Rosetta: a container-centric science platform for resource-intensive, interactive data analysis. DRAFT VERSION

Stefano Alberto Russo[a], Giuliano Taffoni[a], Sara Bertocco[a]

[a]*INAF (Italian National Institute for Astrophysics) – Astronomical Observatory of Trieste, Via G.B. Tiepolo 11, 34143, Trieste, Italy*

Abstract

Rosetta is a science platform for resource-intensive and 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 allow to fully support custom software environments and interaction methods, including remote desktop and GUI applications, besides common web-based data analysis environments as the Jupyter Notebooks. Using software containers allows for safe, effective and reproducible code execution, enabling users to both choose from a set of pre-defined software and to set up their own ones, in order be always free to carry out their analysis tasks with the software and libraries that best suite their needs. Rosetta relies on OCI (Open Container Initiaive) containers, can use a number of container runtimes (as Docker, Singularity, CRI-O, Containerd, Kata, and more) and seamlessly supports several workload management systems (as classic HPC solutions like Slurm and Torque as well as mainstream and commercial solutions as Kubernetes and Amazon ECS Fargate), all with minimum integration effort. Although developed in the astronomy space, Rosetta can virtually support any science and technology domain where resource-intensive and interactive data analysis is required.

Keywords: science platforms, data analysis, containers, Docker, Singularity,

Email address: `stefano.russo@inaf.it` (Stefano Alberto Russo)

Kubernetes, Slurm, HPC, GUI applications, Jupyter Notebooks
PACS: 0000, 1111
2000 MSC: 0000, 1111

1. Introduction

The increasing data volumes produced in several research fields, as bioinformatics (due to next generation sequencing instruments), particle physics (due to new particle detectors), earth sciences (due to the advances in remote sensing techniques) and many others, 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) [1], the Cherenkov Telescope Array (CTA) [2], the Extremely Large Telescope (ELT) [3], the James Webb Space telescope [4], the Euclid satellite [5] and the eROSITA All-Sky Survey [6] 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 the current and upcoming surveys. State of the art cosmological N-body hydrodynamic codes (as OpenGADGET, GADGET4 [7] and RAMSES [8]) 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 [9, 10, 11, 12].

The size and complexity of these new experiments (both observational and numerical) require therefore considerable storage and computational resources for their data to be processed and analyzed, and possibly to adopt new approaches and architectures. High Performance Computing (HPC) clusters, 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 [13], 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 process, which could usually be performed on researchers' workstations and laptops, are getting too resource-intensive and thus progressively required to be offloaded to such systems as well.

Although capable of satisfying the necessary computing and storage requirements, these systems are usually centralised in order to share their (expensive) resources across different users and to allow co-location with the data, which translates in a potentially very hard user interaction. Operating on centralized and dynamically allocated computing resources can be indeed considered as an advanced task itself, especially for the inexperienced users, as it requires to heavily rely on remote connections for shell and graphical access (as SSH and X protocol forwarding), as well as to carefully plan how to locate, access and transfer the data to process. Bringing along the required software can be even more challenging, and without a 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 [14].

To address these challenges, we see an increasing effort in developing the so called *science platforms* [15, 16, 17, 18]. A science platform (SP) is an environment designed to offer users a smoother experience when interacting with centralized computing and storage resources, in order to mitigate some of the issues outlined in the previous paragraph.

However, the ongoing efforts on developing SPs tend to focus on web-based, integrated analysis environments built on top Jupyter Notebooks or similar software, and while this approach does make it easier to access this kind of computing resources, it also introduces two main drawbacks:

1. users are normally restricted in using a set of pre-defined software environments, with limited to no control over setting up custom ones, and
2. using interaction methods other than the web-based analysis environment integrated in the platform is particularly hard, if not impossible.

Not supporting custom software environments, besides limiting the users in their work, can severely affect reproducibility; while not supporting custom interaction methods makes it not possible to use "classic" graphical user interface (GUI) applications and other non web-based tools (which are common for data analysis in Astrophysics as well as other fields).

Moreover, the deployment options for nearly all the science platforms developed today rely on modern technologies from the IT industry (e.g. Kubernetes) and require deep integrations at system-level, which are often hard to achieve in the space of HPC clusters and data-intensive systems, either because of technological factors and legacy aspects as well as a generalized pushback for exogenous technologies from some parts of the HPC community [19, 20, 21, 22].

In this paper, we present a science platform specifically designed 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 environments and interaction methods, including remote desktops and GUI applications, besides standard web-based analysis environments as the Jupyter Notebooks. Its user tasks are run as software containers[1], which allow for safe, effective and reproducible code execution [24], and its users can choose from a set of pre-defined ones or set up their owns, in order be always free to carry out the data analysis with the software and libraries that bestsuite their needs. Rosetta is also designed with real-world deployment scenarios in mind, and to easily integrate with existing computing and storage resources together with their workload management systems (WMS), even when they lack support for containerized workloads or use a container runtime that misses some required features, which is particularly helpful for integrating with HPC clusters and data-intensive systems.

Although astronomy remains its mainstay (Rosetta has been developed in

[1] Software containers are packages that provide an entire runtime environment: an application, plus its dependencies, system libraries, settings and other binaries, and the configuration files needed to run it [23].

the framework of the EU funded project ESCAPE[2]), Rosetta can virtually support any science and technology domain.

This paper is organized as follows. After discussing the related works (Section 2), we give an overview of the Rosetta platform from a user prospective (Section 3). We then discuss more in detail its architecture, implementation and security aspects (Sections 4, 5 and 6). Next, we present the deployment and usage scenario in a real production environment and a few use cases we are supporting (Section 7), leaving the last Section to conclusions and future work.

2. Related works

Over the last years, a number of interactive data analysis platforms have been designed and developed, both in the public and private sectors. Here we present a selection of them that we found out to either show similar characteristics as Rosetta or to stand out in terms of adoption. While this selection has a focus on the public research sector, it has to be noted that the private sector is moving fast with respect to resource-intensive interactive data analysis, mainly driven by the recent advances in artificial intelligence and machine learning.

CERN SWAN [25] is CERN's effort to build towards the science platform paradigm. SWAN is a new service for interactive, web-based data analysis in the Cloud, making Jupyter Notebooks widely available on CERN computing infrastructure together with a Dropbox-like solution for data management. As of today, this solution does not provide support for applications other the Jupyter Notebooks and a built-in shell terminal, does not allow using custom software environments and requires heavy system-level integrations in order to be installed on top of existent computing systems.

ESA Datalabs [26] is a science platform that, similarly to CERN SWAN, allows users to work on ESA's computing infrastructure using interactive com-

[2]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.

puting environments as Jupyter Lab and Octave (or to choose from pre-packaged applications as TOPCAT), and providing direct access to ESA's archives. Datalabs is mainly focused on enabling users gaining 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) science platform [27] shows similar efforts, defining 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 software environments, and refers to its own computing architecture.

The Agave platform [28] is an example from the biology domain and it is presented as a "science as a service" platform for reproducible science. The project is strongly focused on providing APIs for developers to integrate with data and computing resources that can be managed from within Agave, but it also provides Jupyter Notebooks and RStudio integration. It is however somewhat orthogonal to our definition of a science platform, as it relies on a sort of intermediate API-based integration layer we do not foresee in our SP concept.

There are also a number of initiatives entirely focused on supporting Jupyter Notebooks on cloud and HPC infrastructures (such as [29], [30], [31] and [32]), which might fall in our SP definition to some extent, and in particular in Astronomy and Astrophysics it is worth to mention SciServer [18], Jovial [33] and CADC Arcade [34].

In the private sector, we wanted to cite Google Colab [35] and Kaggle Notebooks [36], which are built around heavily customised versions of the Jupyter Notebooks, and Azure Machine Learning [37], which provides a nearly full-optional SP, even if specifically targeted at machine learning workflows. These platforms share similar traits, which consist in the ability for the user to run a specific task using a pre-defined software environments with the option, in some cases, to install extra software packages at run-time. They are however

not suitable for our purpose, mainly because of their "as a service", commercial nature.

