



A Management Framework for Consolidated Big Data and HP

Project number: 45924

Deliverable 5.2 - Intermediate report on the integration, validation and pilot activities

Date 31 March 2022

Activity 5 - Integration, Experimental Validation and Pilot

Version Intermediate

Disclosure Public

Authors: Amit Ruhela (TACC), João Paulo (INESC TEC), Júlio Silva (Wavecom), Mariana Miranda (INESC TEC), Mário David (LIP), Ricardo Macedo (INESC TEC), Richard Todd Evans (TACC), Samuel Bernardo (LIP), Stephen Harrell (TACC), Tiago Gonçalves (LIP), Zacarias Benta (LIP)

Reviewers: Bruno Antunes (Wavecom), João Paulo (INESC TEC)

Partners



Funding



Table of Content

Table of Content	2
Executive Summary	4
Glossary	5
1. Introduction	6
2. Platform components integration and validation	7
2.1. Gitlab workflow	8
2.1.1. Container registry	12
2.2. Jenkins pipeline as code	14
2.3. GitLab Runners	18
2.3.1. Steps to configure a runner	19
2.3.2. How to use runners in the pipelines	20
2.4. Kubernetes and Jenkins deployment	21
2.4.1. Kubernetes	21
2.4.2. Jenkins	22
2.4.3. Authentication	23
3. Pilot activities	25
3.1. Testbed definition	25
3.1.1. Access levels	25
3.2. Monitoring component	27
3.2.1. HECTOR	28
3.2.2. MONICA	28
3.2.3. CI/CD pipeline: Monitoring component	29
3.3. Virtualization component	31
3.3.1. Virtualization Repository	31
3.3.2. Virtualization Controllers and Executors	32

3.4. Software-Defined Storage component	34
3.4.1 PAIO	34
3.4.2 PADLL	36
3.4.3 Cheferd	38
4. Conclusion	41
References	42

Executive Summary

High-Performance Computing (HPC) infrastructures are increasingly sought to support Big Data applications, whose workloads significantly differ from those of traditional parallel computing tasks. This is expected given the large pool of available computational resources, which can be leveraged to conduct a richer set of studies and analysis for areas such as healthcare, smart cities, natural sciences, among others. However, coping with the heterogeneous hardware of these large-scale infrastructures and the different HPC and Big Data application requirements raises new research and technological challenges. Namely, it becomes increasingly difficult to efficiently manage available computational and storage resources, to provide transparent application access to such resources, and to ensure performance isolation and fairness across the different workloads.

The BigHPC project aims at addressing these challenges with a novel management framework, for Big Data and parallel computing workloads, that can be seamlessly integrated with existing HPC infrastructures and software stacks. Namely, the project will develop novel monitoring, virtualization, and storage management components that can cope with the infrastructural scale and heterogeneity, as well as, the different workload requirements, while ensuring the best performance and resource usage for both applications and infrastructures.

These components will be integrated into a single software bundle that will be validated through real use-cases and a pilot deployed on both TACC and MACC data centers. Also, the proposed framework will be provided as a service for companies and institutions that wish to leverage their infrastructures for deploying Big Data and HPC applications.

This deliverable is focused on the BigHPC's integration activities.

Glossary

API	Application Programming Interface
CI/CD	Continuous Integration/Continuous Delivery
CLI	Command Line Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
CLI	Command Line Interface
DevOps	Development and Operations
DSL	(Jenkins) Domain Specific Language
FOSS	Free and Open-Source Software
GitOPS	Pattern of configuration as code putting together Git and CI/CD tools
I/O	Input/Output
INCD	National Distributed Computing Infrastructure
INESC TEC	Institute for Systems and Computer Engineering, Technology and Science
IT	Information Technology
LIP	Laboratory of Instrumentation and Experimental Particle Physics
MACC	Minho Advanced Computing Center
MPI	Message Passing Interface
PaC	Pipeline as Code
RAM	Random-Access Memory
RDMA	Remote Direct Memory Access
REST	Representational State Transfer
SCM	Software Configuration Management
SDS	Software-Defined Storage
SIG	(Kubernetes) Special Interest Group
SQA	Software Quality Assurance
TACC	Texas Advanced Computing Center
YAML	Yet Another Markup Language

1. Introduction

The gap between the developers' environment and the complexity of BigHPC infrastructure brings a challenge that can be solved by joining together teams that normally have different perspectives. The developers and infrastructure managers provide the experience required to understand the software delivery and operation tasks, but lack the contract to provide the deployment. The DevOPS methodology focuses on the deployment of developed software and is elected as the main practice during this activity, fixing the missing brick between development and Information Technology (IT) operations.

Components integration, the next section, had the collaboration of Wavecom for defining the prototype for the Pipeline as Code (PaC) and the PostgreSQL database endpoint. As described in Section 3.2, the monitoring components development and testing provided a way to define the Gitlab template for other projects. To test the template, INESC TEC also created a PaC for SDS, detailed in Section 3.4, using the previously defined Gitlab template.

UT Austin and TACC teams had an important role in usability and performance evaluation, for defining the required environments for the pilot and providing feedback from user experience. This task will include most of these results in the next phase of the project. The requirements and the configurations for the HPC environments are presented in Section 3.3.

All tasks will converge into the pilot deployment. Regarding the integration, usability and performance evaluation, the pilot provides the full-fledged solution where the BigHPC platform backend meets the frontend. GitOPS enters into practice over team collaboration using the Gitlab platform, software quality practices adoption and supporting the research of testbed solutions to abstract the real environments of HPC infrastructures. The pilot development is dependent on the components development that are expected to be ready in September 2022. Section 2 shows the current implementations and research work aimed for the integration of all components and pilot deployment.

2. Platform components integration and validation

The BigHPC project is bringing together innovative solutions to improve the monitoring of heterogeneous HPC infrastructures and applications, the deployment of applications and the management of HPC computational and storage resources. It aims as well to alleviate the current storage performance bottleneck of HPC services. In order to keep all development tasks tracking in a common path, some good practices are needed to get a shorter development life cycle and provide continuous delivery and deployment with software quality.

GitOPS is a way to implement the continuous deployment and software quality best practices. Using a GitOPS framework solution put into practice the methodology advantages, such as:

- deploy faster and more often
- easy and fast error recovery
- easier credential management
- well documented deliveries with complete history of every change made to the system
- share knowledge between teams with great commit messages

As a result, everybody would be capable of reproducing the tough process of changing the infrastructure and also easily find examples on how to set up new systems.

In this work, we are creating the git workflow being adopted for application development and the tools to answer the three components of GitOPS:

- infrastructure as code
- merging changings together
- deployment automation

We will show the technical capabilities and advantages of using this approach, adopting good practices and pursuing fast innovation delivery. Developers should be kept focused on the continuous development of the software. The IT operations team is responsible for the infrastructure management.

2.1. Gitlab workflow

As a starting point, the developers can use the web interface provided by Gitlab, to prepare the environment to start working, by following DevOPS practices and contributing to the GitOPS pilot deployment. For that, we prepared a quick guide to upload project code into gitlab, depicted by the following steps:

- go to the entry page <https://gitlab.com/bighpc>;

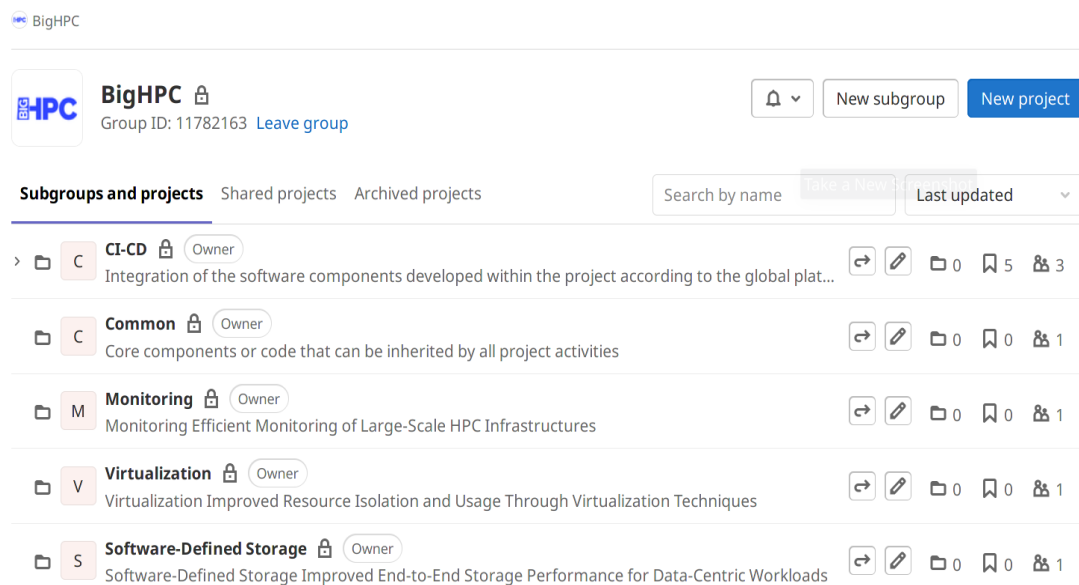


Figure 2.1 - BigHPC Gitlab Group: entry page

- after authentication, users must select the group where the project should be created,
- then they can select the "New project" button to create a new repository,
- a new view opens with 4 options but here, the developer should only care for the blank project and import project options,
- The user must select the blank project option when there only exists a local repository or select import project if the repository was already published remotely;

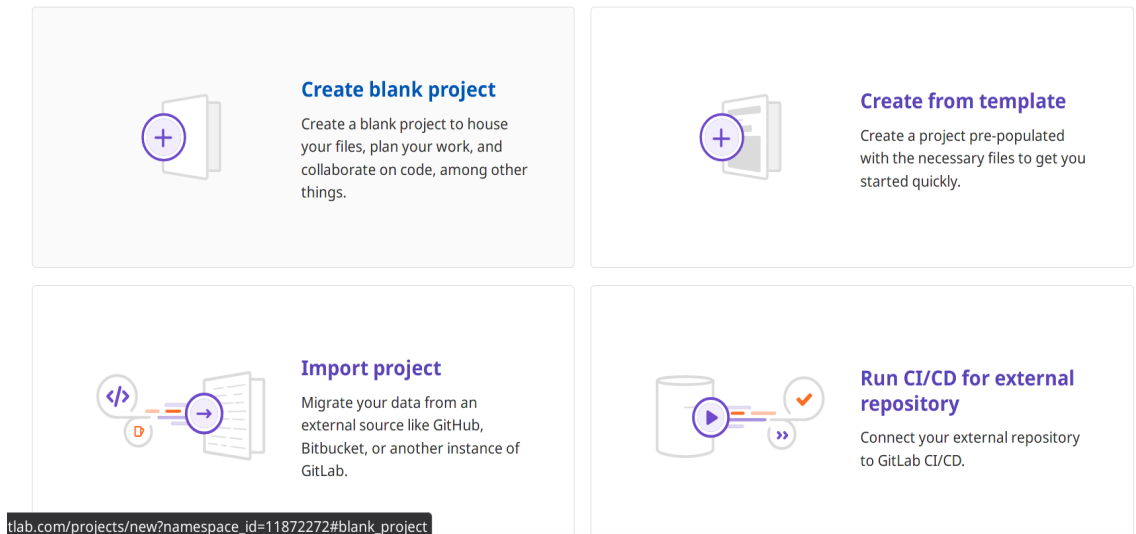


Figure 2.2 - Create a new project on Gitlab

Create a blank project:

- users fill the form as desired, where the project name is the label that appears in the web interface and project slug is the url unique identifier that is appended in the path to the group folder;

New project > **Create blank project**

Project name

My awesome project

Project URL Please fill out this field. **Project slug**

`https://gitlab.com/` `highpc/monitoring` `my-awesome-project`

Want to house several dependent projects under the same namespace? [Create a group](#).

