 10th Annual Computing and Communication Workshop and Conference. CCWC, IEEE, pp. 0975–0980.

Castronova, A.M., Doan, P., Seul, M., 2018. A general approach for enabling cloud-based hydrologic modeling using jupyter notebooks. HydroShare.

CERFACS COOP-Algo Team, 2021. The counter-intuitive rise of Python in scientific computing. https://web.archive.org/web/20210218084422/https://cerfacs.fr/coop/fortran-vs-python. (Accessed 18 February 2021).

Chappell, D., 2015. Introducing azure machine learning. In: A Guide for Technical Professionals. Sponsored By Microsoft Corporation.

Chen, J., Jiang, J., Duan, H., Wan, T., Chen, S., Paxson, V., Yang, M., 2018. We still don't have secure cross-domain requests: an empirical study of {CORS}. In: 27th {USENIX} Security Symposium. {USENIX} Security 18, pp. 1079–1093.

Crafts, R.E., 1990. Department of Energy High Performance Computing Act of 1989: Hearing Before the Subcommittee on Energy Research and Development of the Committee on Energy and Natural Resources on S. 1976 to Provide for Continued United States Leadership in High Performance Computing. Vol. 4. U.S. Government printing office, pp. 197–198.

Cui, C., Tao, Y., Li, C., Fan, D., Xiao, J., He, B., Li, S., Yu, C., Mi, L., Xu, Y., Han, J., Yang, S., Zhao, Y., Xue, Y., Hao, J., Liu, L., Chen, X., Chen, J., Zhang, H., 2020. Towards an astronomical science platform: Experiences and lessons learned from Chinese virtual observatory. Astron. Comput. 32, 100392. doi:10.1016/j.ascom.2020.100392, arXiv:2005.10501.

Cupani, G., D'Odorico, V., Cristiani, S., Russo, S.A., Calderone, G., Taffoni, G., 2020. Astrocook: your starred chef for spectral analysis. In: Software and Cyberinfrastructure for Astronomy VI. Vol. 11452. International Society for Optics and Photonics, p. 114521U.

De Zeeuw, T., Tamai, R., Liske, J., 2014. Constructing the E-ELT. The Messenger 158, 3–6.

Desai, V., Allen, M., Arviset, C., Berriman, B., Chary, R.-R., Cook, D., Faisst, A., Dubois-Felsmann, G., Groom, S., Guy, L., Helou, G., Imel, D., Juneau, S., Lacy, M., Lemson, G., Major, B., Mazzarella, J., Mcglynn, T., Momcheva, I., Murphy, E., Olsen, K., Peek, J., Pope, A., Shupe, D., Smale, A., Smith, A., Stickley, N., Teplitz, H., Thakar, A., Wu, X., 2019. A science platform network to facilitate astrophysics in the 2020s. Bull. Am. Astron. Soc. 51, 146.

Dewdney, P.E., Hall, P.J., Schilizzi, R.T., Lazio, T.J.L.W., 2009. The square kilometre array. IEEE Proc. 97, 1482–1496. doi:10.1109/JPROC.2009.2021005.

Dursi, J., 2021. HPC is dying, and MPI is killing it. https://web.archive.org/web/20211029041512/https://www.dursi.ca/post/hpc-is-dying-and-mpi-is-killing-it. (Accessed 29 November 2021).

Gardner, J.P., Mather, J.C., Clampin, M., Doyon, R., Greenhouse, M.A., Hammel, H.B., Hutchings, J.B., Jakobsen, P., Lilly, S.J., Long, K.S., et al., 2006. The james webb space telescope. Space Sci. Rev. 123, 485–606. doi:10.1007/s11214-006-8315-7.

Gorda, B., 2021. HPC in the cloud? Yes, No and in between. https://web.archive.org/web/20210326114937/https://www.arm.com/blogs/blueprint/hpc-cloud. (Accessed 26 March 2021).

Goz, D., Ieronymakis, G., Papaefstathiou, V., Dimou, N., Bertocco, S., Simula, F., Ragagnin, A., Tornatore, L., Coretti, I., Taffoni, G., 2020. Performance and energy footprint assessment of FPGAs and GPUs on HPC systems using astrophysics application. Computation (ISSN: 2079-3197) 8 (2), doi:10.3390/computation8020034.

Habib, S., Pope, A., Finkel, H., Frontiere, N., Heitmann, K., Daniel, D., Fasel, P., Morozov, V., Zagaris, G., Peterka, T., Vishwanath, V., Lukić, Z., Sehrish, S., Liao, W.-k., 2016. HACC: Simulating sky surveys on state-of-the-art supercomputing architectures. New Astron. 42, 49–65. doi:10.1016/j.newast.2015.06.003, arXiv:1410.2805.