3. Overview

The Rosetta science platform is entirely designed to provide simplified access to centralized computing and storage resources while not restricting the users to a set of pre-defined software environments and interaction methods.

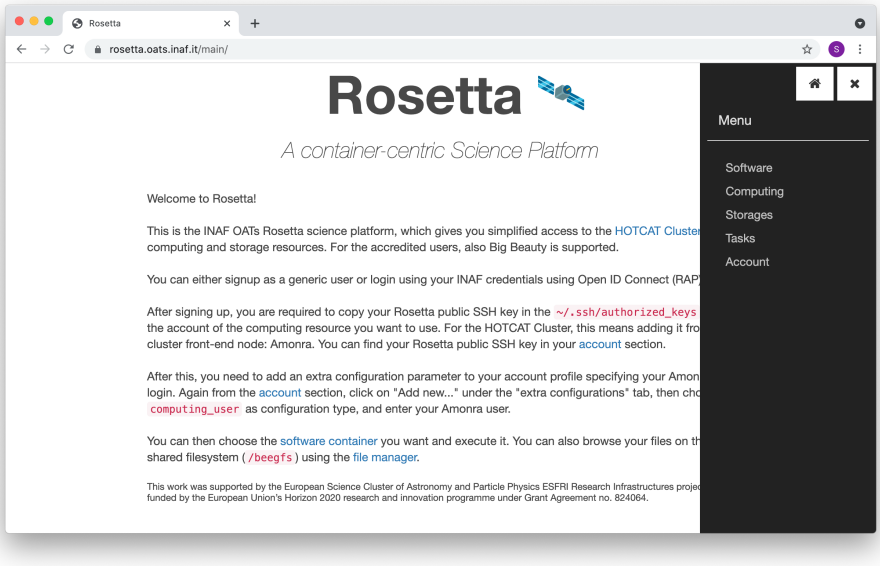

Figure 1: The Rosetta science platform main page and menu

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 Figure 1. The interface is organised in five min areas: the *Software* section, where to browse for the software containers available on the platform or where to add custom ones; the *Computing* section, where to view the available computing resources; the *Storage* area, 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.

To run a typical analysis task, the user first accesses the Software section in order to choose the desired software container, or to add a new one (Figure 2). If adding a new software container, the user has to set its registry, image name and tag, and the container interface port and protocol, plus some optional advanced attributes (Figure 3).

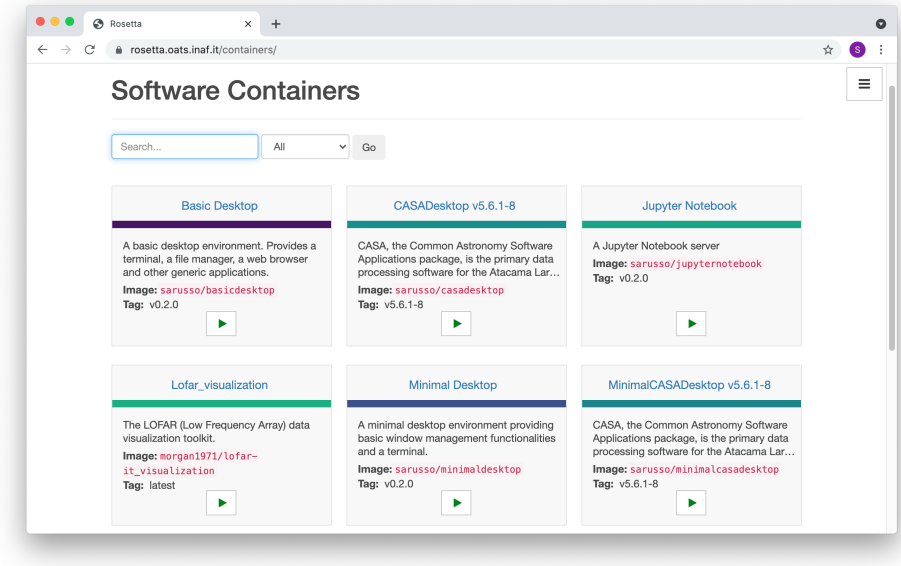

Figure 2: Software container selection

Once the software container is chosen, the user can hit the "play" button to start a task. Then, the platform will ask on which computing resource to run it, and to set a few options including a task name and a one-time task password (Figure 4). The task is then created and submitted.

As soon as the task is starting up on the computing resource, the user will receive a notification email with a direct link for the task page on Rosetta, where its "connect" button just become 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 in the container, where it can start working (Figure 5).