Project description (optional)

Description format

Visibility Level ⓘ

☒ Private
Project access must be granted explicitly to each user. If this project is part of a group, access will be granted to members of the group.

☐ **Initialize repository with a README**
Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.

[Create project](#) [Cancel](#)

Figure 2.3 - Gitlab blank project form

- afterwards, if an empty repository is created (not selecting the readme checkbox), a page with instructions will appear. If the repository was created with a README file the developer only needs to clone the repository with ssh or https.

Import project:

- the developer can select any of the available options to “*Import project from*”, but we will focus here on the generic solution using “*Repo by URL*”;

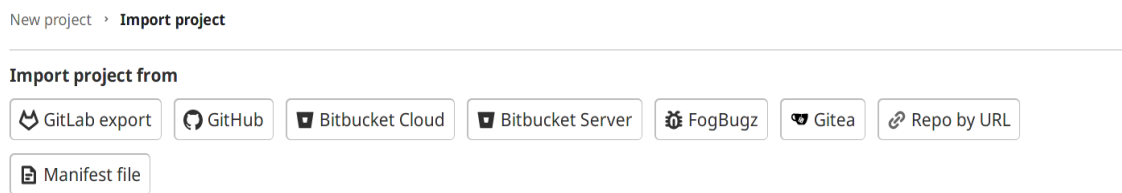
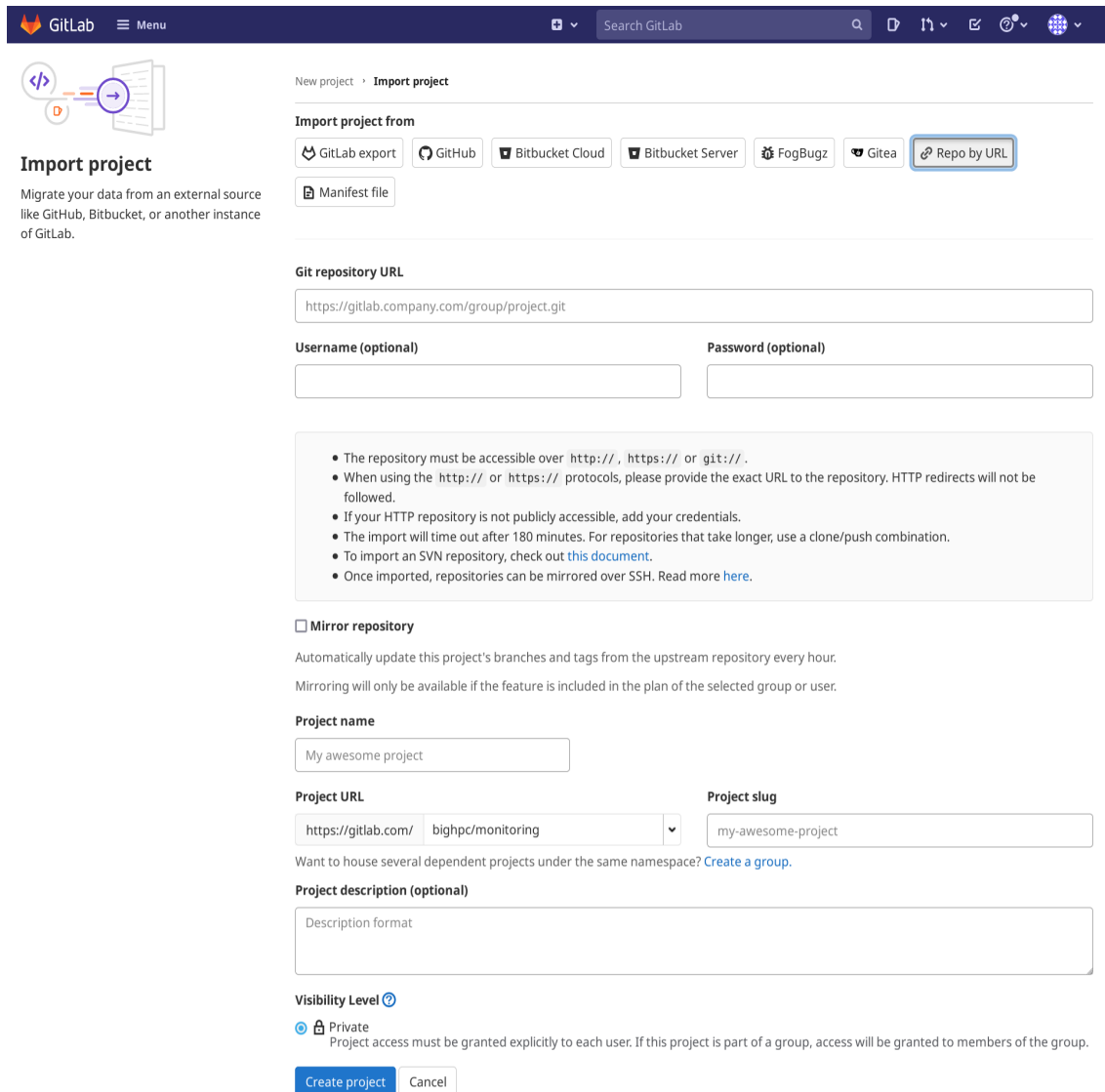


Figure 2.4 - Import a remote project on Gitlab

- after clicking the button "Repo by URL" a form is opened (shown below);



Import project

Migrate your data from an external source like GitHub, Bitbucket, or another instance of GitLab.

New project › Import project

Import project from

GitLab export GitHub Bitbucket Cloud Bitbucket Server FogBugz Gitea **Repo by URL**

Manifest file

Git repository URL

https://gitlab.company.com/group/project.git

Username (optional) Password (optional)

- The repository must be accessible over `http://`, `https://` or `git://`.
- When using the `http://` or `https://` protocols, please provide the exact URL to the repository. HTTP redirects will not be followed.
- If your HTTP repository is not publicly accessible, add your credentials.
- The import will time out after 180 minutes. For repositories that take longer, use a clone/push combination.
- To import an SVN repository, check out [this document](#).
- Once imported, repositories can be mirrored over SSH. Read more [here](#).

☐ Mirror repository

Automatically update this project's branches and tags from the upstream repository every hour.

Mirroring will only be available if the feature is included in the plan of the selected group or user.

Project name

My awesome project


Project URL Project slug

https://gitlab.com/ bighpc/monitoring my-awesome-project

Want to house several dependent projects under the same namespace? [Create a group](#).

Project description (optional)

Description format

Visibility Level 

☒ Private

Project access must be granted explicitly to each user. If this project is part of a group, access will be granted to members of the group.

Create project Cancel

Figure 2.5 - Repo by URL form to import a remote project into Gitlab

- One should place there the git repository URL using the https protocol (to sync also git files if needed), while providing the corresponding username and password, in case the repository is private.
- When the URL is added to the form, the project name and project slug will be automatically filled.
- finally one must select the button "Create project", which will start the import process

2.1.1. Container registry

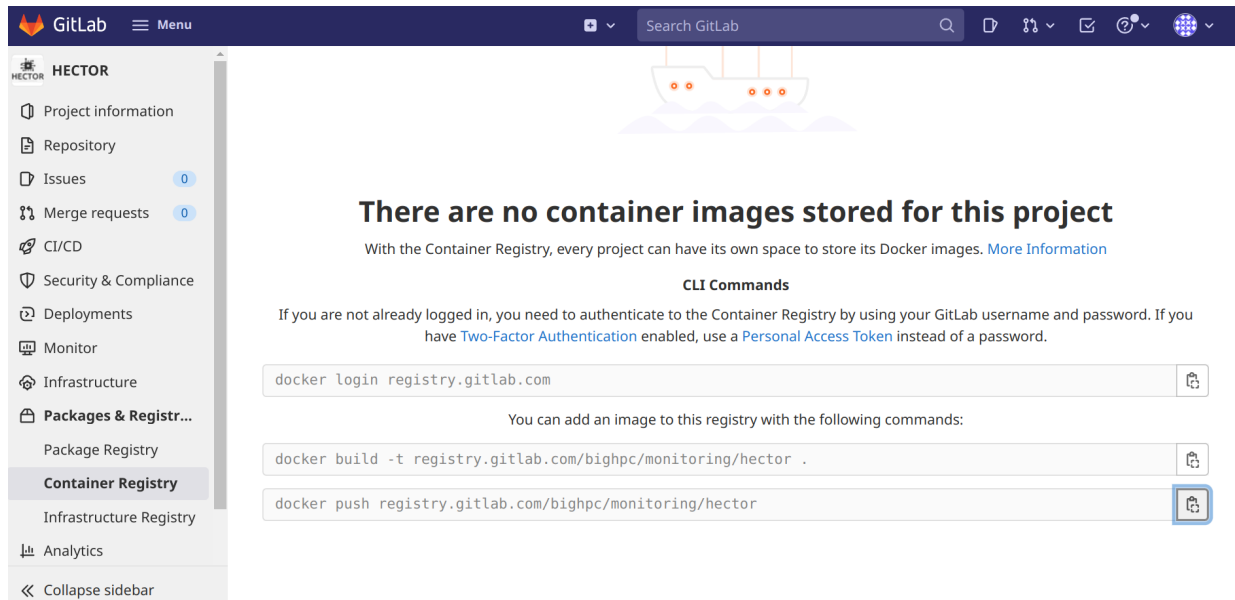


Figure 2.6 - Gitlab container registry

Gitlab also provides a container registry organized by groups and projects. Each image should be placed over each project with the following syntax:

<registry URL>/<namespace>/<project>/<image>

Example:

registry.gitlab.com/bighpc/monitoring/hector/hector:latest

Starting from the project level, there are three additional levels (namespaces). Taking into account the defined testbeds, it is possible to define a specific namespace for each one in the container registry level, without requiring the creation of additional groups.

