Deliverables

Deliverable Number	**D24.4**
Deliverable Title	**Software packages for the selected experiment use cases ready to install and run**
Lead Beneficiary	**PSI**
Type	**Other**
Dissemination Level	**Confidential, only for members of the consortium (including the Commission Services)**
Due date of delivery	**Month 18**

Premise

The present deliverable describes the packaging and deployment strategies for the five scientific data analysis frameworks selected in and described in deliverable D24.3.

Description of Work

WP24: Demonstrator of a Photon Science Analysis Service

1.Introduction

The present document presents a comparison of different software packaging strategies, their suitability in different compute environments and intrinsic advantages and disadvantages. We aim to deploy analysis frameworks in distributed compute environment like the EOSC-Hub. The focus is hence clearly on provisioning of software in Linux environment. Some of the packages might possibly be used on different platforms via containerization or virtualization, or are natively supporting these platforms. We will however not provide, test or support packages of the selected frameworks other than for Linux platforms. This might appear inconvenient for some of the users, provides however additional incentives to use scalable cloud hosted service rather than (poorly maintained) local installations on personal devices.

Regardless of the particular scientific framework the goal has been to produce suitably packaged application which

- Can be deployed and used in almost arbitrary environments
- Have a clear description of the build process suitable for automated deployment
- Can save efforts at individual facilities and/or individual users in need of local installations
- Provide a very user friendly access to applications

Deployment of complex data analysis frameworks and tool-chains is a common task at research facilities and frequently a major hurdle for scientists, hampering rapid data analysis and publications. Simple assembly of integrated and deployable applications reduces the research institutes (RI) efforts as well as it accelerates the scientific process.

The selected use case applications have been implemented as deployable packages, as preconfigured virtual machines or as containers. Virtual machines or containers provide encapsulated user environments, which can be archived together with the experimental data, thereby capturing valuable provenance data and strongly supporting reproducibility of the original experiment and data analysis workflows.

2. Selected Data analysis application

We had selected five most suitable scientific data analysis framework covering very different scientific areas and workflows. Each of the use cases also poses different challenges for deployment, coming with different licenses and compute models. The frameworks have been described in detail in deliverable D2.4. The following table provides a quick overview of the selected use cases:

	CrystFEL	**Ptycho Shelves**	**Savu**	**pyFAI**	**PyMca**
Site	DESY	PSI	Diamond	ESRF	ESRF
License	GPL	See [1]	GPL/Apache	MIT	MIT
Licenses of Dependencies[2]	Standalone	Matlab>R2017b cuda, hdf5, gcc>6.2	MPI, cuda, fftw	Hdf5, silx, Fabio, fftw, PyQt	Fisx, PyQt
Available Recipes	rpm, docker, singularity, cvmfs	zip, docker, singularity	docker, singularity	deb, rpm	deb, rpm, docker, singularity
Original deployments	Source	Local at PSI	Source	Source, Conda, pip	Source, pip, conda
Availabe Repositories	Local git	Local Web	Github	Github, conda, debian	Github, conda, debian
Software Catalogue[3]	yes	?	?	yes	yes

For example pyFAI is providing recipes for building debian packages, is readily available from conda and debian repositories, and as sources from github. Dependencies listed are only those not always available from standard repositories.

[1] https://www.psi.ch/sls/csaxs/software
[2] Licenses of products the particular use depends on. Without these products/licenses the use case can not be executed.
[3] https://software.pan-data.eu/

3. Packaging and deployment strategies

Software deployment

Software deployment is a task common to all facilities, and equally a task of users' daily workflow. The individual software component is usually not very difficult to prepare for on-site deployment, though some applications are rather complex and embedded in a complex environment of heterogeneous dependencies. The task is multiplied by the number of software components, operating systems to support, and infrastructures to serve. The total effort spent in software deployment is not marginal and the result often far from being satisfying.

There are meanwhile several ways of preparing and deploying software in large-scale albeit heterogeneous infrastructures. Each has its own benefits and disadvantages. We are hence looking for ways to minimize efforts for both facilities and user communities while maximizing synergies.