Users can also transfer files to and from the data storages directly from their

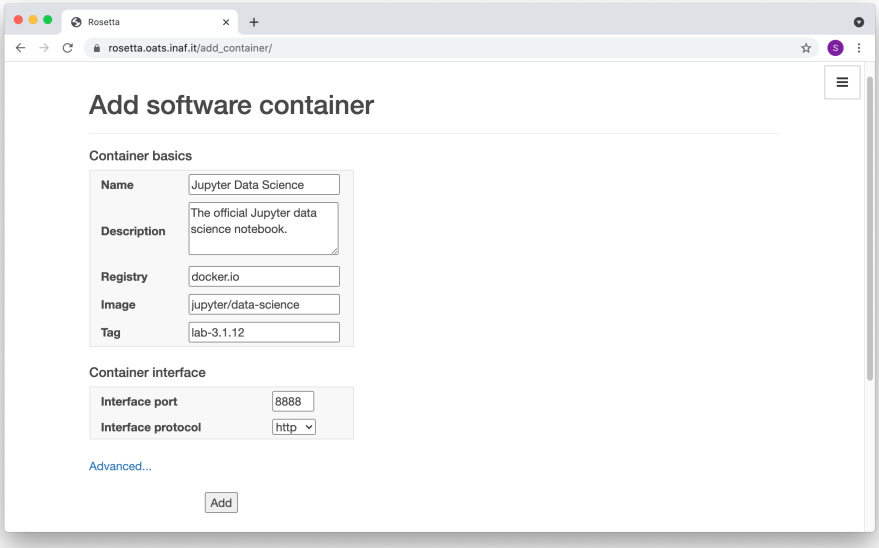

Figure 3: Adding a software container as user

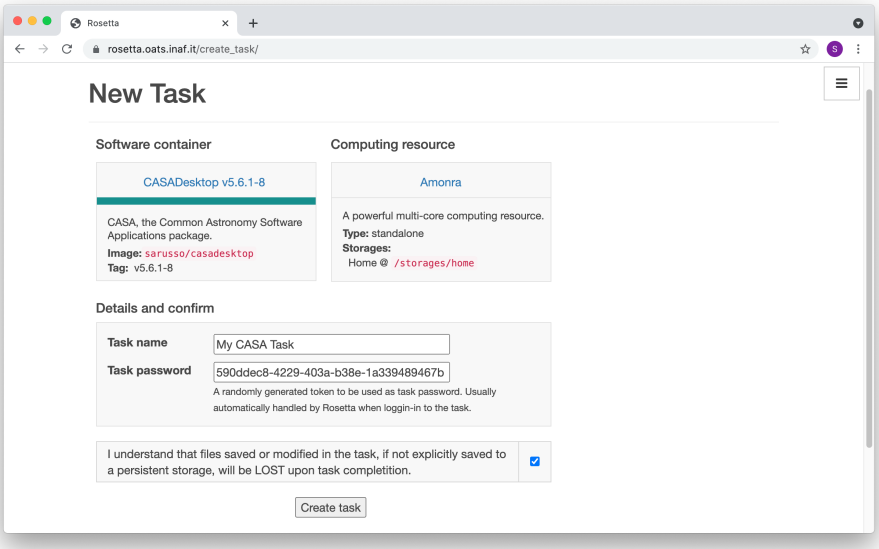

Figure 4: New task

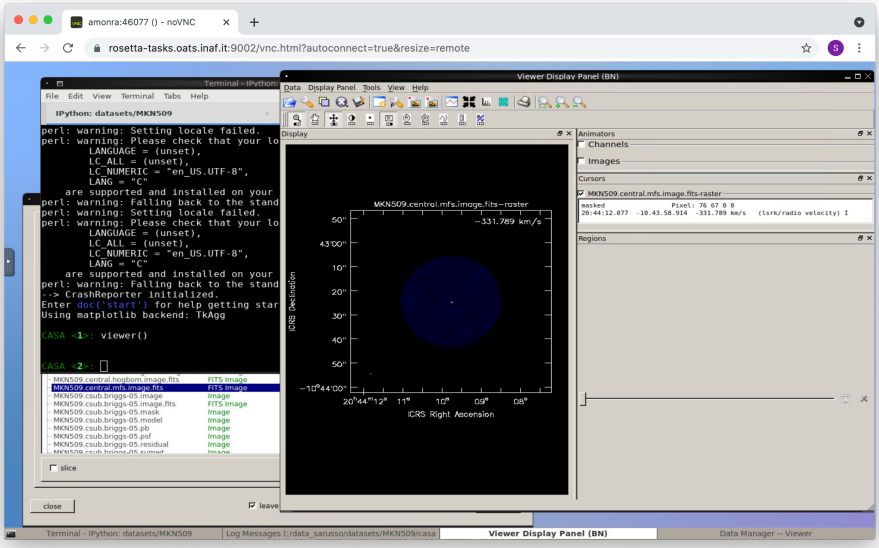

Figure 5: A GUI application running in Rosetta (CASA)

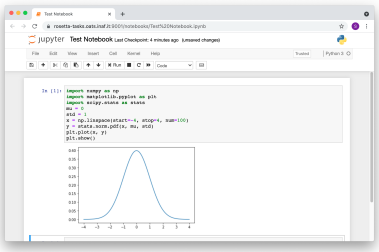

Figure 6: Other examples of user tasks: a Jupyer Notebook with an example plot using Numpy and Matplotlib.

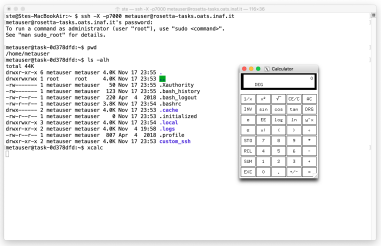

Figure 7: Other examples of user tasks: a SSH server with X forwarding and the Xcalc application.

laptops or workstations, using the built-in file manager in the Storage section, which is shown in Figure 8. While a web-based file manger is clearly not suitable for transferring massive datasets, which are supposed to be already located on a storage (either because the data repository is on the storage itself or because they have been staged using an external procedure), it is an effective solution for light datasets and analysis scripts and results.

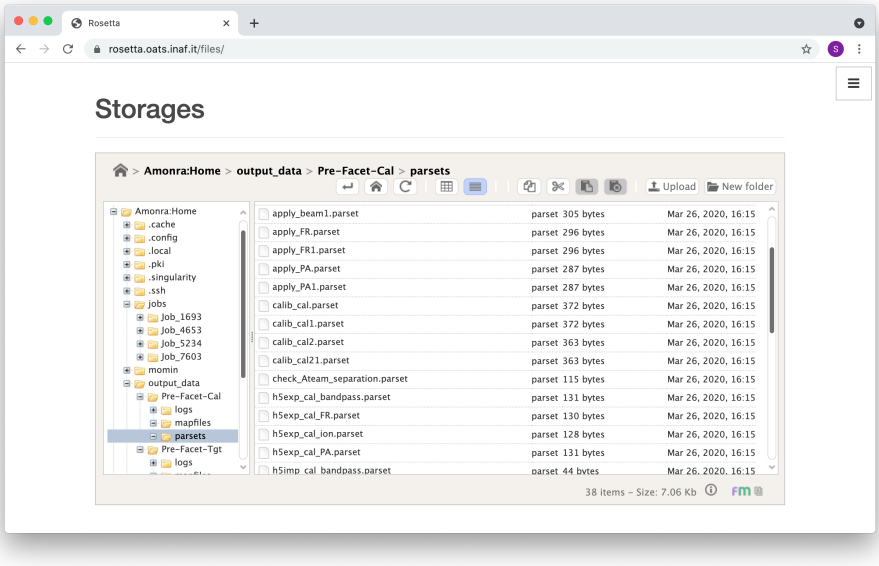

Figure 8: The Rosetta storage file manager

Computing resources are all made to follow the very same behaviour thanks to Rosetta internal architecture and its implementation, as it will be explained in the next sections, so that this flow applies in any scenario, regardless of deployment-specific peculiarities.

4. Architecture

Rosetta Architecture is twofold. The platform services follow a standard web application architecture based on a backend providing the various functionalities to a web-based UI, and comprises the web application service, the database service providing storage for internal data, and the proxy service for securing

the connections. The web application functionalities can be grouped in modules which are responsible for managing the software containers, interacting with the computing and storage resources, orchestrating the user tasks, handling user authentication and so on, reflecting the main five areas of the UI as per figure 1 and that are represented together with the other services in figure 9.

In particular, *Software* functionalities allow to track all the software containers available on the platform, their settings and on which container registry [3] they are stored, providing a sort of software catalog; *Computing* functionalities allow to interact with both standalone and clustered computing resources, hosted either either on premises (e.g. via Openstack) or on cloud systems (e.g. on Amazon AWS); *Storage* functionalities allow browsing and operating on both local and shared file system (as Ext4, NFS, BeeGFS); *Task* functionalities allow interacting with computing resources workload management systems (e.g. Slurm) and/or their container runtimes (e.g. Docker, Singularity, CRI-O, Containerd, Kata, etc.) including submitting and stopping the tasks and viewing their logs; and lastly *Account* functionalities provide user account and profile management including user registration, login and logout, supporting both local and external, federated authentication (e.g. OpenID Connect).

[3] 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-exisist at the same time.

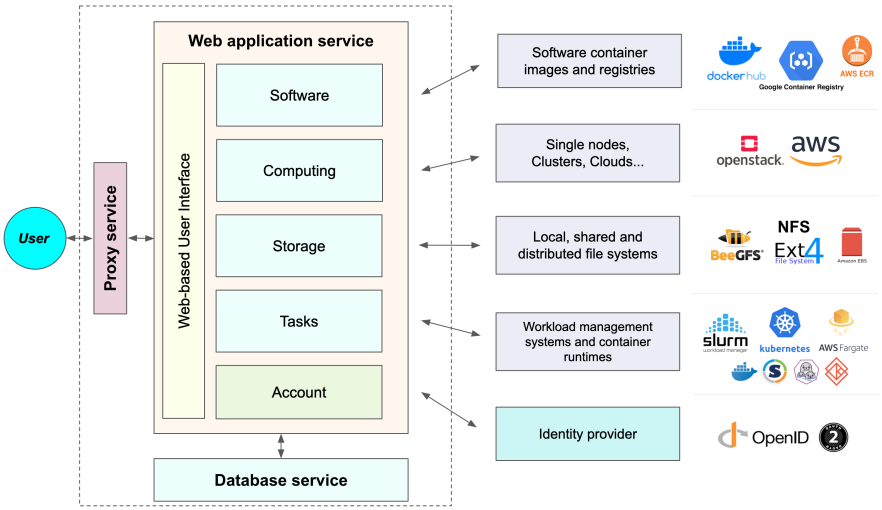

Figure 9: Rosetta platform services architecture.

Rosetta user task orchestration follows instead a novel, microservice-oriented approach [38] based on software containers. Microservices [39] are independent, self-contained and self-consistent units that perform a given task, which can range from a simple functionality (i.e. serving a file to download) to complex computer programs (i.e. classifying images using a neural network). They are interacted with using a well defined interface, usually a REST API over HTTP(S), but many others are possible (as RPC and MQTT). This interface is accessed using a specific transport layer, which can be considered fixed to the TCP/IP for Rosetta[4], and exposed on a given port. Moreover, microservices fit naturally in the containerisation approach, where each microservice runs in its own container, exposing its interface on a TCP/IP port and staying isolated from the underlying operating system, network, and storage layers. User task in Rosetta are thus always executed as software containers, and must behave as microservices. Rosetta can therefore stay agnostic with respect to the task interface, as long as the user knows how to use it (and it is based on a TCP/IP protocol). Examples of task interfaces include the Jupyter Notebooks, a virtual

[4]Other transport layers include the UDP/IP and lower level protocols.

network computing (VNC) server or a web-based remote desktop embedded in the task itself, 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 "official" and "user" containers, provided that the latter respect the requirement of exposing their interface on a TCP/IP port. Rosetta users can thus upload their software containers on a container registry and then add them in the platform by just setting up a few parameters as the registry URL, the image name and tag, and the TCP/IP port where the interface is running together with its protocol.

