CI-CD practices with the TANGO-controls framework in the context of the Square Kilometre Array (SKA) telescope project

Di Carlo M.[a], Yilmaz U.[b], Harding P.[b], and Dolci M.[a]

[a]INAF Osservatorio Astronomico d'Abruzzo, Teramo, Italy
[b]SKA Organisation, Macclesfield, UK

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. The project is now approaching the end of its design phase and gearing up for the beginning of formal construction. The period between the end of the design phase and the start of the construction phase, has been called bridging and, one of its main goals is to promote some 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 to merge all developers local (working) copies into the mainline very often (many times per day). Continuous delivery is the approach of developing software in short cycle ensuring that it can be released anytime and continuous deployment is the approach of delivering the software frequently and automatically. The present paper wants to analyse the decision taken by the system team (a specialized agile team devoted to developing and maintaining the tools that allows continuous practises) together with SKA architects in order to promote the CI-CD practices with TANGO controls framework.

Keywords: CI-CD, SKA, TANGO, Continuous integration, Continuous delivery, System team, TANGO controls framework, Bridging, Software development

1. INTRODUCTION

When creating releases for the end-users, every big software industry faces the problem of integrating the different parts of the software and bring them to the production environment, that is 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 classical waterfall software development process this is usual, but the same happens also following the classical git flow, also known as feature-based branching, that 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 happen to find conflicts and it become impossible, for a single developer, to solve all of them thus creating delay in publishing any release (in literature this is called "merge hell"). This problem was analyses at the Square Kilometre Array (SKA) project, 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. The selected development process is Agile (Scaled Agile framework) that is basically incremental and iterative with a specialized team (known as system team) devoted to support the continuous Integration, test automation and continuous Deployment.

Further author information: (Send correspondence to Di Carlo M.)
Di Carlo M.: E-mail: matteo.dicarlo@inaf.it
Dolci M.: E-mail: mauro.dolci@inaf.it
Harding P.: E-mail: P.Harding@skatelescope.org
Yilmaz U.: E-mail: u.yilmaz@skatelescope.org

1.1 TANGO-controls overview

One of the most important decisions taken by the SKA project is the adoption of 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 object with the concept of Device which represents a real or virtual device to control that expose commands (that are procedures), attributes (like the state) and allows both synchronous and asynchronous communication with events generated from the attributes (for instance a change in the value can generate an event). Fig. **??** shows a module view of the framework.