The most scalable software deployment tools for Linux based operating systems are briefly outlined below.

Standard Packages

The most common way (at least for widely used applications) to deploy software is the encapsulation into standard packages (deb, rpm, dmg, and so on). Packages are usually signed, deposited in standard repositories and can readily be installed on any system with an internet connection. The repositories can easily be federated. Package manager usually allow harvesting packages from an arbitrary number of repositories while prioritizing some repositories (e.g. in-house) over others.

Pros

- Most importantly well maintained packages will guarantee full functionality at run-time.
- Dependencies and conflicts are tracked, so that a purely package based system tend to be highly self-contained.
- The repositories contain all meta-information like licenses which makes it straightforward to obtain a full software catalogue of a running system.
- Packages are usually signed allowing tracking the origin of a package.
- Packaging is done with simple to very complex build-instructions. In most cases updates to newer versions are very simple to realize.
- Deployment and mass-deployment in arbitrary environments is un-problematic in particular with management systems like puppet/foreman.
- Turn-over time to deploy packages and updates is short.
- Package size are small (in most cases)
- Efforts to build and deploy applications are moderate
- It's simple to combine an arbitrary number of package sources
- Anyone with a Linux system at home can re-use the packages (if open source)
- Packages can be used to deploy the application in docker or singularity

Cons

- Packages are OS dependent, e.g. a package for Debian systems cannot so easily be transformed into a package for RedHat systems, without loss of some functionality and consistency. Occasionally the dependency is at the level of minor releases, so that applications built for RedHat 7.3 won't run on RedHat 7.4 (i.e. changes in the crypto/ssl environment).
- Application packages in in-house repositories are fully dependent on the base-system environment. Updates of the base-system can break an application, or lead to conflicts between packages.
- Federating various independent repositories can easily lead to an inconsistent system. In particular the dependencies need to be verified and updated regularly.
- Even though packages can be installed in a user environment, the install process and execution of post- and pre-installation-instructions are performed with full admin rights. Any package coming from a foreign repository needs to be carefully validated.
- There are no good mechanisms to control access to software under special license conditions (e.g. commercial applications like Matlab; non-redistributable software like XDS; software requiring a personalized license agreement like orca). In essence one would need to setup several repositories for each class of non-open applications and control access to repositories. Access to a repository also allows mirroring the entire repository which makes it practically impossible to prevent unwarranted re-distribution.

CVMFS

The CernVM File System (CVMFS) is designed as a scalable, reliable and low-maintenance software distribution service. The main focus is clearly the distribution of open source software. The filesystem has smart mechanisms to minimize the size of the repository, to make distribution of files fast and scalable across an essentially unlimited number of clients. CVMFS is implemented as a POSIX-compliant read-only filesystem in user space, and it hence entirely un-intrusive.

Pros

- Single point of deployment
- Easy mechanisms for federation
- Openly accessible. Anyone can mount cvmfs, so offers scalable deployment for users as well as facilities
- cvmfs comes with smart deduplication mechanisms reducing the
- cvfms supports caching to leverage so load on servers, and reduce the network traffic
- It's entirely non-intrusive
- Uploads and updates are performed by cvmfs admins making the deployment traceable.

Cons

- cvmfs hosted applications need to be done per operating system (like for regular packages), to satisfy basic dependencies (e.g. glibc).
- cvmfs caches are only efficient when applications are used multiple time
- caches are limiting the use of very large binary applications (e.g. container)

- setup relies on environment-scriplets. The environments can easily interfere with base-system operation (e.g. different python-version). This is not a problem for DaaS installations, but can become difficult to support in local or user environments.
- there is no dependency tracking (unless explicitly embedded into environment-scriplets)
- the execution of environment scriptlets has the side-effect of reducing dramatically the performance of the system when it comes to launching programs (due to the evalution of complicated $PATH, $LD_LIBRARY_PATH, ...) which are accessed remotely (and not cached).
- It's perfect for open source software deployment. Deployment of commercial software, or any non-re-distributable software is not impossible, but cumbersome and not actually the scope of cvmfs.