In order to orchestrate this kind of user tasks, Rosetta needs to be able to submit to the WMS a container for execution, and to know how to reach it (i.e. on which IP it is running). On computing resources and WMS not directly supporting containerized workloads (as Slurm), or using container runtimes missing some required features (as Singularity), Rosetta relies on an *agent*. The agent is a small software component in charge of helping to manage the task container life cycle, and that is sent by the platform on the computing resource where the task is scheduled to run. Its features include reporting the host computing resource IP address to the platform, setting up the environment for the container including dynamic port allocation, and running the container itself. Its internal logic is described more in detail in section 5.4.

When a task is started on a computing resource (either directly by the WMS or using the agent), its interface must be made accessible by the user. This is achieved in first place by making the interface port accessible on the internal network between Rosetta and the computing resource, either by opening a TCP/IP tunnel of by more sophisticated techniques which are usually available in modern container orchestrators and WMSs. Then, the connection from the user to the task interface can be established either directly, or via an intermediate step consisting in a proxy service, which allows to enforce access control and to secure the connection.

Once tasks are executed and their interface made accessible, since as mentioned before the SP does not need to know anything about the task interfaces, no further operations are required and the users can be looped in.

A diagram of this flow, with two examples using both a WMS with direct connection to the task interface (Figure 10) and the agent with the proxy as intermediate step (Figure 11) is presented below.

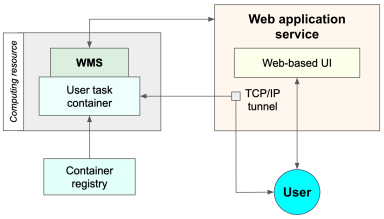

Figure 10: Rosetta user task orchestration using the computing resource WMS and a direct connection to the task interface through the TCP/IP tunnel.

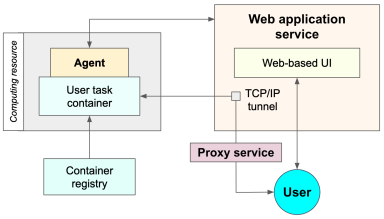

Figure 11: Rosetta user task orchestration using the agent and the proxy service on top of the TCP/IP tunnel for connectiing to the task interface.

5. Implementation

Rosetta is built using open-source technologies only, and in particular Python and the Django web framework. Other technologies include HTML and JavaScript for the UI, Postgres for the database[5], and Apache for the proxy. The platform make use of Docker containers for its services (not to be confused with the user tasks software containers), and Docker Compose as the default orchestrator[6]. Besides the web application, database and proxy services, Rosetta also comes with an optional local Docker registry service, which can be used for storing software containers locally, and a test Slurm cluster with two nodes, which can be used for testing and debugging purposes. A set of management scripts to

[5]The database service can be replaced by another database of choice, provided that it is supported by Django, by changing the web application service configuration.

[6]Other orchestrators can be easily supported as well, as Kubernetes.

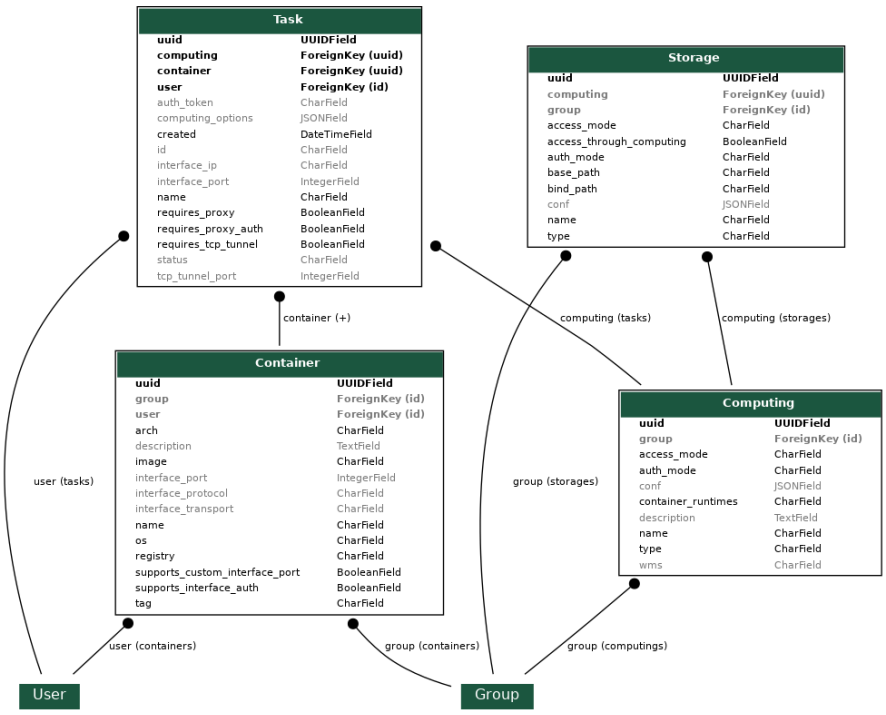

Figure 12: Rosetta Django ORM schema (excluding some minor, less relevant models as the user profile, the login tokens, the key pairs and the custom pages).

build, bootstrap and operate the platform is present as well, and a particular focus has been put on logging, which is capable of handling both user-generated and system errors, separating them and properly capturing exceptions and stack traces.

The implementation of the various web application functionalities introduced in section 4 is described more in detail in the following subsections. In general, they are handled with a mixture 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 Figure 12.

5.1. Software

Software lives in Rosetta only as software containers. Software containers are represented using a Django ORM model which acts as a sort of twin of the "real" container, providing metadata and information about the container itself. Rosetta relies on Open Container Initiative (OCI) containers, which must be stored on an OCI-compliant container registry (as the Docker registry), besides following the microservice-oriented approach presented in section 4. The `Container` ORM model has a generic `name` and `description` fields, a `registry`, `image` and `tag` attributes to identify the container image and the registry where it is stored, a `interface_port` attribute to let Rosetta know the default port on which the container will expose its interface, and a `interface_protocol` attribute to set the interface protocol (e.g. HTTP, SSH, VNC etc.). Other attributes as the `arch` (defaulted to x86_64), `os` (defaulted to Linux) and `interface_transport` (defaulted to TCP/IP) allow for more fine-grained control over non-standard scenarios. Containers can be registered in Rosetta as platform (official) containers, where they are not associated with a specific user, or as user containers, where they belong and are visible to a specif user only. If the container is a platform container, its `user` attribute is left blank, otherwise is set to the user who set up the container. Each container can also be shared with (and made accessible only to) a specific `group`.

As mentioned in the introduction, Rosetta is built with the requirement of integrating with computing resources and WMS not directly supporting containerized workloads and container runtimes missing some required features. One of these features is the capability of mapping TCP/IP ports exposed by a container on *different* ports on the host. If this feature is missing, a container using an interface port already allocated on the host (either by another service or another instance of the same container) would then fail to start. To overcome this limitation, the Rosetta agent (which is always required for this kind of WMS and container runtimes) can provide a specific port to the container where to make its interface listening on, via an environment variable, which is chosen between the free ephemeral ports of the host. If a container is built to

support setting its interface port in this way, it can then take advantage of this mechanism to prevent port clashes. Otherwise, it would just fail to start int the scenario outlined above. In order to let Rosetta know that a given container supports this feature, its extra attribute `supports_setting_interface_port` must be set to to `True` (and the `interface_port` attribute will be then discarded).