<registry URL>/<namespace>/<project>/<testbed>/<image>

By applying the testbed definitions, the previous example would be translated into the following scheme:

- development:

registry.gitlab.com/bighpc/monitoring/hector/dev/hector:latest

- preview:
`registry.gitlab.com/bighpc/monitoring/hector/pre/hector:latest`
- production:
`registry.gitlab.com/bighpc/monitoring/hector/pro/hector:latest`

Private repository visibility is also extended to the associated container registry. The authorization has two possible implementations for secrets management; i.e., when accessing outside or inside a CI/CD context.

To access the docker registry inside a private repository from the developer workstation, a [personal access token](#)¹ or [deploy token](#)² must be used. Both of these require the minimum scope to be:

- For read (pull) access, read_registry.
- For write (push) access, write_registry.

In the next step, one would run the following command to authenticate:

```
docker login registry.gitlab.com -u <username> -p <token>
```

It is also possible to configure a credential helper to avoid renewing the credentials every time a docker command runs. Thus, one of the [available programs](#)³ can be selected.

Gitlab CI/CD provides the variables [CI_REGISTRY_USER](#)⁴ and [CI_REGISTRY_PASSWORD](#)⁵, to access to the container registry. The variable `CI_REGISTRY_PASSWORD` has the same value as [CI_JOB_TOKEN](#)⁶ that is generated by Gitlab when a job starts. `CI_REGISTRY_USER` is a gitlab-ci-token that is the user associated with the CI token.

After retrieving the authenticated session, docker commands could be run from the pipeline using a [special docker image](#)⁷ that allows running docker commands. Using a defined service

¹ https://docs.gitlab.com/ee/user/profile/personal_access_tokens.html

² https://docs.gitlab.com/ee/user/project/deploy_tokens/index.html

³ <https://github.com/docker/docker-credential-helpers>

⁴ <https://docs.gitlab.com/ee/ci/variables/>

⁵ <https://docs.gitlab.com/ee/ci/variables/>

⁶ https://docs.gitlab.com/ee/ci/jobs/ci_job_token.html

⁷ https://docs.gitlab.com/ee/ci/docker/using_docker_build.html#use-docker-in-docker

with that image, it is possible to run docker commands such as docker build and docker push. The syntax for docker push looks as follows:

```
docker push registry.gitlab.com/bighpc/<group name>/<project>/<image name>:<tag>
```

2.2. Jenkins pipeline as code

The official domain for the project has been created and is the following:

<https://jenkins-ql.bighpc.wavecom.pt/>

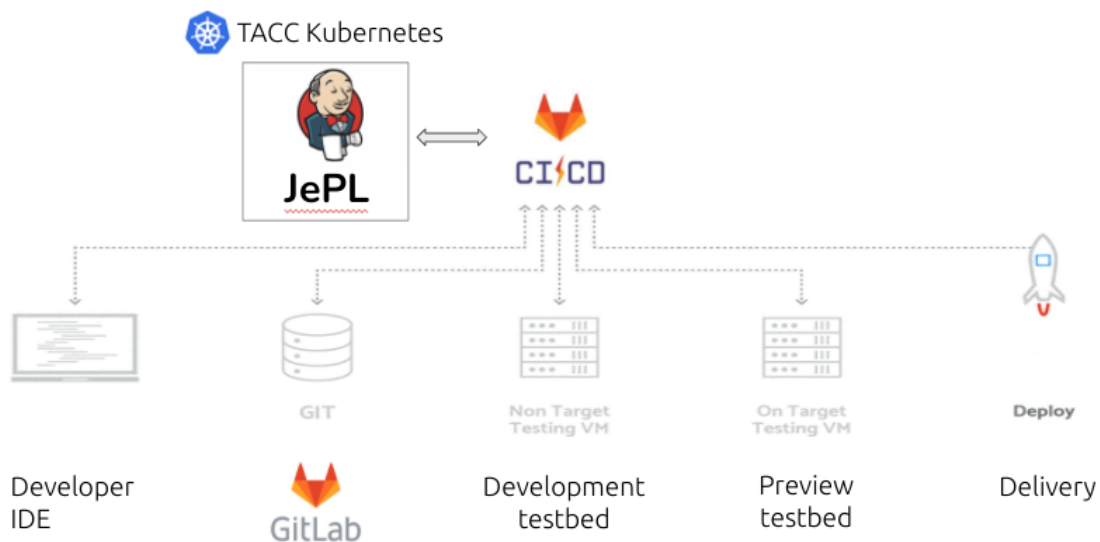


Figure 2.7 - Pilot deployment translated into a pipeline

Jenkins Pipeline Library (JePL)⁸ provides easy means of using configuration files to get a PaC with Jenkins. JePL generates dynamically the required stages and implements the criteria that allows to automate the deployment process with the following advantages:

- Software integration and testing using PaC.
- Facilitate the adoption of DevOps practices.
- Missing available FOSS alternatives or too many targeted alternatives such as Wolox CI.
- Uses (human-readable) YAML format, instead of Jenkins PaC DSL.

⁸ <https://github.com/indigo-dc/jenkins-pipeline-library/>

- Uses Composer abstraction to orchestrate required services.

Gitlab platform also supports CI pipeline and, to start reviewing the BigHPC platform deployment, the PaC example begins with the pipeline template with the draft in figure 2.7.

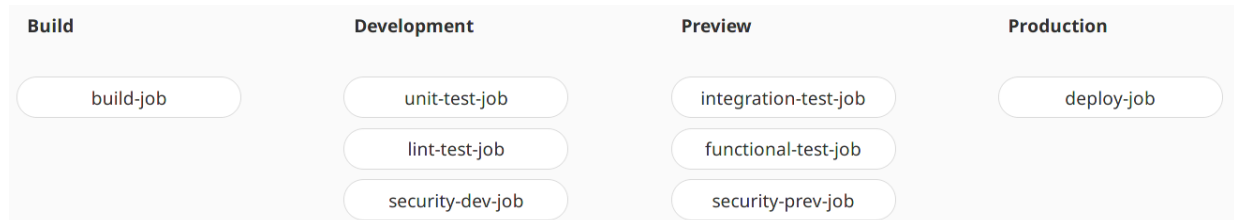


Figure 2.8 - Gitlab pipeline template preview

The pipeline in figure 2.8 has the required stages to test the code, test the BigHPC in the defined testbeds environments and deploy the pilot:

- Build: build container images, compile required code.
- Development: Unit testing, linting (code style checks), static security tests.
- Preview: Integration tests, functional tests, dynamic security tests.
- Production: Delivery to production (create a release), automated deployment.

As a first step for the developers integration task, instead of using JePL, we start with the Gitlab template since it turns out to be easier to start receiving feedback and collect the input for the environments' definition. The procedure can use the web interface provided by Gitlab and the required steps are:

- Create a new project in Gitlab.
- Add pipeline from template.

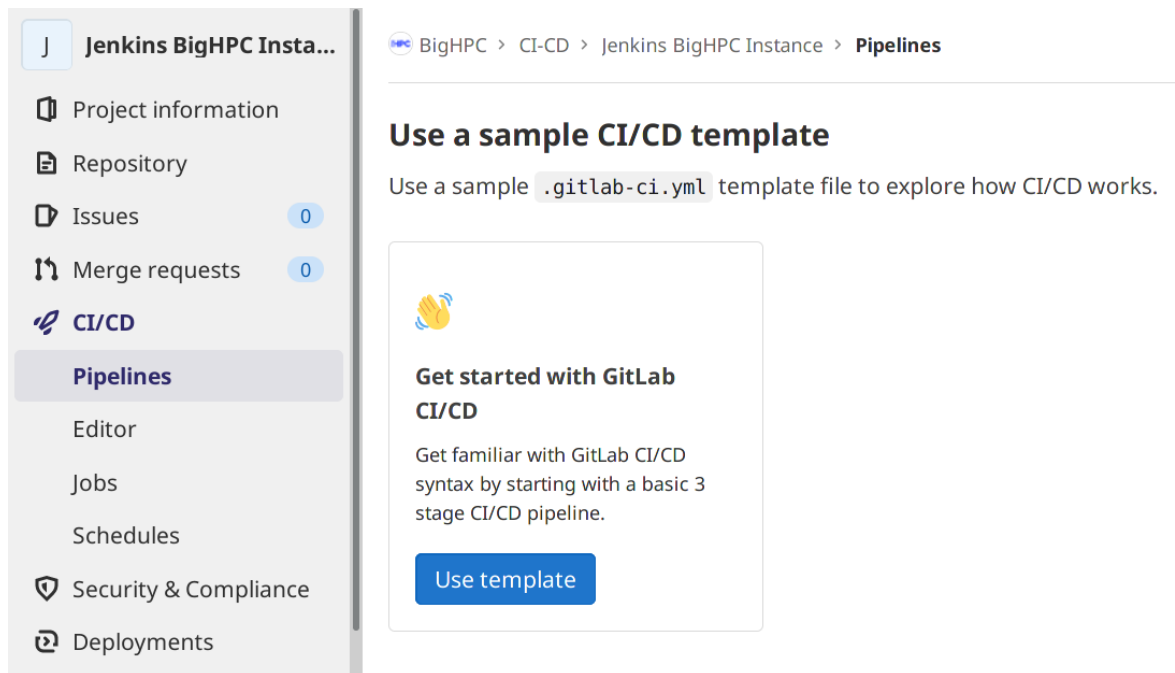


Figure 2.9 - Create a new pipeline using a template

Since the BigHPC project does not comply with requirements to have access to a gitlab enterprise license, the available features are the ones provided by the community license. As such, it is not possible to define a default template for the pipeline. Thus, the solution is to replace the code with our default configuration.⁹