Containers

There are meanwhile several containerization systems available (e.g. docker, shifter, singularity, lxc, openvz), which all aim to facilitate software deployment in heterogeneous and/or distributed environments. Shifter for example is tailored for HPC environments; docker is more focusing on scalable service deployment whereas singularity is oriented towards a pure userspace implementation.

From the perspective of software deployment differences between the different container systems are comparably small: one needs a recipe to build container, a registry to deploy the images and a bit of system installation (daemon) to instantiate the service. We will hence focus on docker and singularity as the most widely used systems at the user facilities.

Pros

- Self-contained environment.
- Easy to deploy in arbitrary distributed environments
- Highly scalable
- Supports reproducibility and provenance. Storing and publishing a container will make the application and the results highly reproducible.
- OS agnostic (docker). A docker container should run on pretty much any (recent enough) operating system. Singularity relies more on the host environment and might be less portable.
- Very effective caching and layering mechanisms

Cons

- Encapsulation is great for service deployment. At the instruments the standard applications are very often augmented by custom add-ons. Importing the add-ons into the container might break encapsulation and OS agnostics.
- Caching and layering are only effective when containers share a common base. Otherwise container might become rather heavy-weight. Storing container in cvmfs circumvents the deduplication mechanism strongly increasing storage and traffic requirements.
- User namespaces in docker might pose problems when importing filesystems and raise security concerns.

- Authentication inside a container (e.g. for data access) can easily be hijacked by admins on the host system. Requires complete trust of the service provider by the user.

Jupyter Notebooks

A Jupyter Notebook does - strictly speaking - not provide a software deployment mechanism. It allows however to document the data analysis procedure in a step-by-step manner, to embed derived data, and to create snapshots of the current or final state of the procedure. As such notebooks provide an interface to DaaS Services and at the same time can become integral components of the software deployment. The typical JupyterHub would instantiate a notebook by launching a docker container with the requested kernels and software environment. The docker container serving the notebook would naturally need to fulfil the requirements (dependencies) of the notebook application. The most consistent way would then to deploy the requirements as regular packages (rpm, deb) inside the docker container, but cvmfs could equally serve the purpose – as long as the applications and dependencies are re-distributable. Providing data analysis as a service via notebooks a frontend will hence at least strongly favour pre-packaging of the applications and dependencies.

Deployment in cloud environments like EOSC

All of the above mentioned packaging strategies can easily be used in arbitrary cloud environment like the EOSC. cvmfs has the slight advantage that additions and updates can more easily be deployed cloud-wide since all modifications become available on all nodes or instances (like docker images) with a single deployment (up to cache-invalidation), unless requiring updates of system-packages. Being entirely non-intrusive, cvmfs installations need to be tested for functionality, but much less for security vulnerabilities (unless providing services operated under elevated privileges).

Deployment close to the instrument

As soon as the applications are integral part of experiments (which is valid for all use cases) a deployment with cvmfs installations is not very appealing. It's crucial for experiments to rely on as little external dependencies as possible. For instrument related and controlled, automated deployment and system management with substantial external dependencies, there currently is hardly a way around pre-packaged installations, either in form of regular packages (rpm, deb), or containers. Though the focus of the work package is on data analysis as a service in remote, distributed environments, it's clear that for the major analysis frameworks, pre-packaged installations can safely be expected to be available.

Summarized

In the following table we aimed to summarize the different methods, and score with respect to various aspects. The scores are quite subjective, but it's apparent that there is not the ultimate packaging tool which outperforms and covers all aspects of deployment. For actually sharing "packages" and minimizing efforts at facilities, for both open source and closed commercial applications, the sharing of build-recipes (for container, packages, conda, cvmfs) is by far the most unproblematic way. Publishing recipes on git together with appropriate web-hooks enables then automated builds, and triggering of automated deployment. Tools are readily available for all major packaging methods, and container builds.