Another important aspect about how containers are handled in Rosetta is tightly related to task access control. Since exposing container interfaces on a TCP/IP port tunneled by the platform to the outside world makes them basically public-facing, they require to be secured. For this to happen, Rosetta allows to setup a one-time password or token at task creation-time to be used for accessing the task afterwards. Tasks can get password-protected in two ways: by implementing a password-based authentication at task-level, or by delegating it to the HTTP proxy service. In order to set the password at task-level, Rosetta can forward it (either directly of via the agent) to the container via an environment variable, provided that the container is built to support it. In this case, the container must be registered on the platform with the extra `supports_interface_auth` attribute set to `True`. If the task software container uses an HTTP interface, it can instead delegate access control to the HTTP proxy service, and just expose a plain, unprotected interface over HTTP. In this case, Rosetta will setup the proxy service on the fly in order to ask for the task password or token when accessing its interface (plus securing it using SSL). Delegating the task authentication to the HTTP proxy service is the default approach for HTTP-based interfaces, since it is more secure than leaving the authentication to be implemented at task-level, as it will be discussed in section 6.

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

- JupyterNotebook, the official Jupyter Notebook container extended to

support custom interface ports;

- GUIApplication, a container 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;
- 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, plus supporting dynamic resolution changes in order to always fit the web browser window, which are essential features in the everyday use.

5.2. Computing

Computing resources are divided in two main classes: *standalone* and *clusters*. The first ones can or can not have a WMS in place, while the second ones always does. If the computing resource have no WMS, then the task execution is synchronous, otherwise the execution is asynchronous and the tasks can get queued. The Django ORM model class used to represent the computing resources is named `Computing`, and it includes a `type`, a `name` and a `description`, plus specific attributes for describing how to access the computing resource and to schedule/submit workloads. These are the `access_mode` attribute, which 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` attribute, which specifies how the platform gets authorized in order to control the computing resource; the `WMS` attribute, which specifies the WMS used by computing resource (or if there is none); and the `container_runtimes` attribute, which specifies the container runtimes available on the computing resource. With respect to

this last attribute, if the WMS natively supports containerized workloads and there is no need of running tasks using a specific container runtime, then it can be set to `auto`.

Some example combinations of these attributes are reported in Table 1, where each row correspond to a physical computing resource. The first row represents a classic HPC cluster using Slurm as workload management system and Singularity as container runtime, requiring an accredited cluster user to submit tasks over SSH using the Slurm command line interface. The second row is the same cluster but supporting both Docker and Singularity container runtimes (more details about how Rosetta allows to safely run Docker containers on this kind of computing resources are provided in section 6). 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 are instead standalone computing resources, using the Docker container runtime, 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 examples all use computing resources managed with Kubernetes. In the first two the container runtime is set to "auto", which as mentioned before delegates its choice to Kubernetes. Kubernetes can indeed be configured (at system-level) with a variety of container runtime environments, including Containerd, CRI-O, Kata-containers, Singularity, and more. However, in the eight row the container runtimes configured in Kubernetes are explicitly stated, in order to allow users to chose the best one to suite their needs. The differences in the access methods reflect instead the various modes in which a Kubernetes cluster can be controlled: in the sixth row Rosetta will use a command line interface from within the Rosetta platform itself, in the seventh it will connect to the Kubernetes master node and use a command line interface from there, while in the eight row it will use the Kubernetes APIs. The last row is an example using Fargate, an hosted container execution service from Amazon Web Services (AWS) built on top of their Elastic Container Service (ECS), accessed using its APIs. Not all these combinations of access modes, authentication modes and workload

ID	access_mode	auth_mode	WMS	container_runtimes
#1	SSH+CLI	user keys	Slurm	Singularity
#2	SSH+CLI	user keys	Slurm	Docker,Singularity
#3	API	JWT	Slurm	Docker,Singularity
#4	SSH+CLI	user keys	none	Docker
#5	API	platform cert.	none	Docker
#6	CLI	platform cert.	Kubernetes	auto
#7	SSH+CLI	platform keys	Kubernetes	auto
#8	API	platform cert.	Kubernetes	Containerd,Kata
#9	API	platform cert.	Fargate	auto

Table 1: Computing resource attributes combination examples, by computing resource ID.

management systems are implemented in Rosetta yet, however we wanted to lie down a general framework in order to easily expand the platform in future.

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` and `WMS` attributes.

A particular role play SSH-based computing resources, which as mentioned above are accessed using standard SSH either on behalf of the user signed into Rosetta (using its account on the computing resource) or on behalf of the platform itself. In order to access on behalf of the user, Rosetta generates a dedicated private/public key pair, and the user is required to add its Rosetta public key on

the computing resource. To instead allow accessing on behalf of the platform, a dedicated account (and keys) are required to be setup. This specific class of computing resources require no integration at all with existent computing infrastructures, since provided that a container runtime is available only an SSH access (even unprivileged) is required, and thus perfectly fits our requirement of operating on HPC cluster and data-intensive systems where system-level integrations are hard to achieve.

5.3. Storage

Storage functionalities provides a way of defining, mounting and browsing data storages. The `Storage` ORM model has a `name`, a `type` and, similarly to the computing resources, an `auth_mode` and an `access_mode` attributes. As for the other models a storage can be made available to a set of users only using the `group` attribute, and it can be attached to a computing resource using the `computing` attribute. In this case, if the storage uses the same access mode as its computing resource, then the `access_through_computing` option can be ticked so that it can just rely on that. 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.

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 `bind_path` set to `/storages/user_data`, in order to separate data form different users at orchestration-level. Moreover, if a data storage is attached to a computing resource and its `bind_path` is set, then it will be made accessible on all the containers running on that computing resource under the location specified by the `bind_path`.

A set of APIs are implemented as well, in order to provide support the file manager embedded in the Rosetta web-based UI, which is built on top of

the Rich File Manager[7] open source project. Thede APIs implement a set of common functionalities (as get, put, dir, rename etc.), in order to perform common file management operations, the internal logic of which depends on the storage type.

5.4. 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 or state, while when a WMS is in place they are created in the queued state and set to 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 (`container`, `computing` and `computing_options` attributes, respectively). A series of other attributes as the `interface_ip`, `interface_port`, `tcp_tunnel_port` and `auth_token`, together with the attributes setting if the task requires to rely on a TCP tunnel or the proxy, are dedicated to let Rosetta know how to instantiate the connection to the task (i.e. if to set up the tunnel and/or configure the proxy service).

Once a task starts on a computing resource, on the Rosetta web application side the task IP address and port are stored in the respective `Task` ORM model fields, the task is marked as running and an email is sent to the user with a link to the task page. This last step is particularly useful for tasks which are long queued, as it allows the users to immediately know when they start, thus preventing to waste computing time. After this, when the user click on the "connect" button in the task page, the `Task` model is queried to obtain the connection parameters, and depending if the task requires them, a TCP/IP tunnel is opened and/or the HTTP proxy is configured on the fly to provide access control for the task interface.

An important part of task handling in Rosetta consists in the agent, which

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

as mentioned in Section 4 allows to support both WMSs not directly supporting containerized workloads and container runtimes missing some required features. The agent is a Python script which is served by the Rosetta web application. When a task is scheduled to run on a computing resource which requires the agent, Rosetta will actually send a bootstrap script, that in turn will pull the agent from Rosetta and execute it. As soon as the agent gets executed, it 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 runtime missing the feature of mapping container ports on different ports on the host, then it also searches for an available ephemeral TCP/IP port, and communicates it to the web application as well. Then, the agent sets up the environment for the user task container, including providing the free ephemeral port just detected via an environment variable should this be the case, and starts it. The agent can run both as a superuser or, most importantly, as a regular, unprivileged user registered on the computing resource. Which of the two modes to use is up to the administrators, and their implications will be described in section 6.

5.5. Account

Account and profile functionalities provide support for both local Django users and external authentication services (e.g. Open ID connect). Users logging in using an external authentication service are created as local Django users as well, which is a common approach for this kind of authentication. The email address is the joint point between local and external identities, and manual intervention is required in case of changes.

