M. Di Carlo*, INAF Osservatorio Astronomico d'Abruzzo, Teramo, Italy
P. Harding[1], U. Yilmaz, M. Bartolini, SKA Organisation, Macclesfield, UK
G. Le Roux, SKA South Africa, SA

Abstract

The Square Kilometre Array (SKA) project is an international effort to build two radio interferometers in South Africa and Australia to form one Observatory monitored and controlled from the global headquarters (GHQ) based in the United Kingdom at Jodrell Bank. There has been a big effort in promoting CI/CD practices among the software development teams. CI/CD is an acronym that stands for continuous integration and continuous delivery and/or continuous deployment. Continuous integration (CI) is the practice of merging all developers local (working) copies into the mainline on a frequent basis (many times per day). Continuous delivery is the approach of developing software in short cycles ensuring that it can be released anytime and continuous deployment is the approach of delivering the software into operational use frequently and automatically. The present paper analyses the decisions taken by the Systems Team (a specialized agile team devoted to developing and maintaining the tools that allow continuous practises) to promote the CI/CD practices with the TANGO controls framework.

CI/CD, SKA, TANGO, Continuous Integration, Continuous Delivery, Systems Team, TANGO controls framework, Construction, Software Development

INTRODUCTION

When creating releases for the end-users, every large software endeavour faces the problem of integrating the different parts of the software solution and bringing them to the production environment where users work. The problem arises when many parts of the project are developed independently for a period of time and when merging them into the same branch, the process takes more than what was planned. In a classic Waterfall Software Development process this is usual, but the same also happens when following the classic Git Flow, also known as feature-based branching, which is when a branch is created for a particular feature. Considering, for example, one hundred developers working in the same repository each of them creating one or two branches. When merging it can easily lead to conflicts and it becomes impossible, for a single developer, to solve all of them thus creating a delay in publishing any release (in literature this is called "merge hell"). This problem becomes evident especially working with over a hundred repositories with different underlying technologies. Therefore, it is essential to develop a standard set of tools and guidelines to systematically manage and control different phases of the software development life cycle throughout the organisation.

* matteodicarlo@gmail.com

In the Square Kilometre Array (SKA) project, The selected development process is SAFe Agile (Scaled Agile framework) that is incremental and iterative with a specialized team (known as the Systems Team) devoted to supporting the Continuous Integration, Continuous Deployment, test automation and quality.

Continuous Integration (CI)

CI refers to a set of development practices that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems as early as possible with early feedback about the state of the integration. According to Martin Fowler [**?**], there are a number of best practices to implement to reach CI:

- Maintain a single source repository (for each component of the system) and try to minimize the use of branching, in favour of a single branch of the project currently under development.
- Automate the build (possibly build all in one command).
- Together with the build, auotmated tests must also be run in order to make the software self-testing (testing is crucial because all the benefits of CI come only if the test suite is of sufficient quality).
- Every commit should build on an integration machine: the more the developers commit the better it is (common practice is at least once per day).
- Frequent commits reduce the number and size of potential conflicts: as the developer workflow is reconciled on short windows of change.
- The mainline must always be stable: If as a consequence of the integrated commit a build fails then it must be fixed immediately, as the mainline must always be in a stable state for production deployment.
- Keep the build fast so that a problem in integration can be found quickly.
- Multi-stage deployment: every software build must be tested in different environments (testing, staging and so on).
- Make it easy for anyone to get the latest executable version: all programmers should start the day by updating the project from the repository.
- Everyone can see what's happening: a testing environment with the latest software should be running.

Continuous Delivery and Continuous Deployment (CD)

Continuous delivery [**?**] refers to an extension of CI that corresponds to automating the delivery of new releases of software in a sustainable way. The release frequency can be decided according to the business requirements but the greatest benefit is reached by releasing as quickly as possible. The deployment has to be predictable and sustainable, irrespective of whether it is a large-scale distributed system, a complex production environment, an embedded system, or an app. Therefore the code must be in a deployable state. Testing is one of the most important activities and it needs to cover enough of the codebase. While it is often assumed that frequent deployment means lower levels of stability and reliability in the systems, this is not the reality and, in general, in software, the golden rule is "if it hurts, do it more often, and bring the pain forward" ([**?**], page 26).