JupyterHub, 2016. Security overview. https://web.archive.org/web/20211127104158/https://jupyterhub.readthedocs.io/en/1.4.2/reference/websecurity.html. (Accessed 27 November 2021).

Jurić, M., Ciardi, D., Dubois-Felsmann, G., 2017. LSST science platform vision document. LSE-319, LSST.

Kaggle, 2018. How to use kaggle - notebooks. https://web.archive.org/web/20211112085454/https://www.kaggle.com/docs/notebooks. (Accessed 24 November 2021).

Katevenis, M., Ammendola, R., Biagioni, A., Cretaro, P., Frezza, O., Lo Cicero, F., Lonardo, A., Martinelli, M., Paolucci, P.S., Pastorelli, E., Simula, F., Vicini, P., Taffoni, G., Pascual, J.A., Navaridas, J., Lujn, M., Goodacre, J., Lietzow, B., Mouzakitis, A., Chrysos, N., Marazakis, M., Gorlani, P., Cozzini, S., Brandino, G.P., Koutsourakis, P., van Ruth, J., Zhang, Y., Kersten, M., 2018. Next generation of exascale-class systems: ExaNeSt project and the status of its interconnect and storage development. Microprocess. Microsyst. (ISSN: 0141-9331) 61, 58–71. doi:10.1016/j.micpro.2018.05.009.

Laureijs, R., Gondoin, P., Duvet, L., Criado, G.S., Hoar, J., Amiaux, J., Auguères, J.-L., Cole, R., Cropper, M., Ealet, A., et al., 2012. Euclid: ESA's mission to map the geometry of the dark universe. In: Space Telescopes and Instrumentation 2012: Optical, Infrared, and Millimeter Wave. Vol. 8442. p. 84420T. doi:10.1117/12.926496.

Major, B., Kavelaars, J., Fabbro, S., Durand, D., Jeeves, H., 2019. Arcade: An interactive science platform in CANFAR. In: Teuben, P.J., Pound, M.W., Thomas, B.A., Warner, E.M. (Eds.), Astronomical Data Analysis Software and Systems XXVII. In: Astronomical Society of the Pacific Conference Series, vol. 523, p. 277.

Mendez, K.M., Pritchard, L., Reinke, S.N., Broadhurst, D.I., 2019. Toward collaborative open data science in metabolomics using jupyter notebooks and cloud computing. Metabolomics 15 (10), 1–16.

Merloni, A., Predehl, P., Becker, W., Böhringer, H., Boller, T., Brunner, H., Brusa, M., Dennerl, K., Freyberg, M., Friedrich, P., Georgakakis, A., Haberl, F., Hasinger, G., Meidinger, N., Mohr, J., Nandra, K., Rau, A., Reiprich, T.H., Robrade, J., Salvato, M., Santangelo, A., Sasaki, M., Schwope, A., Wilms, J., The German eROSITA Consortium, 2012. eROSITA science book: Mapping the structure of the energetic universe. doi:10.48550/arXiv.1209.3114, arXiv:1209.3114.

Milligan, M., 2018. Jupyter as common technology platform for interactive HPC services. doi:10.48550/arXiv.1807.09929, arXiv e-prints arXiv:1807.09929.

Newman, S., 2015. Building Microservices. O'Reilly Media, Inc.

Nicklas, J.W., Johnson, D., Oottikkal, S., Franz, E., McMichael, B., Chalker, A., Hudak, D.E., 2018. Supporting distributed, interactive Jupyter and RStudio in a scheduled HPC environment with spark using open OnDemand. In: Proceedings of the Practice and Experience on Advanced Research Computing. PEARC '18, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450364461, pp. 1–8. doi:10.1145/3219104.3219149.

Piparo, D., Tejedor, E., Mato, P., Mascetti, L., Moscicki, J., Lamanna, M., 2018. SWAN: A service for interactive analysis in the cloud. Future Gener. Comput. Syst. 78, 1071–1078.

Ragagnin, A., Dolag, K., Biffi, V., Cadolle Bel, M., Hammer, N.J., Krukau, A., Petkova, M., Steinborn, D., 2017. A web portal for hydrodynamical, cosmological simulations. Astron. Comput. 20, 52–67. doi:10.1016/j.ascom.2017.05.001, arXiv:1612.06380.

Russo, S.A., Cupani, G., Bertocco, S., Molinaro, M., Taffoni, G., 2021. A microservice-oriented science platform architecture. In: Ruiz, J.-E., Pierfederici, F. (Eds.), Astronomical Data Analysis Software and Systems XXX. In: Astronomical Society of the Pacific Conference Series, 532, ASP, San Francisco, p. 23.