Locally and externally-authenticated users can co-exist at the same time, provided that if a user originally singed up using an external authentication service then it will be always required to login 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.

The user profile supports also some user-based configuration parameters for the computing resources that require them (e.g. the computing resource user-

name if using SSH-based access with user keys). Other minor functionalities, as password recovery, login tokens and time zone settings are provided as well.

As already introduced in the previous sections, on top of single users Rosetta also provides groups, so that a given computing or storage resource, as well as a specific software container, can be made available to a set of users only.

6. Security

Rosetta is developed to be as secure by design as possible. The first layer of security is represented by using software containers for user tasks. Most importantly, the base executable unit is the container itself, meaning that in Rosetta users has no control outside the container at all: once the container is sent for execution, and Rosetta handles all the orchestration, the user is dropped inside the container, and cannot escape it. From within Rosetta-controlled computing resources, users have therefore no access outside their containers at all, meaning that if one of them gets compromised, all the other ones as well as the underlying host system does not get affected.

When Rosetta is integrated on existent HPC clusters and standalone computing resources using SSH-based access on behalf of the users, the administrators can opt for revoking their access at shell level, so that Rosetta - and its containerized tasks - becomes the only access point to the computing system even if relying on classic Linux users (i.e. for accounting purposes). This approach allows to both fully adopt the extra security layer provided by containers on the entire system, and to allow using container runtimes as Docker which require granting more permission to the Linux users.

As mentioned in section 5.1, a container exposing its interface on a port tunneled from Rosetta is basically public facing and it requires to be secured both in term of access control 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 proto-

col. The connection from the proxy service to the webapp service (which will in turn tunnel the connection to the task) is on a virtual dedicated network within Rosetta services, and thus assumed safe. Generic task interfaces (i.e. a VNC or X servers) are instead required to be secured at task-level, since they are exposed directly by the webapp using a tunnel to the task bypassing the proxy service. Access control is achieved by forwarding to the task a one-time password (set by the user at task creation-time) via an environment variable, which is then to be used by the service running in the container to restrict access. Connection encryption can instead be implemented in first instance using self-signed certificates, or implementing more complex solutions as dynamic certificate provisioning. For what concerns the tunnel from the webapp service to the task, this is protocol-agnostic (above the TCP/IP transport layer) and is either accomplished by a direct connection on a private and dedicated network (i.e. if using Kubernets) or using an SSH-based TCP/IP tunnel on an internal shared network, secured using user's public/private keys.

An important feature in the task security context is that Rosetta makes a strong distinction between regular and power users, where only the latter can setup custom software containers using generic task interface protocols other that the HTTP, since handling security beyond HTTP-based task interfaces is error prone from the users. Who is a power user is up to the administrators, which can set the user status in their profile in Rosetta, and could possibly involve institutional paperwork and sign-off.

In terms of security of the Rosetta platform intended as a web application, 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 [40, 41], 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 (i.e. by setting up a task serving a web page). In this context, we focused on isolating user tasks from the rest of the platform even on the client side. While using the same domain for both the platform and its HTTP-based tasks (i.e. `https://rosetta.platform/tasks/1` is definitely not a viable option as it does not even allow to enforce the SOP policy, also as using dedicated subdomains (i.e. `https://task1.rosetta.platform`) has several issues, in particular involving cookies [42, 43, 44]. We thus opted for serving user tasks from a *separate* domain (i.e. `rosetta-tasks.platform`). However, managing and securing subdomains like `task1.rosetta-tasks.platform` would require wildcard DNS and SSL, which for many institutional domains are not available [45]. For this reason, we decided to offer an intermediate solution using different ports for different task, under the same SSL certificate: `https://rosetta-tasks.platform:7001` is the URL form which to reach the (running) task number 1, secured by the same SSL certificate covering `rosetta-tasks.platform` but treated as a different origin with respect to `https://rosetta-tasks.platform:7002`. The SSL certificates are indeed port-agnostic, while the SOP (which basically involves the triplet protocol, host and port) it is not, thus enabling to be enforced at client-side.

We acknowledge that security of computing systems and web applications is a wide chapter and and 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 while designing and implementing Rosetta.

7. Deployment and use cases

Rosetta is deployed in production at the computing centre of INAF – Osservatorio Astronomico di Trieste [46], using an aggregated authentication system named RAP [47] and serving a set of different users with different software requirements.

To support our user community, we offer a pre-defined portfolio of container-

ized 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 quasar spectral analysis*, and *the HPC FPGA bitstream design*.

7.1. 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 [48], 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. The task is run on an HPC cluster managed using the Slurm WMS, allocating a set of resources in terms of RAM and CPUs as requested by the scientists in the task creation phase.

The task is competing 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 [8] 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.

7.2. The SKA data challenges

INAF participated in the SKA Data Challenges[9] 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 optimise their analyses for extracting scientific results from them.

The participants in the second SKA Data Challenge analysed a simulated dataset of $1TB$ in size, in order to find and characterise 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 possibly a GPU, 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 LOFAR pipelines use case, we provided a software container that provided all 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 ($> 4GBs$) 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.

[8] https://www.ict.inaf.it/gitlab/lofarit/containers

[9] https://sdc2.astronomers.skatelescope.org/sdc2-challenge

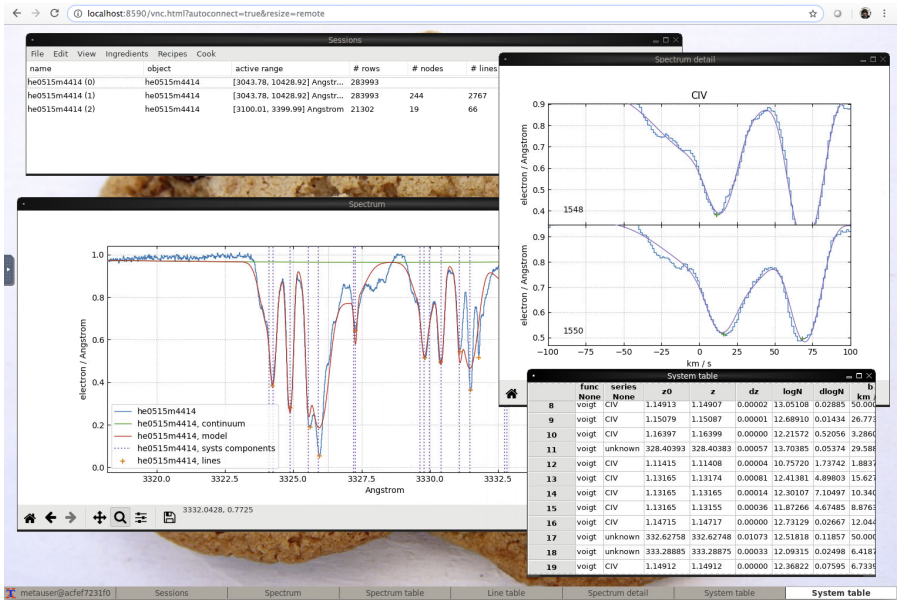

Figure 13: Astrocook running via Rosetta

7.3. The quasar spectral analysis

Astrocook[49] is a 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 relevant computing power in order to apply the various recipes.

Astrocook comes as a GUI application with some common and less common Python dependencies, which are sometimes hard to install, and it is a great example about how to provide a one-click access to a GUI application which might require some extra computing power using Rosetta.

Figure 13 shows the Astrocook software container running via Rosetta on a mid-sized, standalone computing resource, and accessed by the web-based desktop interface.

7.4. 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 prototype [50], a liquid-cooled cluster composed by 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 [51, 52, 53].

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 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 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: 128GB of RAM, extensive multi-threading support and 100 GB of hard disk space are required for creating 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.

We adopted Rosetta as the primary tool for FPGA programming at INAF computing cluster, that suited very well the use case using web-based remote desktops on computing resources capable of providing the required computing power and storage, and that allowed the users to let the HLS tools to work even if they disconnected and reconnected the day after.