	Packages (score: 15)	CVMFS (12)	Docker [Registry] (14)	Singularity [Registry] (14)	Build Recipes (19)	Notebook Hub (13)
Federation	Simple. Conflicts!	Simple. No conflicts.	Simple. Conflicts!	Simple. Conflicts!	Simple.	Not that simple
Re-use of deployment across labs (open source)	high. requires Validation.	high. Requires Validation.	High. requires Validation.	High. requires Validation.	Medium	High.
Re-use of deployment by users (open source)	Quite simple.	Quite simple	Medium. Requires local docker	Medium. Requires local singularity	Medium. Requires a bit of knowledge	High.
Re-use of deployment across labs (licensed sw)	Not simple at all	no	no	no	yes	Depends on the application
Re-use of deployment across user (licensed sw)	no	no	Possible with special registry	Possible with special registry	yes	Possible with special registry
OS agnostic	no	no	yes	yes	yes	somewhat
Permission	Install with admin rights	No system attachment other than mount	Usually requires admin privileges inside container	Admin privileges on build engine	None required. Building and deploying see registries	None required.
Security	Safe. Validation!	Safe. Non-intrusive	Medium to high. Breakouts risky	High. Runs in user space	Container security applies	Fairly safe. Runs in user spaces.
Cloud usability	Good if images are under your control	Good	Good.	Good. If cloud supports it	Good. If build engine available	Ok in container
HPC usability	Good	Medium	Good.	Good.	Good. If build engine available	Ok in container
Mass deployment	Good	Good	Good	Good	Good	Ok in container
Analysis Reproducibility	Poor	Poor	Ok	Good	Ok	Ok in container
Dependency tracking	Good	Has to be self-contained.	Good. Depends on container-build	Good. Depends on container-build	Good. Depends on container-build	Ok when using container
Deployment speed	Good	Good	Medium. Depends on daemons cache	Medium to slow. Big images.	Slow. Need to build and deploy.	Medium.
Update speed	Good	Medium	Good for base images widely used	Good for base images widely used	Slow	Medium
Effort per application	Medium. Validation.	Low-Medium	Low-Medium	Low-Medium	Low	High. Requires at least some coding
Total effort for lab	Medium	Low-Medium	Low-Medium	Low-Medium	Low	High
Total effort for user	Low	Low	Medium	Medium	High	Low

🟩	Score: +2	Favourable
🟩	Score: +1	Ok
🟦	Score: +0	Has some issues
🟧	Score: -1	Poor