Check Pipeline results:

- Check the status of pipeline after passing the tests:

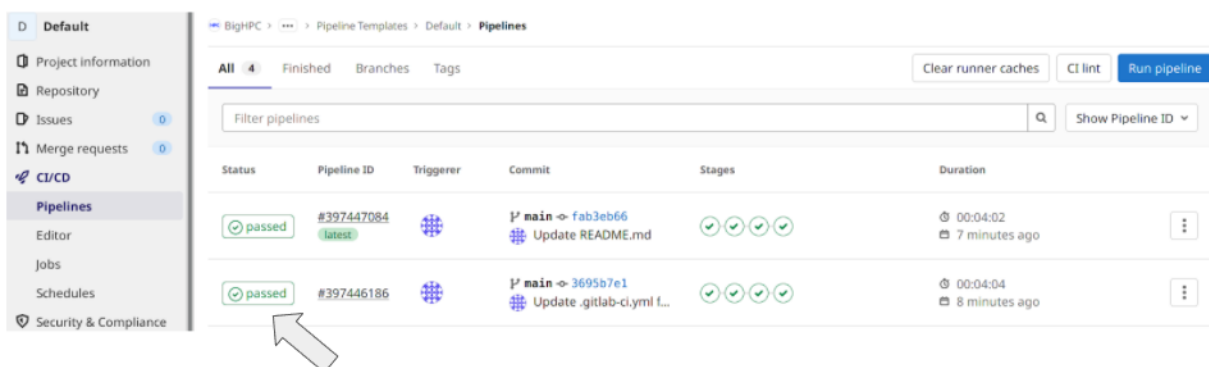
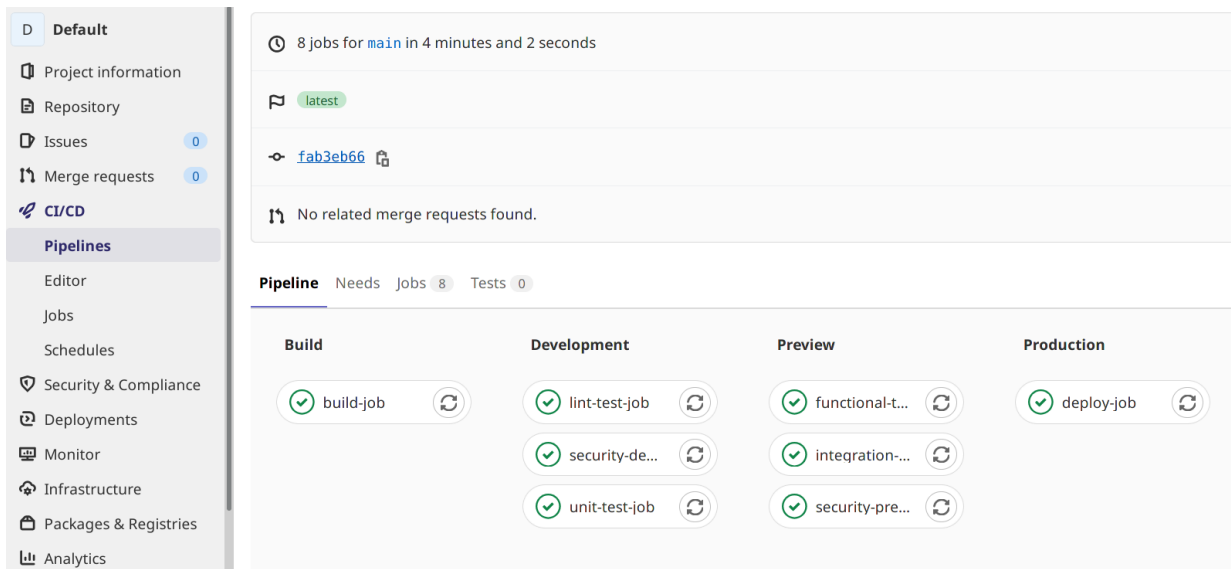


Figure 2.10 - Select the pipeline to review the results in Gitlab project

⁹ <https://gitlab.com/bighpc/ci-cd/pipeline-templates/default/-/blob/main/.gitlab-ci.yml>

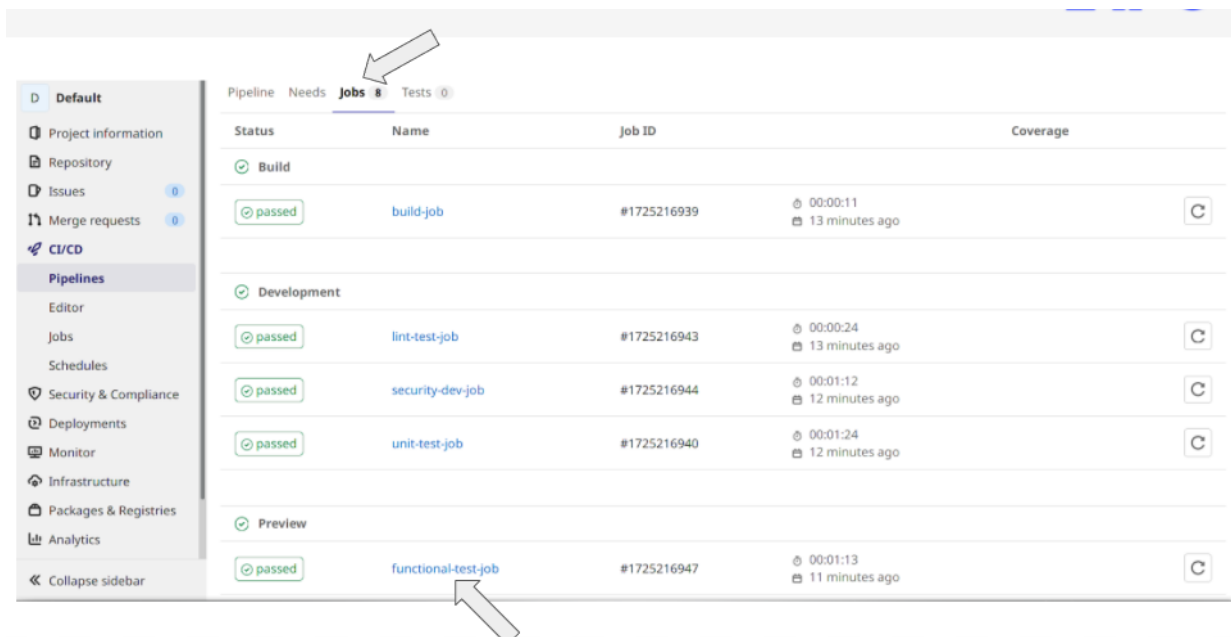
- Check the pipeline stages:



The screenshot shows the GitLab CI/CD Pipelines page for a project. The left sidebar contains navigation links: Default, Project information, Repository, Issues (0), Merge requests (0), CI/CD, Pipelines (selected), Editor, Jobs, Schedules, Security & Compliance, Deployments, Monitor, Infrastructure, Packages & Registries, and Analytics. The main content area shows the pipeline status: 8 jobs for main in 4 minutes and 2 seconds. The pipeline is labeled 'latest' and has a commit hash 'fab3eb66'. Below this, the pipeline stages are displayed: Build (build-job), Development (lint-test-job, security-de..., unit-test-job), Preview (functional-t..., integration-..., security-pre...), and Production (deploy-job). Each job is marked with a green checkmark and a refresh icon.

Figure 2.11 - Check the pipeline stages in Gitlab project

- Check the jobs state:



The screenshot shows the GitLab CI/CD Jobs page. The left sidebar is the same as in Figure 2.11. The main content area shows the 'Jobs' tab selected, displaying a table of jobs. The table has columns: Status, Name, Job ID, and Coverage. The jobs are grouped by stage: Build, Development, Preview, and Production. Each job is marked with a green checkmark and a refresh icon. The 'functional-test-job' in the Preview stage is highlighted with a blue arrow. The 'build-job' in the Build stage is also highlighted with a blue arrow.

Status	Name	Job ID	Coverage
Build			
passed	build-job	#1725216939	00:00:11 13 minutes ago
Development			
passed	lint-test-job	#1725216943	00:00:24 13 minutes ago
passed	security-dev-job	#1725216944	00:01:12 12 minutes ago
passed	unit-test-job	#1725216940	00:01:24 12 minutes ago
Preview			
passed	functional-test-job	#1725216947	00:01:13 11 minutes ago

Figure 2.12 - Check the job status after selecting a Gitlab pipeline

- Check the code execution and job execution logs:

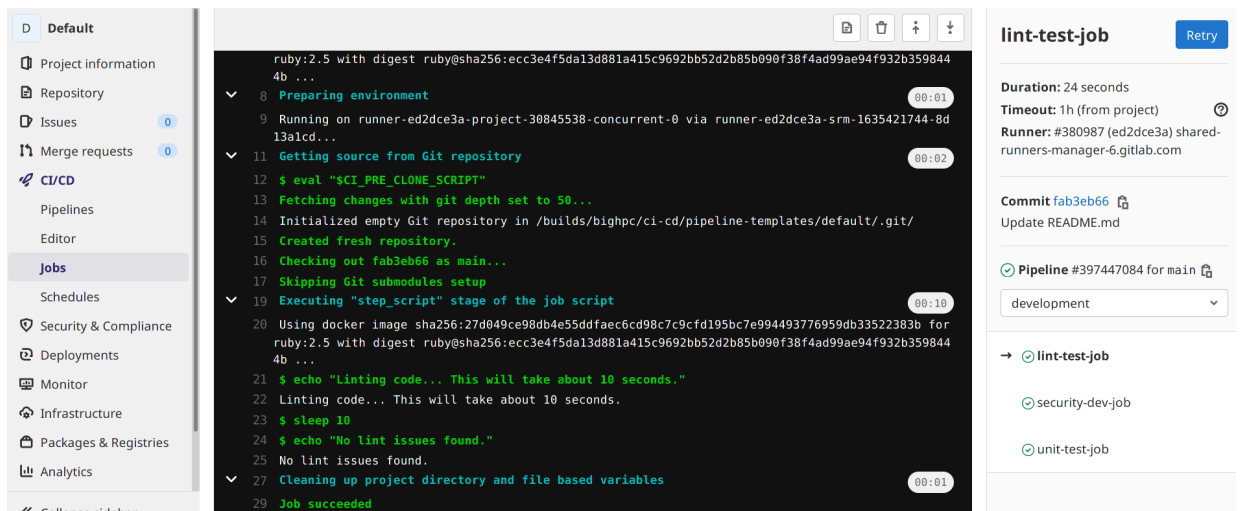


Figure 2.13 - Check the job logs after selecting a Gitlab job

JePL is going to be used afterwards with a generated configuration file from the defined template using Gitlab in the previous steps. The background execution behind the scene can be described as follows:

- Job translation to JePL to create a job in Jenkins.
- Submit the code to the defined testbeds and run the jobs.
- Connect the Gitlab project with the Jenkins instance, providing the reports in Gitlab.
- Perform the required experimental validation checking the automated results over the pilot testbeds (development and preview).
- Deploy the validated release to the production infrastructure.
- Create the docker images to package the Software.
- Run the testing jobs in a TACC dedicated kubernetes cluster.