8. Discussion

The two main challenges we faced in designing and developing Rosetta consisted in finding how to support custom software environments and interaction

methods on centralized computing resources, and how not to drop support for well established technologies powering HPC clusters and data-intensive systems.

With respect to the first challenge, the novel architecture we designed and adopted, based on framing user tasks as microservices, allowed to fully support custom software environments and interaction methods (as GUI applications and remote desktops, both as web-based and using an X server) and together with software containers allowed to ensure a safe, consistent and reproducible code execution across different computing resources.

How we were able to support well established technologies powering HPC clusters and data-intensive systems not fully supporting containerized workloads deserves instead a more articulated discussion.

These systems indeed still relies on Linux users for a number of reasons, including accounting purposes and local permission management. This means that containerisation solution born in the IT industry, where they are assumed to be operated using an administrative account, are usually not suitable.For this reason, the Singularity container runtime was designed to operate exclusively at user-level, and quickly become the standard in the HPC space.

However, it has to be noted that Singularity is designed to act more as an environment than a containerization solution, and lacks proper isolation between the host system and the container. In Singularity containers, by default directories as the home folder, `/tmp`, `/proc`, `/sys`, and `/dev` are indeed all shared with the host, the 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 that Singularity provides 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 (often with a default size of 16MB) and not on the host file system, thus very limited in space and easy to fill up.

We had therefore to address all these issues before being able to use Singularity 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, thus greatly improving reproducibiliy.

Once we were able to ensure a standardised behaviour of container runtimes and workload management systems, we were able to provide uniform task execution across different kinds of computing resources, and providing the very same user experience. In this sense, Rosetta can be considered as an umbrella for a variety of computing resources, and that can act as a sort of bridge in the transition towards software containers.

9. Conclusions and future work

We presented Rosetta, a science platform for resource-intensive and interactive data analysis aimed at filling the gap between providing simplified access to centralized computing and storage resources while not restricting the users to a set of pre-defined software only.

To achieve this goal, we developed a novel architecture based on framing user tasks as microservices – independent and self-contained units – and implemented them as software containers, which allowed to fully support custom software environments and interaction methods as remote desktops and GUI applications, besides web-based environments as the Jupyter Notebooks. In particular, adopting software containers allowed for safe, effective and reproducible code execution, and to let our users to add their own software should the pre-defined ones not suite their needs.

We also took real-world real-world deployment scenarios in mind, and designed Rosetta to easily integrate with existent computing resources and workload management systems, even where they lack support for containerized workloads or make use of container runtimes missing some required features (as for Slurm and Singularity), which proved to be particularly helpful for integrating with HPC clusters and data-intensive systems.

We successfully tested Rosetta in the context of the ESCAPE project, which funded this work, for running LOFAR data reduction pipelines at INAF computing centres, as well as for other use cases of our user community including the SKA data challenges.

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

The work we carried out in order to let different workload management systems and container runtimes to co-exist, while at the same time moving to a uniformed, container-centric approach, allowed us to build a platform projected to the future while still supporting well established technologies powering HPC and data-intensive systems, and that can act as a sort of bridge in the transition towards software containers.

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

Future work include adding support for distributed workloads (i.e. MPI, Ray), developing a command line interface, integrating with data staging solutions (i.e. CERN's Rucio) and continuing the implementation efforts for current and new workload management systems (i.e. Torque, Openshift, Rancher, Nomad, and more).

10. Acknowledgements

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 centre of INAF-Osservatorio Astronomico di Trieste, [46, 54], for the availability of computing

resources and support.

References

[1] P. E. Dewdney, P. J. Hall, R. T. Schilizzi, T. J. L. W. Lazio, The Square Kilometre Array, IEEE Proceedings 97 (2009) 1482–1496. doi:10.1109/JPROC.2009.2021005.

[2] B. S. Acharya, M. Actis, T. Aghajani, et al., Introducing the CTA concept, Astroparticle Physics 43 (2013) 3–18. doi:10.1016/j.astropartphys.2013.01.007.

[3] T. de Zeeuw, R. Tamai, J. Liske, Constructing the E-ELT, The Messenger 158 (2014) 3–6.

[4] J. P. Gardner, J. C. Mather, M. Clampin, et al., The james webb space telescope, Space Science Reviews 123 (2006) 485–606. doi:10.1007/s11214-006-8315-7.

[5] R. Laureijs, P. Gondoin, L. Duvet, et al., 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, 2012, p. 84420T. doi:10.1117/12.926496.

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

[7] V. Springel, R. Pakmor, O. Zier, M. Reinecke, Simulating cosmic structure formation with the GADGET-4 code, arXiv e-prints (2020) arXiv:2010.03567arXiv:2010.03567.

[8] A. Bleuler, R. Teyssier, Towards a more realistic sink particle algorithm for the RAMSES CODE, MNRAS 445 (4) (2014) 4015–4036. arXiv:1409.6528, doi:10.1093/mnras/stu2005.

[9] V. Springel, R. Pakmor, A. Pillepich, R. Weinberger, D. Nelson, L. Hernquist, M. Vogelsberger, S. Genel, P. Torrey, F. Marinacci, J. Naiman, First results from the IllustrisTNG simulations: matter and galaxy clustering 475 (1) (2018) 676–698. arXiv:1707.03397, doi:10.1093/mnras/stx3304.

[10] A. Ragagnin, K. Dolag, V. Biffi, M. Cadolle Bel, N. J. Hammer, A. Krukau, M. Petkova, D. Steinborn, A web portal for hydrodynamical, cosmological simulations, Astronomy and Computing 20 (2017) 52–67. arXiv:1612.06380, doi:10.1016/j.ascom.2017.05.001.

[11] G. Taffoni, G. Murante, L. Tornatore, M. Katevenis, N. Chrysos, M. Marazakis, Shall Numerical Astrophysics Step Into the Era of Exascale Computing?, in: M. Molinaro, K. Shortridge, F. Pasian (Eds.), Astronomical Data Analysis Software and Systems XXVI, Vol. 521 of Astronomical Society of the Pacific Conference Series, 2019, p. 567. arXiv:1904.11720.

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

[13] M. Asch, T. Moore, R. Badia, M. Beck, P. Beckman, T. Bidot, F. Bodin, F. Cappello, A. Choudhary, B. de Supinski, et al., Big data and extreme-scale computing: Pathways to convergence-toward a shaping strategy for a future software and data ecosystem for scientific inquiry, The International Journal of High Performance Computing Applications 32 (4) (2018) 435–479.

[14] J. Bhandari Neupane, R. P. Neupane, Y. Luo, W. Y. Yoshida, R. Sun, P. G. Williams, 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, Organic letters 21 (20) (2019) 8449–8453.

[15] G. Taffoni, G. Lemson, M. Molinaro, A. Schaaff, D. Morris, Z. Meyer–Zhao, Science Platforms: Towards Data Science, in: R. Pizzo, E. R. Deul, J. D. Mol, J. de Plaa, H. Verkouter (Eds.), Astronomical Society of the Pacific Conference Series, Vol. 527 of Astronomical Society of the Pacific Conference Series, 2020, p. 777.

[16] V. Desai, M. Allen, C. Arviset, B. Berriman, R.-R. Chary, D. Cook, A. Faisst, G. Dubois-Felsmann, S. Groom, L. Guy, G. Helou, D. Imel, S. Juneau, M. Lacy, G. Lemson, B. Major, J. Mazzarella, T. Mcglynn, I. Momcheva, E. Murphy, K. Olsen, J. Peek, A. Pope, D. Shupe, A. Smale, A. Smith, N. Stickley, H. Teplitz, A. Thakar, X. Wu, A Science Platform Network to Facilitate Astrophysics in the 2020s, in: Bulletin of the American Astronomical Society, Vol. 51, 2019, p. 146.

[17] C. Cui, Y. Tao, C. Li, D. Fan, J. Xiao, B. He, S. Li, C. Yu, L. Mi, Y. Xu, J. Han, S. Yang, Y. Zhao, Y. Xue, J. Hao, L. Liu, X. Chen, J. Chen, H. Zhang, Towards an astronomical science platform: Experiences and lessons learned from Chinese Virtual Observatory, Astronomy and Computing 32 (2020) 100392. arXiv:2005.10501, doi:10.1016/j.ascom.2020.100392.