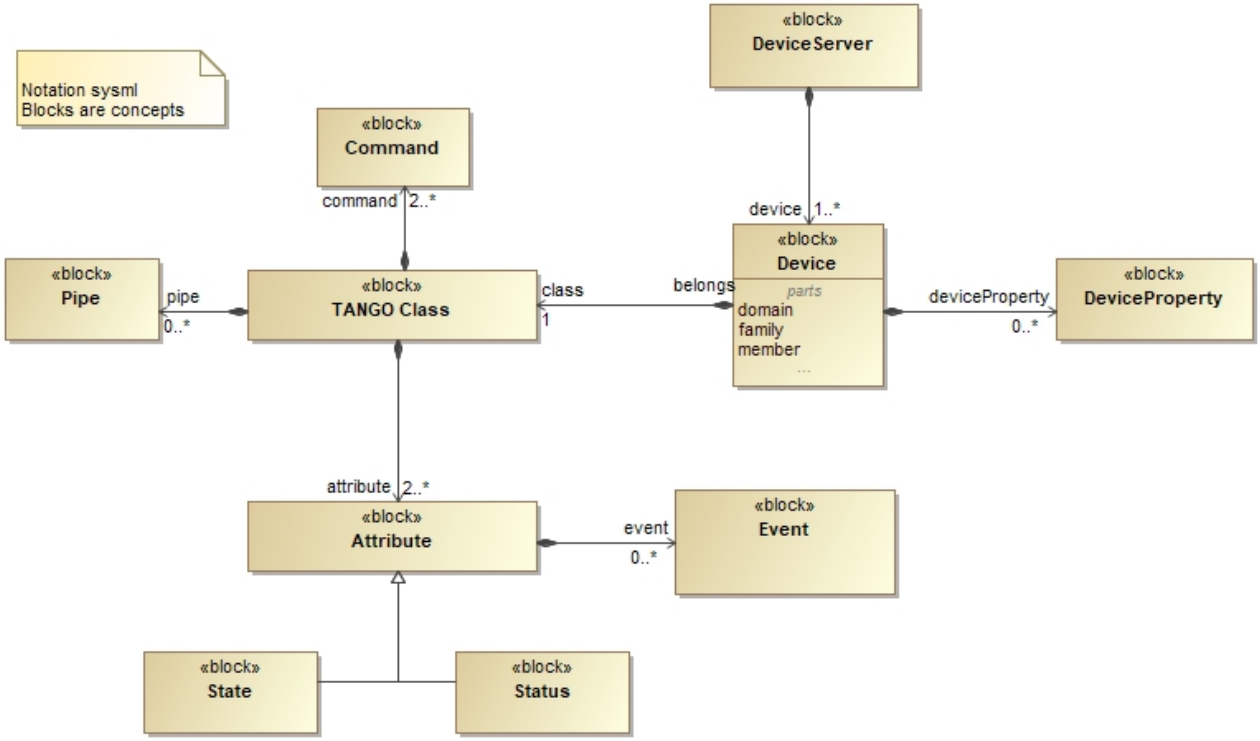


Figure 1. TANGO-Controls simplified data model.

1.2 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. 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 favor of a single branch of the project currently under development.
- Automate the build (possibly build all in one command).
- Together with the build, it must run also tests so to make the software self-testing (testing is crucial in CI because all the benefits of it come only if the test suite is good enough).
- 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). In fact, this reduces the number of potential conflicts and once a conflict is found, since the change is small, the fix is easier (as a consequence if a build fails then it must be fixed immediately).

- Keep the build fast so that a problem in integration can be found quickly.
- Multi-stage deployment: every build software 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.

1.3 Continuous delivery and Continuous deployment (CD)

Continuous delivery[?] refers to an extension of the CI that correspond to automating the delivery of new releases of software in a sustainable way. The release frequency can be decided according to the business requirement but the greatest benefit is reached by releasing as quickly as possible. The deployment has to be predictable and sustainable, no matter if 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 your 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 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 requirement analysis, development and testing) and operations (intended as deployment, operations and maintenance). In the era of mainframes 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 operations aspects (managed by a different team). Having a shared responsibility means that development teams share the problems of operations by working together in automating deployment operations and maintenance. It is also very important that teams are autonomous: they should be empowered to deploy a change to operations with no fear of failures. 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.

2. INFRASTRUCTURE FOR CI-CD

The *system engineering* development process has been adopted in the initial design phase of the SKA project in order 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 requires a repository (each of them is a component of the system).

Since all components need to get deployed and tested together, the first decision taken is on how they need to be packaged. A container is a standard unit of software that packages up code and all its dependencies so 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 in general binary), runtime, system tools, system libraries and settings.

The final product will be a containerized application which will be running in a system for managing those kind of applications. In specific the selection made for SKA was *Kubernetes (k8s)* for container orchestration[?] and *Helm*[?] for packaging k8s applications. In k8s everything is a resources which can be a service, a volume or a simple pod which is the smallest deployable units of computing. On the other hand, helm is a tool for managing Kubernetes deployments with charts where a chart is a package of pre-configured Kubernetes resources.

Every chart contains at least information concerning the version of the docker image and the pull policy for the orchestration. It also contains the needed information to initialize correctly the TANGO database (configuration of devices).

Another important decision taken for having a good infrastructure for CI-CD is the use of a *Makefile* in each project that allows to simplify the work on containerization 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 creating a container for that purpose. The Makefile also allows to push the docker image to the SKA repository. The use of makefiles also enables the reusability of same build tool chain in different environments such as local development and CI-CD lifecyles.

2.1 Subcharts architecture

2.1.1 Introduction to helm

A chart can have one or more dependencies charts, 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:

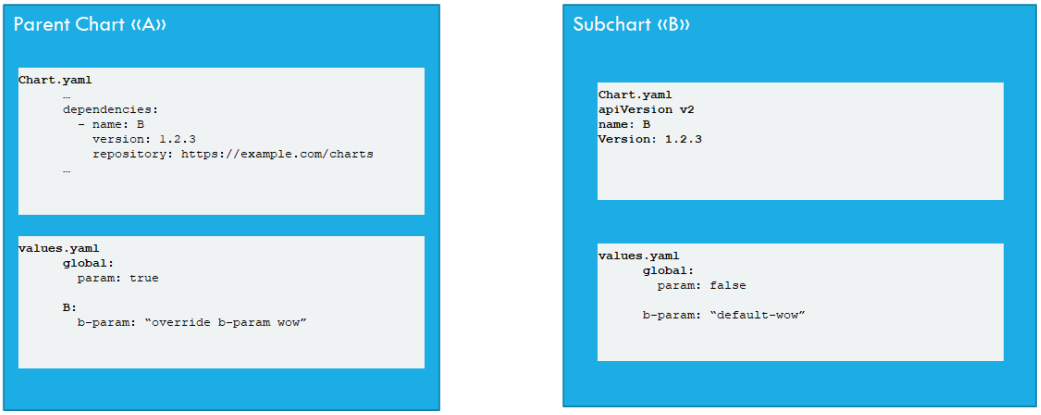


Figure 2. Chart A parent of chart B

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.