2.3. GitLab Runners

Gitlab offers shared runners for users to run their pipelines for free. But this feature has a limitation. Currently GitLab offers 400 CI minutes per month per group for private projects. In order to provide runners for BigHPC without limits, it was decided to have runners in TACC infrastructure.

Two runners were deployed in the *BigHpc* VM:

- **highpc-runner-01**: Runner deployed on the BigHpc VM with executor **"Shell"**.

- **bighpc-runner-02**:Runner deployed on the BigHpc VM with executor “**Docker**”. This is the default runner.

2.3.1. Steps to configure a runner

Get the registration token to link the deployed Gitlab runner with gitlab.com endpoint (figure 2.14).

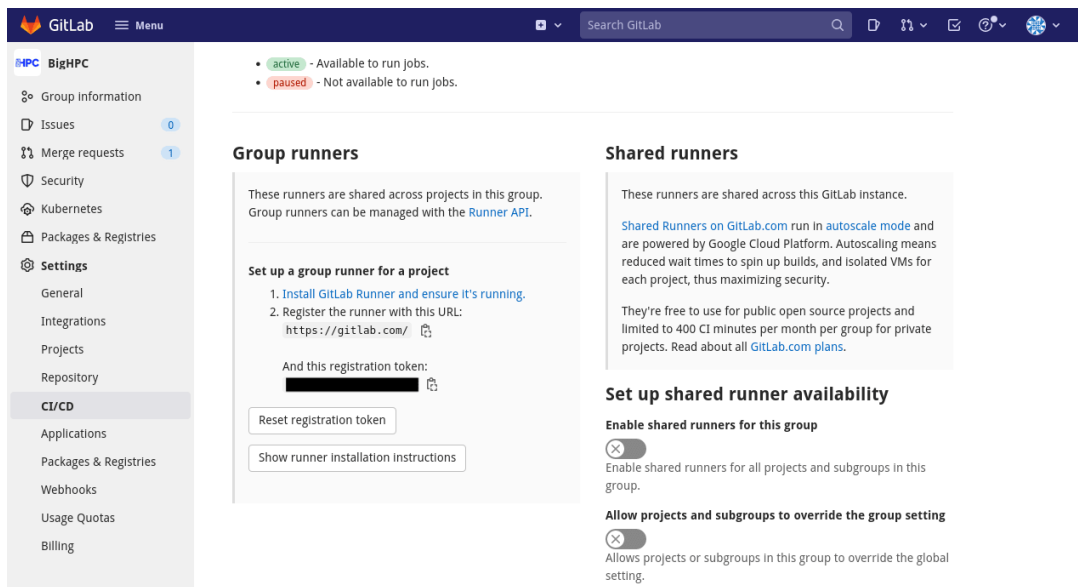


Figure 2.14 - Get the registration token to install a Gitlab Runner

In order to install a runner, developers should login in the VM (*BigHpc*) and issue the following commands:

- **curl -LJO**
"https://gitlab-runner-downloads.s3.amazonaws.com/latest/rpm/gitlab-runner_amd64.rpm"
- **rpm -i gitlab-runner_amd64.rpm**
- **gitlab-runner register**

A file will be created to store the configurations at runner registration: **/etc/gitlab-runner/config.toml**. The runners can be updated at any given time, by modifying the config.toml, without the need of restarting the service.

2.3.2. How to use runners in the pipelines

Both runners have been configured and are available to the bighpc gitlab group <https://gitlab.com/bighpc>. All the projects and subgroups below bighpc group are able to use them as well.

Furthermore, when nothing is set out, the default runner of the pipelines is the **bighpc-runner-02**. The following example shows a simple pipeline that will run, by default, on that runner:

```
build-job:
  script:
  - echo "Compiling the code..."
```

Figure 2.15 - Pipeline simple configuration example

The **bighpc-runner-02** was configured to run the code inside a container which has a docker version 20.10.10 image. This default can be changed to some other container by the user. This is accomplished by inserting the **"image"** keyword as follows:

```
build-job:
  image: docker:latest
  script:
  - echo "Compiling the code..."
```

Figure 2.16 - Set specific docker image in pipeline configuration

On the other hand, if the **bighpc-runner-01** runner is to be used to run the pipeline (running directly on the VM), we must insert the **"tags"** keyword with the tag of the runner, as follows:

```
build-job:
  script:
  - echo "Compiling the code..."
  tags:
  - bighpc-runner-01
```

Figure 2.17 - Select a specific Gitlab Runner using tag keyword in pipeline configuration

2.4. Kubernetes and Jenkins deployment

2.4.1. Kubernetes

A Kubernetes cluster was deployed using the Kubespray tool on TACC Virtual Machines to support the deployment of Jenkins.

<https://gitlab.com/bighpc/ci-cd/inventory-of-bighpc-kubernetes>

The Kubernetes cluster has the following main components:

- Services exposure to an external IP address: NodePort
- Storage: OpenEBS

After having Kubernetes up and running, the Jenkins deployment on Kubernetes was divided in two parts:

- Jenkins Operator: manages operations for Jenkins on Kubernetes.
- Jenkins Instance: BigHPC Jenkins server itself.

The image below shows the view of *octant*, a web interface for Kubernetes that allows to inspect a Kubernetes cluster and its applications. The Jenkins namespace is shown below:

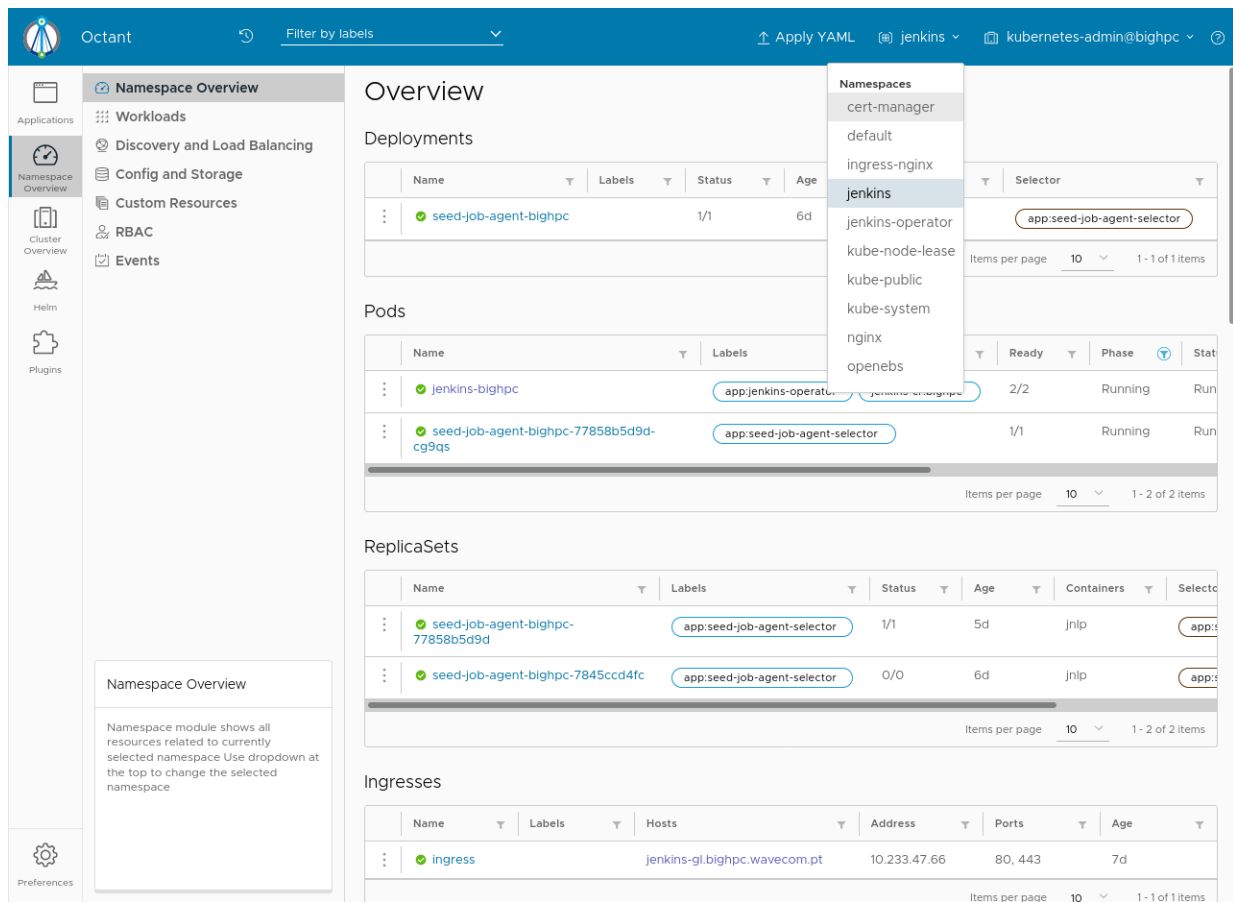


Figure 2.18 - *jenkins* namespace in deployed kubernetes cluster for BigHPC platform

2.4.2. Jenkins

The BigHPC Jenkins server is accessible through the following URL:
<https://jenkins-gl.bighpc.wavecom.pt/>