There are many patterns around deployment and, nowadays, all of them are related somehow to the DevOps culture. According to [**?**], "DevOps is the outcome of applying the most trusted principles from the domain of physical manufacturing and leadership to the IT value stream. [...] The result is world-class quality, reliability, stability, and security at an ever lower cost and effort; and accelerated flow and reliability throughout the technology value stream, including Product Management, Development, QA, IT Operations, and Infosec". Practically it corresponds to an increased collaboration between development (intended as requirements analysis, development and testing) and operations (intended as deployment, operations and maintenance) within IT. In the era of mainframe applications, it was common to have the two areas managed by different teams with the end result of having the development team with low (or zero) interest in the operational aspects (managed by a different team) and vice versa. Having a shared responsibility means that development teams share the problems of operations by working together in automating deployment operations and maintenance, and in return operations have a deeper understanding of the applications being supported. It is also very important that teams are autonomous: they should be empowered to deploy a change to production with no fear of failures. This is only possible by supplying the necessary testing/staging platform and required infrastructure tools so that developers can engage with the platforms. It is also necessary to architect applications and deployment processes so that they can be rolled out and reverted if required. Moreover, automation is one of the key elements in implementing a DevOps strategy, as it allows the teams to focus on what is valuable (code development, test result, etc. and not the deployment itself) and it reduces human errors. The importance of those practices can be summarized in reducing risks of integration issues, of releasing new software and overall in having a better software product. Continuous deployment goes one step further as every single commit (!) to the software that passes all the stages of the build and test pipeline is deployed into the production environment (preferably automatically).

CONTAINERISATION

The *system engineering* development process has been adopted in the initial design phase of the SKA project to reduce the complexity by dividing the project into simpler and easier to resolve elements. For every element of the system, an initial architecture has been developed, which comprises the software modules needed which corresponds to a repository (each of them is a component of the system).

Since all components need to be deployed and tested together, the first decision taken is on how they need to be packaged. A container is a standard run-time unit of software that packages up code and all its dependencies so that the component runs quickly and reliably across different computing environments. A *Docker container image* is a lightweight, standalone, executable package of software that includes everything needed to run an application: code (or more generally binary), runtime, system tools, system libraries and settings.

One of the main dependency in the SKA software is the TANGO-controls [**?**] framework which is a middleware for connecting software processes together mainly based on the CORBA standard (Common Object Request Broker Architecture). The standard defines how to exposes the procedures of an object within a software process with the RPC protocol (Remote Procedure Call). The TANGO framework extends the definition of an object with the concept of a Device which represents a real or virtual device to control. This exposes commands (that are procedures), and attributes (like the state) and allows both synchronous and asynchronous communication with events generated from the attributes (for instance a change in an attribute value can generate an event). Fig. **??** shows a module view of the framework.

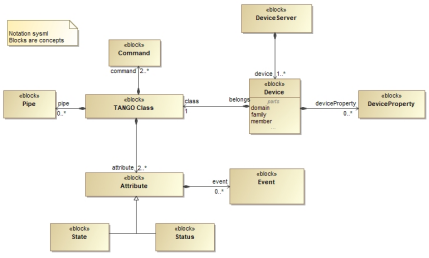

Figure 1: TANGO-Controls simplified data model.

The final product will be a containerized application which will be running in a system for managing these kinds of applications. Specifically the selection made for SKA is *Kubernetes (K8s)* for container orchestration [**?**] and *Helm Charts* [**?**] for declaring runtime dependencies for K8s applications. In K8s all deployment elements are resources, that are abstracted away from the underlying infrastructure implementation. For example these may be a Service (network configuration), a PersistentVolume (file-system type storage) or a simple Pod which is the smallest deployable unit of computing consisting of one (or more) container(s). The resources reside in a cluster (a set of machines connected together) and share a predefined network (for service dis-

covery), storage and other resources like computing power. Helm is a tool for managing Kubernetes deployments with charts where a chart is a package of pre-configured Kubernetes resources, married to run-time instance configuration. Namespaces are an important concept in Kubernetes where they are used to create a logical separation of resources within a shared ulti-tenant environment. The Namespace enforces a separate network and set of access rights enabling a virtual private space for contained deployment.

ska-tango-images

The TANGO-controls framework offers many specific tools and libraries for its use and, for this reason, a SKA repository called *ska-tango-images* [**?**] has been created which incapsulates all its components has been created in some docker images. Fig. **??** shows a diagram (of images) for this project. By extending one of them, the TANGO-controls becomes a layer inside the base images of any SKA module.