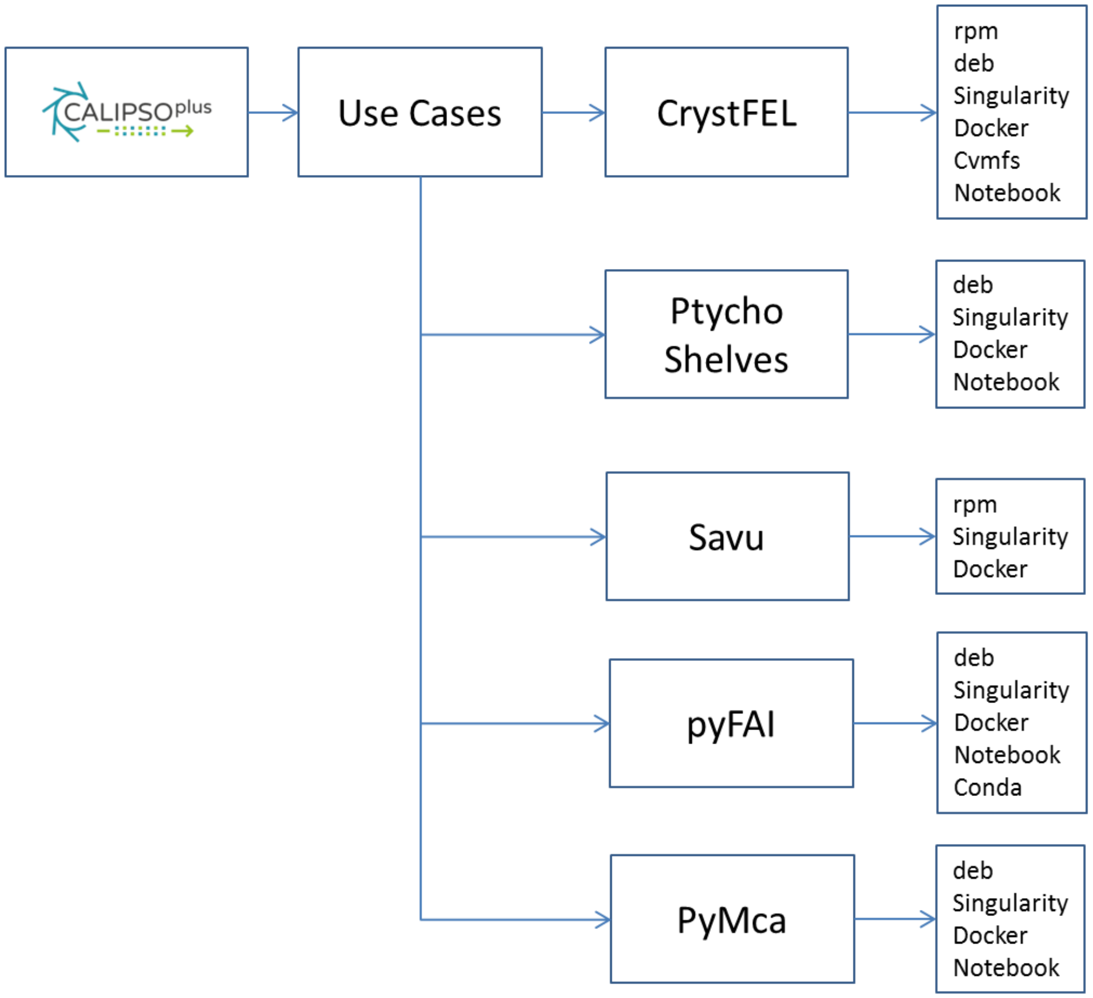

Figure 1: proposed layout for publication of recipes and packages on github/calipsoplus

4. Provisioning packages

As described above the most suitable deployment strategy depends on a number of factors, and is continuously evolving as the containerization, storage and cloud platforms evolve. In most cases the same recipe can be recycled for different forms of packaging with minor modifications. For example a conda recipe can easily be used to package an application as a docker or singularity container.

The following table lists the currently available recipes and packages

	CrystFEL	Ptycho Shelves	Savu	pyFAI	PyMca
Docker	Yes (4)	Yes (1)	Yes	Yes (2)	In preparation
Docker recipe	Yes (3)	Yes (1)	Yes	Yes (1)	Yes (1)
Singularity	Yes (3)	Yes (1)	Yes	Yes (1)	Yes (2)
Singl. recipe	Yes (3)	Yes (1)	Yes	Yes (1)	Yes (1)
CVMFS	In preparation	In preparation	in preparation	In preparation	In preparation
cvmfs recipe	Yes (1)	Yes (1)	Yes (1)	Yes (1)	Yes (1)
RPM recipe	Yes (3)	In preparation	In preparation	Yes (1)	Yes (1)
RPM package	Yes (2)	In preparation	In preparation	Yes (3)	Yes (1)
DEB recipe	In preparation	In preparation	In preparation	Yes (1)	Yes (1)
DEB package	Yes (1)	In preparation	In preparation	Yes (3)	Yes (1)
Notebook	Yes (3)	No	-	-	-
Other	-	-	-	Conda, pypi	Conda, pypi

Table 1: Available packages. Numbers in brackets indicate the number of sites where a package or recipe has been tested or is being used in productions.