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

[19] 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, 1990, pp. 197–198.

[20] B. Gorda, HPC in the Cloud? yes, no and in between, https://arm.com/blogs/blueprint/hpc-cloud, accessed: 2021-10-18.

[21] CERFACS COOP-Algo Team, The counter-intuitive rise of Python in scientific computing, https://cerfacs.fr/coop/fortran-vs-python, accessed: 2021-10-18.

[22] J. Dursi, HPC is dying, and MPI is killing it, https://www.dursi.ca/post/hpc-is-dying-and-mpi-is-killing-it, accessed: 2021-10-18.

[23] SUSE, Technology definitions - Containers, https://www.suse.com/suse-defines/definition/containers/, accessed: 2021-10-18.

[24] C. Boettiger, An introduction to Docker for reproducible research, ACM SIGOPS Operating Systems Review 49 (1) (2015) 71–79.

[25] D. Piparo, E. Tejedor, P. Mato, L. Mascetti, J. Moscicki, M. Lamanna, SWAN: A service for interactive analysis in the Cloud, Future Generation Computer Systems 78 (2018) 1071–1078.

[26] Esa datalabs: Towards a collaborative e-science platform for esa, in: J.-E. Ruiz, F. Pierfederici (Eds.), ADASS XXX, Vol. TBD of ASP Conf. Ser., ASP, San Francisco, 2021, p. 999 TBD.

[27] M. Jurić, D. Ciardi, G. Dubois-Felsmann, Lsst science platform vision document, https://ls.st/LSE-319, accessed: 2020-10-18 (2017).

[28] R. Dooley, S. R. Brandt, J. Fonner, The Agave platform: An open, science-as-a-service platform for digital science, in: Proceedings of the Practice and Experience on Advanced Research Computing, PEARC '18, Association

for Computing Machinery, New York, NY, USA, 2018, pp. 100–108. doi:10.1145/3219104.3219129.
URL https://doi.org/10.1145/3219104.3219129

[29] J. W. Nicklas, D. Johnson, S. Oottikkal, E. Franz, B. McMichael, A. Chalker, D. E. Hudak, 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, 2018, pp. 1–8. doi:10.1145/3219104.3219149.
URL https://doi.org/10.1145/3219104.3219149

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

[31] M. Milligan, Jupyter as Common Technology Platform for Interactive HPC Services, arXiv e-prints (2018) arXiv:1807.09929arXiv:1807.09929.

[32] A. M. Castronova, P. Doan, M. Seul, A general approach for enabling cloud-based hydrologic modeling using jupyter notebooks, HydroShare (2018).
URL http://www.hydroshare.org/resource/075664b0f0df4c58892cb4665e77e497

[33] M. Araya, M. Osorio, M. Díaz, C. Ponce, M. Villanueva, C. Valenzuela, M. Solar, JOVIAL: Notebook-based astronomical data analysis in the Cloud, Astronomy and computing 25 (2018) 110–117.

[34] B. Major, J. Kavelaars, S. Fabbro, D. Durand, H. Jeeves, Arcade: An Interactive Science Platform in CANFAR, in: P. J. Teuben, M. W. Pound, B. A. Thomas, E. M. Warner (Eds.), Astronomical Data Analysis Software and Systems XXVII, Vol. 523 of Astronomical Society of the Pacific Conference Series, 2019, p. 277.

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

[36] Kaggle, Notebooks, `https://www.kaggle.com/docs/notebooks`, accessed: 2020-10-18 (2018).

[37] D. Chappell, Introducing azure machine learning, A guide for technical professionals, sponsored by Microsoft corporation (2015).

[38] S. A. Russo, et al., A microservice-oriented science platform architecture, in: J.-E. Ruiz, F. Pierfederici (Eds.), ADASS XXX, Vol. TBD of ASP Conf. Ser., ASP, San Francisco, 2021, p. 999 TBD.

[39] S. Newman, Building microservices, " O'Reilly Media, Inc.", 2015.

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

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

[42] M. Zalewski, The tangled Web: A guide to securing modern web applications, No Starch Press, 2012.

[43] M. Zalewski, Browser security handbook, `https://code.google.com/archive/p/browsersec/wikis/Main.wiki`, accessed: 2021-10-18 (2009).

[44] M. Squarcina, M. Tempesta, L. Veronese, S. Calzavara, M. Maffei, Can i take your subdomain? exploring same-site attacks in the modern web, in: 30th {USENIX} Security Symposium ({USENIX} Security 21), 2021, pp. 2917–2934.

[45] JupyterHub, Security overview, `https://jupyterhub.readthedocs.io/en/1.4.2/reference/websecurity.html`, accessed: 2021-10-18 (2016).

[46] S. Bertocco, D. Goz, L. Tornatore, A. Ragagnin, G. Maggio, F. Gasparo, C. Vuerli, G. Taffoni, M. Molinaro, INAF Trieste Astronomical Observatory Information Technology Framework, in: R. Pizzo, E. R. Deul, J. D. Mol, J. de Plaa, H. Verkouter (Eds.), Astronomical Society of the Pacific Conference Series, Vol. 527 of Astronomical Society of the Pacific Conference Series, 2020, p. 303.

[47] F. Tinarelli, S. Zorba, C. Knapic, G. Jerse, The authentication and authorization inaf experience, Astronomical Data Analysis Software and Systems XXVII 522 (2020) 727.

[48] C. Tasse, B. Hugo, M. Mirmont, O. Smirnov, M. Atemkeng, L. Bester, M. Hardcastle, R. Lakhoo, S. Perkins, T. Shimwell, Faceting for direction-dependent spectral deconvolution, Astronomy & Astrophysics 611 (2018) A87.

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

[50] M. Katevenis, R. Ammendola, A. Biagioni, P. Cretaro, O. Frezza, F. Lo Cicero, A. Lonardo, M. Martinelli, P. S. Paolucci, E. Pastorelli, F. Simula, P. Vicini, G. Taffoni, J. A. Pascual, J. Navaridas, M. LujÃ¡n, J. Goodacre, B. Lietzow, A. Mouzakitis, N. Chrysos, M. Marazakis, P. Gorlani, S. Cozzini, G. P. Brandino, P. Koutsourakis, J. van Ruth, Y. Zhang, M. Kersten, Next generation of exascale-class systems: Exanest project and the status of its interconnect and storage development, Microprocessors and Microsystems 61 (2018) 58–71. doi:https://doi.org/10.1016/j.micpro.2018.05.009.
URL https://www.sciencedirect.com/science/article/pii/S0141933118300188

[51] G. Taffoni, L. Tornatore, D. Goz, A. Ragagnin, S. Bertocco, I. Coretti,

M. Marazakis, F. Chaix, M. Plumidis, M. Katevenis, R. Panchieri, G. Perna, Towards exascale: Measuring the energy footprint of astrophysics hpc simulations, in: 2019 15th International Conference on eScience (eScience), 2019, pp. 403–412. doi:10.1109/eScience.2019.00052.

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

[53] D. Goz, G. Ieronymakis, V. Papaefstathiou, N. Dimou, S. Bertocco, F. Simula, A. Ragagnin, L. Tornatore, I. Coretti, G. Taffoni, Performance and energy footprint assessment of fpgas and gpus on hpc systems using astrophysics application, Computation 8 (2) (2020). doi:10.3390/computation8020034.
URL https://www.mdpi.com/2079-3197/8/2/34

[54] G. Taffoni, U. Becciani, B. Garilli, G. Maggio, F. Pasian, G. Umana, R. Smareglia, F. Vitello, CHIPP: INAF Pilot Project for HTC, HPC and HPDA, in: R. Pizzo, E. R. Deul, J. D. Mol, J. de Plaa, H. Verkouter (Eds.), Astronomical Society of the Pacific Conference Series, Vol. 527 of Astronomical Society of the Pacific Conference Series, 2020, p. 307.