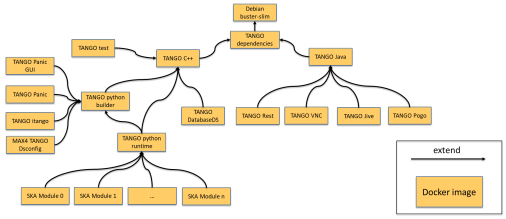

Figure 2: SKA-tango-images repository

The SKA repository *ska-tango-images* contains also the definitions two Helm Charts: *ska-tango-base* and *ska-tango-util*.

The *ska-tango-base* helm chart is an application chart which defines the basic TANGO ecosystem in kubernetes. In specific it defines the following k8s services:

- tangodb: it is a mysql database used to store configuration data used at startup of a device server.
- databaseds: it is a device server providing configuration information to all other components of the system as well as a runtime catalog of the components/devices.
- itango: it is an interactive Tango client.
- vnc: it is a debian environment with x11 server and vnc/novnc installed on it.
- tangorest: it defines the rest api for the TANGO ecosystem.
- tangotest: it is the tango test device server.

The *ska-tango-util*, instead, is a library chart which helps other application chart defines TANGO device servers. In specific, it defines the following helm named template:

- multidevice-config: it creates a k8s ConfigMap which contains the generated dsconfig json configuration file, the boostrap script for the dsconfig application and a python script for multi class device server startup;
- multidevice-job: it creates a job for the dsconfig application to apply a configuration json file set into the values file;
- multidevice-sacc-role: it creates a k8s service account, a role and role binding for waiting the configuration job to be done;
- multidevice-svc: it creates a k8s service and a k8s statefulset for a device server tag specified in the values file.

Every Helm Chart contains at a minimum the information concerning the version of the docker image and the pull policy (retrieval rule for the image) for the deployment. It also contains the necessary information to correctly initialize the TANGO database (configuration of devices) and how it is exposed to other applications for discovery in the cluster.

As any other SKA repositories, *Makefiles* are selected as an abstraction and organisation layer to eliminate language-specific scripts for building, testing and deployment and to promote ease of use in CI/CD. The use of a *Makefile* in each project simplifies the work of containerisation and, overall, the automation of the code building, testing and packaging processes. In fact, with one single command, it is possible to compile the project, generate the docker image and test it by dynamically installing the related Helm Chart for that purpose. The Makefile also enables the publishing of the docker images and Helm Charts to the SKA artefact repository and, in general, it also promotes the reusability of same build toolchain in different environments such as local development and CI/CD lifecycle.

SKA INFRASTRUCTURE

The SKA infrastructure consists of a standard footprint of VPN/SSH JumpHost gateway, Monitoring, Logging, Storage and Kubernetes services to support the GitLab runner architecture, and MVP testing facilities as shown in Fig. **??** used to support DevOps and Integration testing facilities.

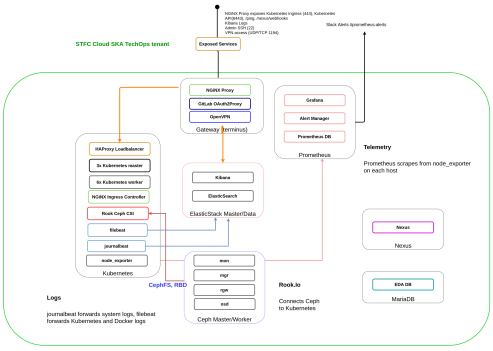

Figure 3: STFC cloud components

An initial Kubernetes cluster has been deployed with the 1x loadbalancer, 3x master, and 6x worker configuration. Fig. **??** illustrates how the load balancer ties the Kubernetes services together and exposes deployed applications such as

Ingresses to the outside world. The API Server is exposed externally from Terminus using TCP pass-through, and the NGiNX Ingress Controller is SSL terminated for external user access. These services are exposed using an NGiNX reverse proxy. The Ingress access on port 443 is password protected using oauth2-proxy integration with GitLab.