Schwenk, J., Niemietz, M., Mainka, C., 2017. Same-origin policy: Evaluation in modern browsers. In: 26th {USENIX} Security Symposium. {USENIX} Security 17, pp. 713–727.

Springel, V., Pakmor, R., Pillepich, A., Weinberger, R., Nelson, D., Hernquist, L., Vogelsberger, M., Genel, S., Torrey, P., Marinacci, F., Naiman, J., 2018. First Results from the IllustrisTNG Simulations: Matter and Galaxy Clustering. Vol. 475. (1), pp. 676–698. doi:10.1093/mnras/stx3304, arXiv:1707.03397.

Springel, V., Pakmor, R., Zier, O., Reinecke, M., 2020. Simulating cosmic structure formation with the GADGET-4 code. doi:10.48550/arXiv.2010.03567, arXiv e-prints arXiv:2010.03567.

Squarcina, M., Tempesta, M., Veronese, L., Calzavara, S., Maffei, M., 2021. Can I take your subdomain? Exploring same-site attacks in the modern web. In: 30th {USENIX} Security Symposium. {USENIX} Security 21, pp. 2917–2934.

SUSE, 2022. Technology definitions - containers. https://web.archive.org/web/20220112132030/https://www.suse.com/suse-defines/definition/containers/. (Accessed 12 January 2022).

Taffoni, G., Becciani, U., Garilli, B., Maggio, G., Pasian, F., Umana, G., Smareglia, R., Vitello, F., 2020. CHIPP: INAF pilot project for HTC, HPC and HPDA. In: Astronomical Data Analysis Software and Systems XXIX. In: Astronomical Society of the Pacific Conference Series, vol. 527, p. 307.

Taffoni, G., Bertocco, S., Coretti, I., Goz, D., Ragagnin, A., Tornatore, L., 2020. Low power high performance computing on arm system-on-chip in astrophysics. In: Arai, K., Bhatia, R., Kapoor, S. (Eds.), Proceedings of the Future Technologies Conference (FTC) 2019. Springer International Publishing, Cham, ISBN: 978-3-030-32520-6, pp. 427–446.

Taffoni, G., Lemson, G., Molinaro, M., Schaaff, A., Morris, D., Meyer–Zhao, Z., 2020. Science platforms: Towards data science. In: Astronomical Data Analysis Software and Systems XXIX. In: Astronomical Society of the Pacific Conference Series, vol. 527, p. 777.

Taffoni, G., Murante, G., Tornatore, L., Katevenis, M., Chrysos, N., Marazakis, M., 2019. Shall numerical astrophysics step into the era of exascale computing? In: Astronomical Data Analysis Software and Systems XXVI. In: Astronomical Society of the Pacific Conference Series, vol. 521, p. 567. doi:10.48550/arXiv.1904.11720, arXiv:1904.11720.

Taffoni, G., Tornatore, L., Goz, D., Ragagnin, A., Bertocco, S., Coretti, I., Marazakis, M., Chaix, F., Plumidis, M., Katevenis, M., Panchieri, R., Perna, G., 2019. Towards exascale: Measuring the energy footprint of astrophysics HPC simulations. In: 2019 15th International Conference on EScience. EScience, pp. 403–412. doi:10.1109/eScience.2019.00052.

Taghizadeh-Popp, M., Kim, J.W., Lemson, G., Medvedev, D., Raddick, M.J., Szalay, A.S., Thakar, A.R., Booker, J., Chhetri, C., Dobos, L., Rippin, M., 2020. SciServer: A science platform for astronomy and beyond. Astron. Comput. 33, 100412. doi:10.1016/j.ascom.2020.100412, arXiv:2001.08619.

Tasse, C., Hugo, B., Mirmont, M., Smirnov, O., Atemkeng, M., Bester, L., Hardcastle, M., Lakhoo, R., Perkins, S., Shimwell, T., 2018. Faceting for direction-dependent spectral deconvolution. Astron. Astrophys. 611, A87.

Tinarelli, F., Zorba, S., Knapic, C., Jerse, G., 2020. The authentication and authorization INAF experience. In: Astronomical Data Analysis Software and Systems XXVII. Vol. 522. p. 727.

Zalewski, M., 2009. Browser security handbook. https://web.archive.org/web/20211113072530/https://code.google.com/archive/p/browsersec/wikis/Main.wiki. (Accessed 13 November 2021).

Zalewski, M., 2012. The Tangled Web: A Guide to Securing Modern Web Applications. No Starch Press.