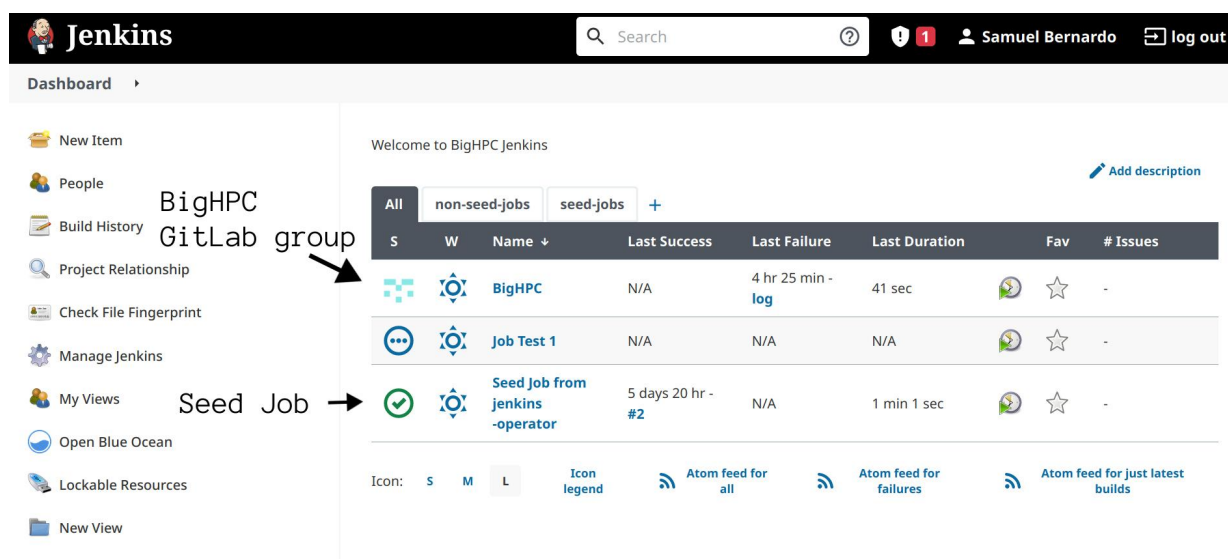


Figure 2.19 - BigHPC group and seed job in Jenkins Server

Seed Job: The seed job is a normal Jenkins job that runs the Job DSL script. The script contains instructions that create additional jobs.

Folder BigHPC: Jenkins folder that was created by the seed job. The folder contains all pipelines of all repositories belonging to the BigHPC GitLab group and subgroups.

2.4.3. Authentication

Authentication is crucial to manage the access to Jenkins. The BigHPC Jenkins instance has incorporated the GitLab OAuth plugin as an authentication and authorization mechanism, offloading authentication and authorization to GitLab.

The GitLab OAuth plugin authenticates by using a GitLab OAuth Application. Multiple authorization strategies are available to authorize users. Thus, GitLab users are translated as Jenkins users for authorization. GitLab organizations and teams are translated as Jenkins groups for authorization.

The users that have access to BigHPC Jenkins are the ones that belong to the BigHPC GitLab group.

When a user connects to Jenkins for the first time, he is asked to authorize the application “Jenkins Instance” to have access to the user’s API. This application is maintained by “Jenkins

INCD" user, a user bot that is managed by INCD. This procedure is illustrated in the image below.

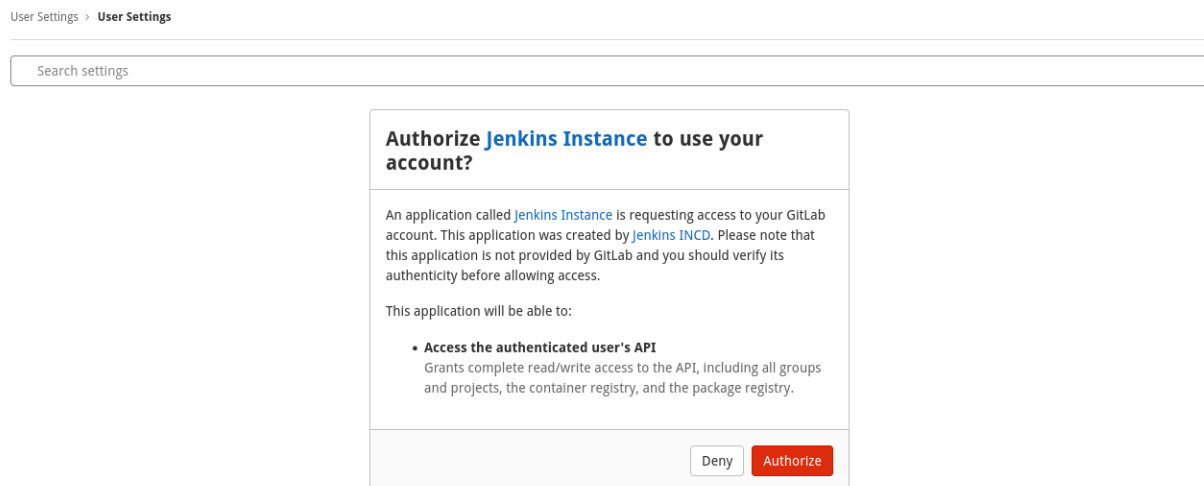


Figure 2.20 -Authorize Jenkins Instance in Gitlab.com platform

3. Pilot activities

For the BigHPC platform, it was defined in Deliverable 5.1 two kinds of testbeds: development and preview. In Section 3.1, the management of the testbeds is going to be reviewed by taking into account the behavior of real HPC clusters. Related to the components integration, the work of each activity will also be summarized: Monitoring (Section 3.2), Virtualization (Section 3.3) and Storage (Section 3.4).

3.1. Testbed definition

The creation of a testbed is the first step to accomplish a working execution environment for the end user. It creates a safe environment for verifying software compatibility as well as the platform's ease of use.

The usage of batch systems as a resource for computing data and processing information is not a trivial process for the average user, so we have to take into account the less experienced users and grant them some resources to experiment and validate the data resulting from the simulations/jobs that have been submitted.

3.1.1. Access levels

Depending on the experience of the user, the access levels to the BigHPC platform should take that into consideration, distinguishing the more experienced users from the least experienced ones.

The proposal has two access levels: A1 and A2. Access to the resources is granted to those who have more extensive knowledge of the platform, and to limit those who are less familiar with it or advanced users who simply wish to test the new software.

A1 level

Intended for users that have limited or no experience with the platform, both in the submission of jobs and the way that problems should be prepared for the usage of such a platform. Some more extensive support is required to help these users become

familiar with the platform and provide some basic examples of how a job should be submitted, as well as tutorials on how to connect to the platform. We propose the creation of a wiki with information about the available resources, both hardware and software, and the examples that steer the users in the correct way to interact with the platform.

More experienced users can also be granted this level of access as a “staging” area for the real work to be done, the time and resources granted could be used to test the new software and verify if the problem in hand can take advantage of different resources, such as cpu ram, storage space and software, that are available, as well as propose the installation of specific new software and compilers.

The proposal is that 5% to 10% of the nodes should be reserved for such an access type, depending on the number of users that will be granted A1 level access. If the majority of users are new to the platform, 5% of the cluster may be provided. If the majority of users are experienced ones, this percentage would increase to 10% of the available nodes so that they can validate their new hypothesis (specific combination of software and compilers) more rapidly and move to the A2 level quicker.

Limits to the maximum number of nodes that are used simultaneously and for how much time, should be established in order to prevent resource starvation. The proposal is that a fraction of 25% of the nodes for the maximum of 4 days, is the best solution.

Regarding the time to be granted to this access level, the proposal is 25.000 core/hour within a maximum of two months.

After the users have experienced the system and validated all combinations of available software, eventually proposing new versions to be installed and tested, they can be granted access to the second level.

A2 level

Users that are granted this access level belong to three categories:

1. They have previous experience with such a platform and have been granted access to the A2 level immediately;

2. They are experienced users that validated their new software combination using the A1 level;
3. They are new users that gained some initial experience using A1 level.

These users expect to find a platform that has no issues regarding compatibility of software version combinations. This means that all the software combinations that they might use to accomplish their final goal must have been tested previously, whether they have used the A1 access level to validate them or they have tested those combinations in similar platforms.

This access level must not be used for such validations, since it is a “pre-production grade” access level.

Users can take advantage of this access level to pre-process data for their final job, or just to expose their tests to a production level platform.

The allocated resources for this access level should be about 10% of the cluster and the time to be granted for this access level should be between 25.000 to 100.000 core/hour within a maximum of two months time frame.

With the usage of these testbed access levels, we can grant users time to both, experience the platform, and prepare for the submission of some pre-process data needed for the final job. We believe that by defining two access levels, we can separate the lengthier and larger time consuming tests from the shorter ones, allowing the users to concentrate on their tasks and the platform administrators on the support. The creation of testbeds allow the platform administrators some insights regarding the requirements of the end users, allowing us to prepare and fit the final platform for the end users.

3.2. Monitoring component

BigHPC’s Monitoring solution is composed of two interconnected components; HECTOR and MONICA, which are further detailed on Deliverable 2.2 [D2.2].

3.2.1 HECTOR

HECTOR is a simple probe that runs inside a container with the aim of continuously monitoring compute nodes and reporting their state to MONICA.

Software requirements. HECTOR only requires singularity to be installed on the system. The current prototype has been successfully tested both at TACC with singularity version 3.7.2-3.el7 and at MACC with singularity version 3.5.3.

Hardware requirements. HECTOR is focused on being lightweight and capable of running in any Linux x86 system that supports singularity therefore it has no system hardware requirements or dependencies over any hardware component.

3.2.2 MONICA

MONICA is the architecture that comprises all services related with the storage, management and visualization of metrics.

Software requirements MONICA requires docker engine and docker compose to be installed on the system. The current prototype has been successfully tested at TACC with Docker version 20.10.12, and docker-compose version 1.29.2. At MACC it was tested with Docker version 20.10.7 and docker-compose version 1.25.0.

Hardware requirements. The hardware requirements are mainly dependent on the total number of nodes being monitored by the HECTOR probes and the number of queries to the database.

In order to handle 100 parallel HECTOR probes, sending metrics from the same compute node, with an equal collection period of 10 seconds, the following values can be used for reference as the minimum hardware requirements:

- 8GB of RAM ;
- 4 Core CPU clocked at 1.9GHz;
- 20GB of disk space.

It was observed that the largest consumption of resources was due to Postgres, the memory consumption increased from 40% to 60% during the testing period. It was also observed an

increase of CPU consumption up to 220% and of disk space up to 1GB. Promscale's memory consumption was 25%. Additionally, after 1 hour the delay taken for the metrics to reach the database started to increase, this suggests that PostgreSQL was not capable of handling all the metrics being pushed from VMAgent due to resource bottlenecks. Thus, for larger scale scenarios, the recommended hardware requirements are as follows:

- 16GB of RAM ;
- 8 Core CPU;
- 50GB - 100GB of free disk space.