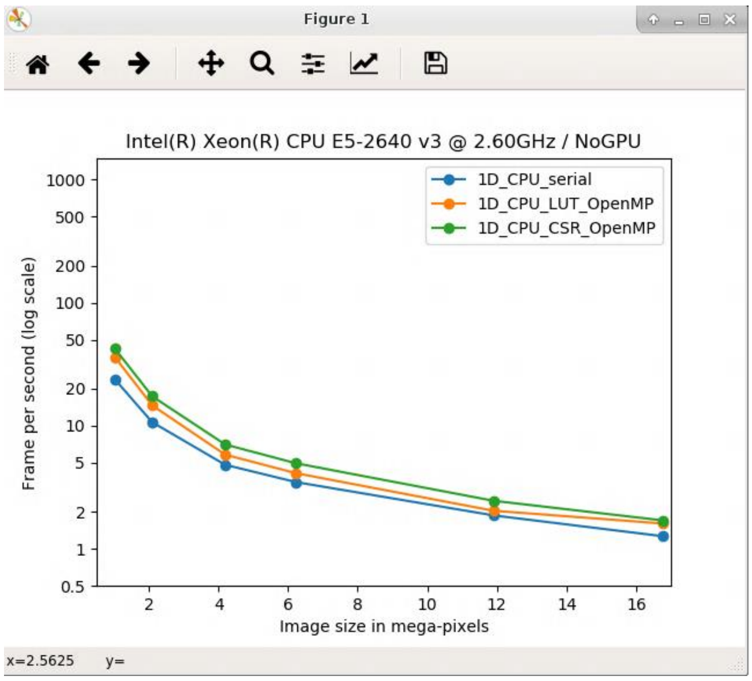

Figure 2: Running pyFAI from singularity: singularity run --app pyFAI-benchmark pyFAI.simg

The following table describes the deployment and availability of the five scientific data analysis frameworks on different sites or platforms.

	CrystFEL	Ptycho Shelves	Savu	pyFAI	PyMca
Project Partner in JRA2					
ALBA				Production	Production
DESY	Production	Yes		Production	Production
DIAMOND			Production	Production	
ELETTRA					
ESRF				Production	Production
PSI		Production		Production	
SOLEIL					
Other Project Partner					
Eur.XFEL	Production				
HZB					
MAX IV				Production	
External Infrastructures					
EOSC	Yes			Yes	
ESS	Yes				
HPC	Production				

Table 2: Yes: has successfully been tested on site. Production: is used in production at site

The availability of a "remote desktop environment" as a docker container[4], developed for the architecture blueprint D24.2, allows to execute all of the use cases in such an encapsulated desktop environment. We can hence guarantee that an identical framework can be offered at each of the facilities, with fully functional use case applications. In addition it will allow users executing almost any application of choice available as a singularity package.

In addition to the encapsulated environment, all five use cases can readily be executed on arbitrary platforms (meeting the minimal hardware requirements) which extends the capabilities to multi-host environments and the utilization of hard accelerators (i.e. GPGPUs).

[4] https://hub.docker.com/r/danielguerra/ubuntu-xrdp/

5. Next steps

All applications required to execute the use cases on (almost) arbitrary platforms have been packaged in all relevant formats. From a user perspective, the singularity packages are most convenient to execute, from an administrators perspective regular packages (deb, rpm) are easiest to control and deploy in managed environments. In particular for Docker and Singularity automated builds have been enabled via github and the registries for docker and singularity, respectively. Only exception is the Ptychoshelves framework, which is currently awaiting publication and release of the code.

However, the packages need to be tested at a larger number of sites and a larger number of actual users, as envisaged for the next deliverable. Particularly important will be the integration into the facilities portals (selectable via a central hib) as outlined by the architecture blueprint. To facilitate local testing at all sites we aim to establish a registry of packages (repository) for package sharing. Recipes are being published on github, and are readily available for further tests.

Some sites will want to add additional packages which are used locally. All sites are encouraged to add their major Packages to the PaNdata software catalogue (https://software.pan-data.eu/) and publish recipes on github (https://github.com/Calipsoplus).