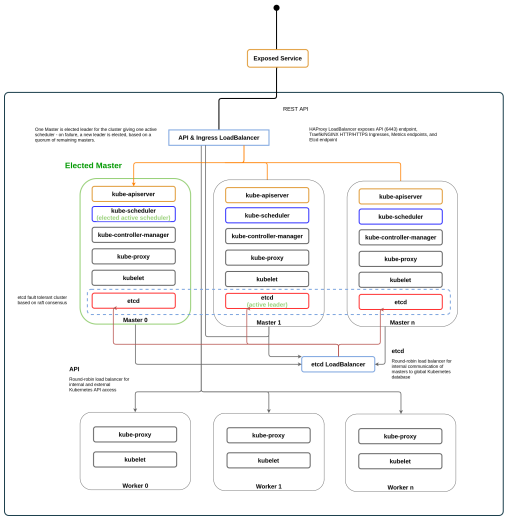

Figure 4: Kubernetes components

The Kubernetes runner [**?**] works as a multiplexer that receives requests from GitLab for jobs, and then launches Pods for each one up to a configured scaling limit (currently 15 concurrent jobs). GitLab runners use an intermediate cache to speed up jobs by passing dependencies between them. The cache for this is based on Minio which presents S3 compatible buckets for storage.

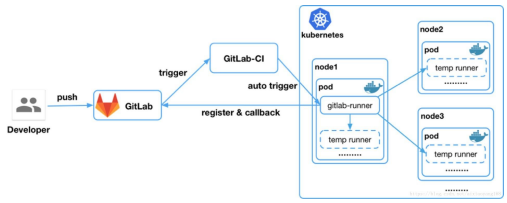

Figure 5: Gitlab runner

SUB-CHARTS ARCHITECTURE

Helm dependencies

A chart can have one or more chart dependencies, called sub-charts. According to the Helm documentation:

- a sub-chart is stand-alone (cannot depend on a parent chart),
- a sub-chart cannot access the values of its parent,
- a parent sub-chart can override values for its sub-charts and
- all charts (parent and sub-chart) can access the global values.

Let's consider two charts, A and B where A depends on B. The file Chart.yaml for the chart A will specify the dependency and in the values file it is possible for chart A to override any value of chart B. Fig. **??** shows how to do it.

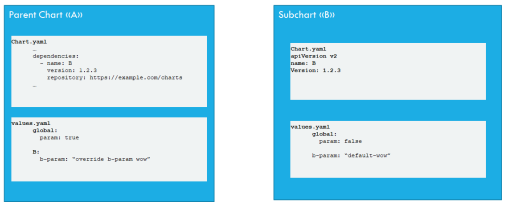

Figure 6: Chart A parent of chart B

It is also important to consider the operational aspects of using dependencies which state that when Helm installs/upgrades a chart, the Kubernetes objects from the chart and all its dependencies are

- aggregated into a single set; then
- sorted by type followed by name; and then
- created/updated in that order.

This means that if chart A defines the following K8s resources:

- namespace "A-Namespace"
- statefulset "A-StatefulSet"
- service "A-Service"

and chart B defines the following K8s resources:

- namespace "B-Namespace"
- statefulset "B-ReplicaSet"
- service "B-Service"

Then the result of the helm install command for chart A will be:

- A-Namespace
- B-Namespace
- A-Service
- B-Service
- B-ReplicaSet
- A-StatefulSet.

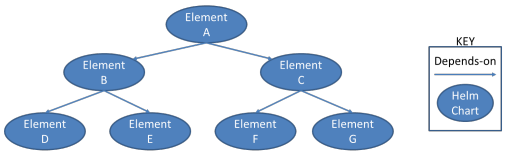

Figure 7: SKAMPI

Architecture

Since an helm chart can be in a dependency relationship with another chart, this concept can be used for integrating the various SKA elements which comprise the SKA MVP Product Integration (SKAMPI [**?**]) in a composable way that represents the bulk of the effort for integrating all the SKA software sub-systems. Fig. **??** shows a simplicistic view of the above concept and it is easy to see how it resemble a hierarchy.

Taking into consideration the operational aspect of the helm dependencies describe in section **??**, the selected helm sub-charts architecture enables a single-level hierarchy with a parent chart, called an umbrella, that pulls together the charts of the hierarchy. While SKAMPI is the composition of the entire hierarchy, it is possible to think of different umbrella charts for other purposes like integration testing between a select few elements of the hierarchy. Fig. **??** shows the umbrella chart concept: the blue umbrella chart is the entire hierarchy while the red and green ones are for other purposes. This means that every SKA element can perform its integration testing just creating an umbrella chart with the sub-elements needed for its integration.

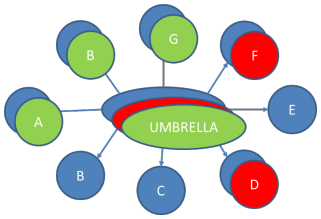

Figure 8: The umbrella chart concept

PIPELINE

In order to bring everything together for a complete CI/CD toolchain, GitLab [**?**] has been selected. The data model for a generic SKA software is shown in figure **??**.