Build phases.

Monica is built by running the docker-compose command on the yml file.

3.2.3 CI/CD pipeline: Monitoring component

Figure 3.1 presents a diagram of the CI/CD pipeline for the monitoring component.

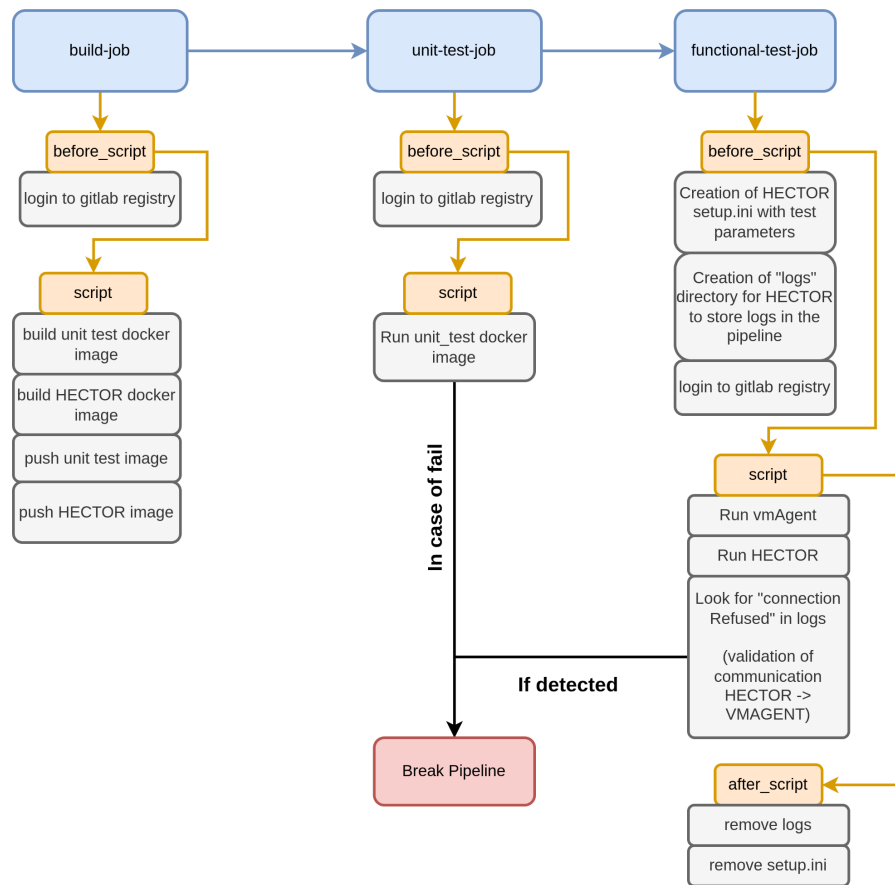


Figure 3.1 -Monitoring component pipeline diagram.

The build-job stage is where both the unit test and HECTOR images are built and pushed to the gitlab registry. Regarding the unit-test-job stage, the unit test image is run and in case of failure, the pipeline stops.

Finally, the functional-test-job validates if the HECTOR container is able to push metrics to the VMAGENT container (simulating the communication between HECTOR and MONICA), in case of failure the pipeline stops.

Future plans are to add more unit tests to the pipeline, to test HECTOR with singularity, and add other MONICA services to the pipeline.

3.3. Virtualization component

The BigHPC's Virtualization and Application Management infrastructure consists of three interacting components: Repository, Controller, and Executors. Initial prototypes for these components are described in Deliverable 3.1 [D3.1] and the current prototypes in Deliverable 3.2 [D3.2]. We describe below the requirements for the build and functional tests of these components.

Generally, because all virtualization components are written in Python and tables are stored in PostgreSQL, the requirements only consist of specific Python versions and libraries and a PostgreSQL server. These requirements are satisfied by the container-packaged virtualization test environment.

3.3.1 Virtualization Repository

The virtual Repository test primarily consists of the database unit-tests for this stage in the prototype, which are Python unit tests that use the unittest library. All current Repository APIs are in this test. This will test general connectivity to the database, table schemas and DB functions (include the Repository API).

Requirements:

- Hardware: 10 CPU cores, memory footprint is less than 1GB of memory
- Infrastructure: Postgres Database
- Software: Python > 3.8
 - Python Libraries: sqlalchemy, psycopg2

Execution example:

```
[sharrell@bighpc tests]$ ./big_db_test.py
```

```
.....
```

```
-----  
Ran 5 tests in 2.445s
```

```
OK
```

3.3.2 Virtualization Controllers and Executors

This is a regression test that instantiates one Controller service and one Executor. Then it submits a workload that the Controller sends to the Executor. After, it executes the workload, in this case a simple test script, helloworld.py. The stdout, stderr and return code are updated in the database and the test confirms that the correct data is stored. Currently, only serial jobs are tested since programmatically testing an MPI job without a known MPI stack is not possible. We plan to mitigate this shortcoming in the final prototype. Additionally, we plan to support accelerators such as GPUs in the final prototype.

Requirements:

- Hardware: 10 CPU cores, memory footprint is less than 1 gig of memory
- Software: Python > 3.8
 - Python Libraries: BigDB (BigHPC Database Prototype), websockets > 10.2
- Infrastructure: Postgres Database

Execution Example:

```
[sharrell@bighpc tests]$ ./exec_sched_test.py
```

```
Using test config for database with db name
bighpc_unittest_auto_c42741f3-3743-4bb3-a09c-916a07bb03dd
```

```
Starting the scheduler
```

```
Connecting to debug server
```

```
Executing <Task pending name='Task-1'
coro=<IsolatedAsyncioTestCase._asyncioLoopRunner() running at
/home/sharrell/bighpc-python/lib/python3.9/unittest/async_case.py:102>
wait_for=<Future pending cb=[<TaskWakeupMethWrapper object at
0x7ffaf3c139d0>()] created at
/home/sharrell/bighpc-python/lib/python3.9/asyncio/base_events.py:424>
created at
/home/sharrell/bighpc-python/lib/python3.9/unittest/async_case.py:118>
took 2.043 seconds
```

```
Trying to connect to controller: ws://localhost:8500
```

```
Successfully connected
```

```
Sending hardware info
```


Got the following message from client: {'metadata': {'op_type': 'HW_INFO'}, 'data': {'cpu': 48, 'ram': 68719476736, 'hostname': 'host1'}}

Updated conn_map = {'host1': {'websocket': <websockets.legacy.server.WebSocketServerProtocol object at 0x7fe2ecc2a7c0>, 'node_info': {'hostname': 'host1', 'cpu': 48, 'ram': 68719476736, 'timestamp': 1648141723.2185755}}}

Found node(s) for job 1

Scheduling job

Scheduling job with primary node's used cores = 10

Processing message from controller: {"metadata": {"op_type": "CREATE_JOB"}, "data": {"name": "test_job", "mode": "SERIAL", "uuid": 1, "hosts": ["host1"], "cores": [10, 10], "cmd": "python3 /home/sharrell/virtual-manager/tests/./test_data/test_basic_execution_regression_workload.py"}}

Executing command: python3 /home/sharrell/virtual-manager/tests/./test_data/test_basic_execution_regression_workload.py

Got the following message from client: {'metadata': {'op_type': 'JOB_STARTED'}, 'data': {'job_uuid': 1, 'status': 'RUNNING'}}

Got the following message from client: {'metadata': {'op_type': 'JOB_COMPLETE'}, 'data': {'return_code': 0, 'stdout': 'Hello world!\n', 'stderr': '', 'job_uuid': 1}}

Marking job 1 as completed

Received the following message from the server: {"job_complete_map": {"1": {"data": {"return_code": 0, "stdout": "Hello world!\n", "stderr": "", "job_uuid": 1}}}}

.

Ran 1 test in 5.511s

OK

3.4. Software-Defined Storage component

BigHPC's Software-Defined Storage (SDS) solution is composed of three main components, namely PAIO, PADLL and Cheferd [D4.2].

PADLL implements the SDS data plane component of BigHPC. The data plane is composed of stages that are installed at different compute nodes in order to mediate I/O operations between HPC jobs (e.g., BigData applications) and storage resources (e.g., local disks at compute nodes, Parallel File System). These stages are implemented with the aid of PAIO's framework.

Cheferd implements BigHPC's control plane component. This component is responsible for receiving QoS storage policies (e.g, I/O prioritization, I/O fairness) from system administrators and users, and ensuring that these policies are correctly applied at the HPC infrastructure. Therefore, Cheferd must collect I/O metrics from PADLL's data plane stages and, with these, fine-tune the configurations and optimizations (e.g., tune I/O rate-limiting configuration) to be employed at each data plane stage.

Each of these components is described in further detail on Deliverables D4.1 and D4.2 [D4.1, D4.2]. Next, we detail the main software and hardware requirements for building and automatically testing each of the components.

3.4.1 PAIO

PAIO is a framework that enables system designers to build custom-made SDS data plane stages. A data plane stage built with PAIO targets the workflows of a given user-level layer, enabling the classification and differentiation of requests and the enforcement of different storage mechanisms according to user-defined storage policies.

Software requirements. The prototype is written in C++17 and was built and tested with g++-9.3.0 and cmake-3.16. It depends on the spdlog (v1.8.1) and gflags (v2.2.2) libraries, which are dynamically installed at compile time by using CMake's FetchContent dependency management feature. PAIO was successfully tested with Ubuntu Server 18.04 LTS, Ubuntu Server 20.04 LTS and CentOS 7.5, with kernel 3.10 and 5.8.9, under xfs and ext4 file systems.

Hardware requirements. This component does not have strong dependencies over any hardware component, including processing (CPU or GPU), memory, storage, or network devices.

Build phases. PAIO is built under 3 main phases: *build*, *unit tests*, and *performance tests*. The build phase fetches all dependencies, compiles and installs the software component. The unit and performance tests phases execute functional and performance experiments to validate the component.

Unit tests. PAIO provides the following set of functional tests:

- *agent_test*: functional tests related to the Agent class, including creation and insertion of rules, statistic collection, and setting stage info variables;
- *channel_test*: functional tests related to the Channel object, including the creation and configuration of EnforcementObjects, creation of tickets, enforcement of requests, and statistic collection;
- *channel_statistics_test*: functional tests related to the collection of statistics in ChannelStatistics objects, including constructors creation and structure initialization, random statistic generation and storage, and collection of general, single, and detailed statistics;
- *differentiation_table_test*: functional tests related to the DifferentiationTable class, including the creation, insertion, and removal of DifferentiationRules;
- *housekeeping_table_test*: functional tests related to the HousekeepingTable class, including the creation, insertion, and removal of HousekeepingRules;
- *posix_layer_test*: functional tests related to the PosixLayer class (InstanceInterface extension, including the submission of Posix(-like) requests and setting/unsetting of stage info variables;
- *rule_parser_test*: functional tests related to the RulesParser class, including the parsing, creation, insertion, retrieval, and removal of housekeeping, differentiation, and enforcement rules;
- *stage_info_test*: functional tests related to the StageInfo class, including the constructors, setting of environment variables, and serialization;
- *status_test*: functional tests related to the PStatus class, including the constructors and error codes;

- *southbound_test*: functional tests related to the SouthboundConnection class, which handles all functions to establish the communication between the control plane and data plane stage;
- *token_bucket_test*: functional tests related to the TokenBucket class, including initialization, configuration, and enforcement;
- *token_bucket_threaded_test*: functional tests related to the TokenBucketThreaded class, including initialization, configuration, and enforcement.

Performance tests. PAIO also provides the following set of performance tests:

- *drl_bench*: benchmarks the DynamicRateLimiter enforcement object;
- *murmur_bench*: benchmarks the hashing scheme used for I/O differentiation;
- *noop_bench*: benchmarks the Noop enforcement object;
- *paio_bench*: benchmarks a PAIO data plane stage, determining the maximum performance of PAIO under different configurations.

3.4.2 PADLL

PADLL is an SDS data plane software component targeted at HPC infrastructures supporting Big Data applications. The current implementation of PADLL uses the PAIO framework to dynamically rate limit I/O requests between a given application and the PFS (e.g., Lustre). Also, it uses LD_PRELOAD to intercept I/O requests made by the applications in a transparent fashion. This decision promotes the applicability of stages to different applications without requiring any modification to their original source code.

Software requirements. PADLL is written in C++17 and was built and tested with g++-9.3.0 and cmake-3.16. It depends on spdlog (v1.8.1) and the PAIO libraries, which are dynamically installed at compile time by using CMake's FetchContent dependency management feature. Also, to apply PADLL I/O optimizations, one needs to use LD_PRELOAD (change libc calls for those implemented by PADLL) on job execution.

Hardware requirements. This component does not have strong dependencies over any hardware component, including processing (CPU or GPU), memory, storage, or network devices. PADLL can be deployed and executed under one (local) or more (distributed) servers.

Build phases. PADLL is built under 4 main phases: *build*, *unit tests*, *performance tests*, and *deployment*. The build phase fetches all dependencies, compiles and installs the software component. The unit and performance tests phases execute functional and performance experiments to validate the component. The *deployment* phase, which is still in development, deploys PADLL on different servers. In more detail, the data plane stages will be deployed along (i.e., by using LD_PRELOAD) with the targeted jobs at the corresponding compute nodes. Stages will run exclusively in user space.

Unit tests. PADLL provides the following set of functional tests:

- *namespace_test*: functional tests related to the namespace differentiation module, including the differentiation of I/O requests, selections of channels, and creation of Context objects;
- *load_balancing_test*: functional tests related to the channel selection module, including the load balancing of channels during creation of Context objects;
- *ld_dir_test*: functional tests to validate the efficiency of enforcing directory-based system calls with PADLL;
- *ld_xattr_test*: functional tests to validate the efficiency of enforcing extended attribute based system calls with PADLL;
- *ld_meta_test*: functional tests to validate the efficiency of enforcing metadata-based system calls with PADLL;
- *ld_io_test*: functional tests to validate the efficiency of enforcing I/O-based system calls with PADLL.

Performance tests. PADLL will also provide the following set of performance tests, which are currently under development:

- *padll_dp_bench*: benchmarks the PADLL data plane stage, determining the maximum performance of PADLL under different configurations;
- *padll_policies_bench*: tests different PADLL policies at different levels of granularity (per-operation, per-user, per-application);

3.4.3 Cheferd

Cheferd is a distributed SDS control plane. It receives Quality of Service (QoS) policies (e.g., I/O bandwidth, scheduling, fairness) from users and system administrators and enforces these holistically throughout the PADLL data plane.

Software requirements. Cheferd is written in C++17 and was built and tested with g++-9.3.0 and cmake-3.16. It depends on gRPC (v1.37), spdlog (v1.8.1), Asio (v1.18) and Boost (v1.77) libraries, which are dynamically installed at compile time, by using CMake's FetchContent dependency management feature.

Hardware requirements. This component does not have strong dependencies over any hardware component, including processing (CPU or GPU), memory, storage, or network devices. Cheferd must be installed on multiple servers. Namely, the local controllers are supposed to run in the same servers (e.g., compute nodes) as PADLL's data plane stages, while the global controller should be deployed on a different set of servers.

Build phases. Cheferd is built under 4 main phases: *build*, *unit tests*, *performance tests*, and *deployment*. The build phase fetches all dependencies, compiles and installs the software component. The unit and performance tests phases execute functional and performance experiments to validate the component. The *deployment* phase, which is still in development, deploys Cheferd instances in different servers. In more detail, the global controllers will be deployed on independent servers (i.e., outside of compute nodes), either as regular processes or inside containers. The local controllers will be deployed, as regular user space processes, in the compute nodes where the corresponding jobs and data plane stages are being executed.

Unit tests. Cheferd provides the following set of functional tests:

- *housekeeping_rules_file_parser_test*: functional tests to verify the correct configuration of the initial housekeeping rules file;
- *control_rules_file_parser_test*: functional tests to verify the correct configuration of the control rules files (which mimic the system's administrator configurations);
- *controller_flags_test*: functional tests to validate the controllers' (global and local) ability to correctly process input flags and arguments;

- *local_controller_connection_test*: functional tests to check if multiple local controllers are able to correctly connect to the global controller. These tests include the validation of the controllers' initialization, of the network links between the local and global controllers, and of the propagation of the initial housekeeping rules;
- *data_plane_connection_test*: functional tests to check if multiple data plane stages are able to correctly connect to a local controller;
- *collect_statistics_test*: functional tests to validate that the control plane is able to collect and aggregate statistics from multiple data plane stages;
- *control_static_rule_test*: functional tests to verify if static control rules (i.e., operation, data/metadata, job or user related) are correctly sent to different data plane stages;
- *control_dynamic_rule_test*: functional tests to verify if dynamic control rules are correctly sent to the data plane stages;
- *control_mds_rule_test*: functional tests to verify if PFS's metadata servers related rules are correctly sent to the data plane stage.

Performance tests. Cheferd will also provide the following set of performance tests, which are currently under development:

- *cheferd_gcontroller_bench*: benchmarks Cheferd's global controller , determining the maximum performance of it under different configurations;
- *cheferd_lcontroller_bench*: benchmarks Cheferd's local controller, determining the maximum performance of it under different configurations;
- *cheferd_algorithm_bench*: benchmarks different control algorithms implemented by Cheferd in terms of performance and resource usage.
- *cheferd_statistics_bench*: tests the collection of data plane statistics by Cheferd in terms of performance and resource usage.
- *cheferd_rules_bench*: tests the generation of policy rules by Cheferd in terms of performance and resource usage.
- *cheferd_distributed_bench*: benchmarks Cheferd under a distributed deployment combining different numbers of local and global controllers;

Integration tests. Finally, Cheferd will provide the following functional and performance integration tests, which are currently under development:

- *cheferd_padll_functional_test*: validates the integration between Cheferd controllers and PADLL data plane stages. This test includes the validation of the different APIs for the enforcement of rules and collection of statistics;
- *cheferd_padll_bench*: benchmarks the integration between Cheferd controllers and PADLL data plane stages. This test includes the benchmarking of the different APIs for the enforcement of rules and collection of statistics, under different workloads and distributed setups;
- *cheferd_virtualization_functional_test*: validates the integration between Cheferd global controller and the virtualization manager. This test includes the validation of the different APIs that enable communication between the two components;
- *cheferd_virtualization_bench*: benchmarks the integration between Cheferd global controller and the virtualization manager. This test includes the benchmarking of the different APIs that enable communication between the two components;

4. Conclusion

This deliverable reports the ongoing work concerned with the pilot deployment. The tasks are still in progress and here is presented the main plan for the final release.

This activity depends on the other tasks to collect the requirements for the development and preview testbeds. For Activity 4, the required resources and testbed details are going to be reviewed during project development, since they depend on the distributed storage policies to have the required privileges for the HPC workspace.

Activity 3 is collaborating in providing the job submission scripts that will allow access to the HPC clusters. These scripts need to tackle multiple implementation details, such as: the access to the resources; the requirements to bypass normal user procedure to answer the automation of continuous tasks; and translating administrators' management tasks related to most common HPC use cases into the required monitoring metrics.

Overseeing the other BigHPC platform components comes the monitoring task managed by Activity 2. To gather the required logs to debug and validate the pilot, tests must be prepared to collect all data needed by the developer or reviewer. Using the Gitlab platform it is possible to get all output from jobs along the submitted code, keeping the complete report of the development. This will allow the developer to save time avoiding the access to the infrastructure and focus on code development. At the same time, this will bring together the system administrator to help troubleshoot the issues with complete logs of all tests and, if applicable, create a docker image with all tools and dependencies required to run and test the software taking into account the particularities of each environment.

References

[D2.2] Bruno Antunes, Ricardo Leitão, Júlio Silva, “Deliverable 2.2 - Intermediate prototype of the monitoring framework”, BigHPC deliverable, 31 March 2022.

[D3.1] Richard Todd Evans, Stephen Lien Harrell, Amit Ruhela, “Deliverable 3 .1 - Initial Prototype of the orchestrator”, BigHPC deliverable, 30 September 2021.

[D3.2] Richard Todd Evans, Amit Ruhela, Stephen Harrell, “Deliverable 3 .2 - Intermediate prototype of the orchestrator”, BigHPC deliverable, 30 September 2021.

[D4.1] João Paulo, Ricardo Macedo, Alberto Faria, Vijay Chidambaram, “Deliverable 4 .1 - Initial SDS prototype”, BigHPC deliverable, 31 March 2022.

[D4.2] João Paulo, Ricardo Macedo, Alberto Faria, Mariana Miranda, Vijay Chidambaram, “Deliverable 4 .2 - Intermediate SDS prototype”, BigHPC deliverable, 31 March 2022.