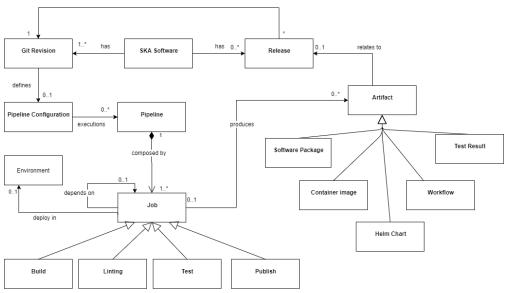

Figure 9: Pipeline definition data model.

The entry point of the diagram is the Pipeline that is composed of many jobs (i.e. shell scripts). This has been standardised for each project regardless of the artefact each project delivers so that the same standardised steps for code/configuration and helm charts are followed:

- Linting, where code is analysed against a set (or multiple sets) of coding rules in order to check if it follows the best practices decided;
- Build, where code is compiled and a docker image is created;
- Test, where the compiled package (and docker image) are tested; tests are grouped into Fast / Medium / Slow / Very Slow categories.
- Publish, where the code artefacts are published;
- Pages, where test results, documentation and logs are published (note: the name comes directly from the GitLab technology).

The pipeline is respected as the main hub of the software development in which code is built, tested, verified, published and integrated. The above steps are used in local development (as same shell scripts are available thanks to the *Makefile* targets), merge workflow, QA, integration and release. Moreover, by having an almost identical platform environment for different stages of the software lifecycle, sufficient differences between development and operations are eliminated.

Fig. **??** shows (without a specific formalism) the run-time behaviour of the selected technologies working together. At the centre of the diagram, there is a Kubernetes cluster defined for every project in the SKA telescope group (and called *syscore*). Outside the K8s cluster, there are the GitLab code repositories and Pages [**?**] (part of GitLab which represent a generic artefact used for storing pipeline artefacts such as test result), the Nexus Artifact repository [**?**] to store packaged code artefacts, the ELK stack (Elasticsearch, Logstash and Kibana) [**?**] for logging, prometheus [**?**] for metric collection and Ceph [**?**] for a distributed storage solution. Inside the cluster, in an isolated K8s Namespace, there are the GitLab runners related K8s resources which check every 30 seconds, if there are pending pipelines triggered manually or pulled by the resources. If the runner finds a pipeline, it creates a K8s Pod for each job defined in the configuration file (and part of the git revision). Each created Pod can (potentially) deploy a (umbrella) chart needed for the specific testing of the repository in an isolated Namespace (i.e. an isolated environment). The deployment installed can then be tested and the result of the job will be reported to GitLab producing artefacts that will be stored in the correct artefact repository. During any stage of the pipeline, jobs could also download required dependencies from the artefact repository. Depending on the type of job, the pipeline is also used for deploying the permanently running version of

SKAMPI or any other resources that are needed. *syscore* Kubernetes cluster is also equipped with monitoring solutions to examine the health and performance of the cluster and any resources that are deployed in it. Storage and logging solutions are also integrated to provide a consistent logging and distributed storage framework for the resources as needed. Finally, this architecture for creating temporal k8s resources for the pipeline steps (testing, building, packaging, etc.) ensures that necessary environments for the jobs are always clean (not affected by the previously run pipelines).

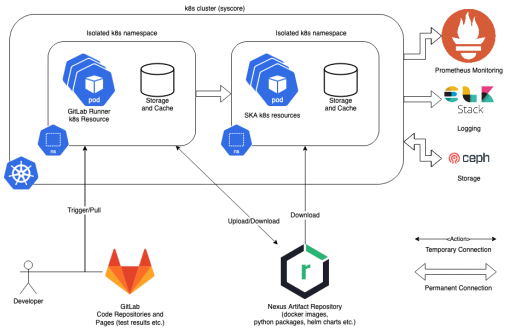

Figure 10: CICD at run time

TESTING

The most important best practice for CI is testing so the main question now is how a generic component of the SKA can be tested within the above architecture? In the SKA, testing has been split into two distinct types: pre-deployment and post-deployment tests. The deployment happens when a runner executes a job with an environment keyword. By doing so, the job is linked to the K8s cluster *syscore* through GitLab configuration. While the pre-deployment tests (namely unit tests) are all made without the real system online (using stubs and mocks), the other tests (namely integration and system tests) need more than one live system component to be up and running as the tests are mostly using other services and applications. The SKA is composed of many different modules, each of them with its own repository and different requirements for the components needed for its integration and system testing. For each of them, an umbrella chart has been introduced which enabled the specific component to be deployed together with its dependencies. Specifically, to enable the GitLab pipeline to deploy and test the chosen component each repository must:

- contain at least one helm chart
- have an environment
- have a Makefile for K8s testing

The test job, introduced in section **??**, is composed of the following steps (all made with the help of a Makefile):

- install: installs the chart (with the sub-charts needed) in the Namespace specified in the environment
- wait: wait for every container to be running
- test:
  - Create a container in the Namespace specified in the environment
  - Run pytests inside the above container
  - Return the results of the tests
- post test: delete all resources allocated for the tests

The artefacts are the output of the tests and it will have the report both in XML and JSON but also other information like the pytest output so that thw next steps (mostly packaging and releasing) in the pipeline can be run.

DEVELOPMENT WORKFLOW

There are two important assumptions behind understanding the SKA development workflow: the master branch shall always be stable and branches shall be short-lived. With the term stable, it means that the master branch always compiles and all automated tests run successfully. This also means that every time a master branch results in a condition of instability, reverting to a condition of stability shall have precedence over any other activity on the repository (and by the responsible developers). As a result, the selected development workflow for SKA is the following:

- A developer takes a copy of the current code base on which to work
- Work is started on a new branch based on the story being implemented
- As the developer advances in the implementation commits are done on the local git repo.
- Unit tests are written and run in the development environment until successfully executed
- Once the tests pass the developer pushes the changes into a remote branch
- The CI server (GitLab)
  - Checks out changes when they occur
  - Runs static code analysis and provide feedback to the developer
  - builds the system and runs unit and integration tests on the branch
  - Provide feedback to the developer about the status of the tests (fail or success)
  - Provide feedback about coverage metrics
- Once all tests execute successfully on the branch, the developer makes a pull request (*i.e. Merge Request in GitLab terms*) for merging the changes into master.
- As part of the pull request, the code is reviewed and approved by other developers.

- Code is merged into the master branch
- Then, the CI server (GitLab)
  - Runs the whole pipeline again including all the tests on the master branch
  - Releases deployable artefacts for testing (reports, code analysis, etc.)
  - Assigns a build label to the version of the code it just built (i.e. docker image version)
  - Alerts the team if the build or tests fail which fixes the issue asap
  - Publishes the successfully build artefacts to the artefact repository

CI-CD AUTOMATION AND QUALITY

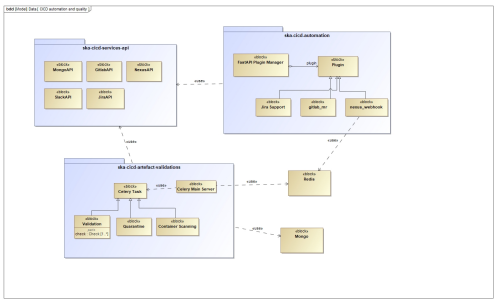

Figure 11: CI-CD automation framework

CONCLUSION

The majority of the decisions taken by the Systems Team follow the workflow as described by the Continuous Integration process outlined in Martin Fowler's paper and inspired by the state-of-the-art industry practices of [**?**, **?**, **?**]. In particular:

- For each component of the system, there is only one repository with minimal use of branching that is short-lived;
- build, tests and publish of artefacts are automated with the use of few commands;
- Every commit triggers a build in a different machine (a container within the K8s cluster);
- Once the artefacts are built (docker images, helm charts, etc.), the repository SKAMPI will create automatically a new deployment of the system and more tests are done at that level (i.e. system tests);
- Having a common repository (Nexus and GitLab page) for the code artefacts and the test results artefacts make it very easy to download the latest changes from every team and for each component to enable fast development;
- The integration environment is accessible for every developer and, is deployed in a unique Namespace in a K8s cluster.

In addition, with the new sub-charts architecture, integration testing is done within the repositories of teams and brought in to the SKAMPI integration testing repository when a new version is created. This enables developers to test not only their own work in isolation but also their work in conjunction with the developments contributed by other teams.

This work has been supported by Italian Government (MEF - Ministero dell'Economia e delle Finanze, MIUR - Ministero dell'Istruzione, dell'Università e della Ricerca).