2.1.2 Architecture

The SKA MVP Product Integration, or SKAMPI,[?] is both the set of software artefacts, and the corresponding repository and continuous integration facilities that allow for the development, testing, and integration of the SKA prototype software systems. It represents the main effort to integrate the components from the different SKA elements with each other, with the goal to provide first deployable versions of SKA software. A partial dependency diagram for the helm charts available within this project is represented in fig. **??**.

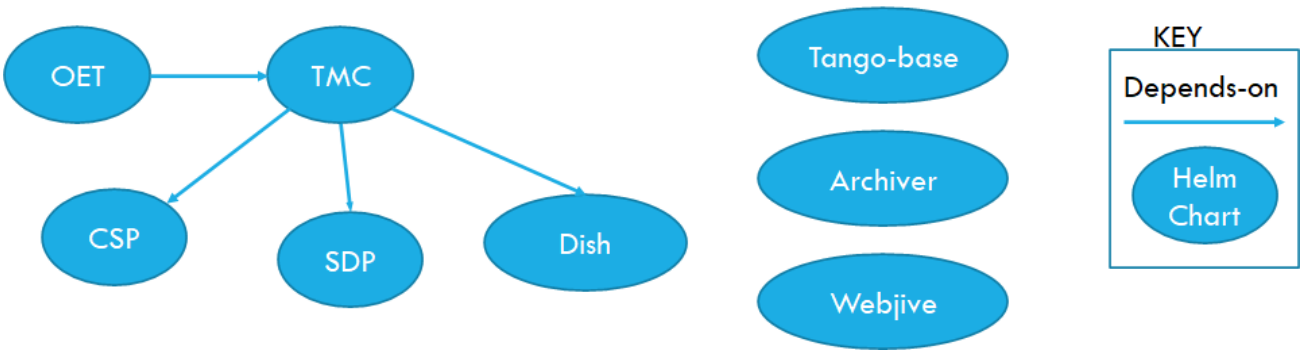


Figure 3. SKAMPI

Leaving out the details of the specific components of SKA, fig. **??** shows that all charts depend on some shared charts, like tango-base (refer to section **??**) and, optionally, archiver (refer to section **??**) and webjive. At the moment, this is modelled in the repository SKAMPI where all charts are installed with helm templating instead of the normal installation. There are a number of disadvantages in this model specifically:

- Common testing, one place for all testing with an unclear distinction the various types of tests
- Not easy to find logs
- Same k8s namespace for many deployments

- No versioning: charts are not versioned

Because of the above problems, a new architecture has been selected which enables a single level hierarchy for the shared charts (related to TANGO) and umbrella charts for charts composition (i.e. specific deployment or testing purpose). The rational can be summarized in the following items:

- every chart can be deployed with its own tango eco-system,
- every chart can have tango-base, webjive and the archiver as dependencies.

Fig. **??** shows how a generic chart (in this case TMC) which includes the shared charts. Every dependency must have a common condition on it, so that it will be possible to disable the shared charts if they are included in the parent umbrella. For instance if there is the need (for testing purposes) to have the TMC and the OET charts together the result will be fig. **??**.

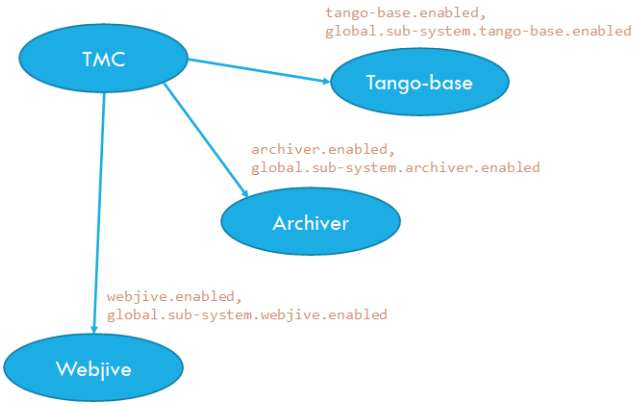


Figure 4. Chart TMC with shared charts

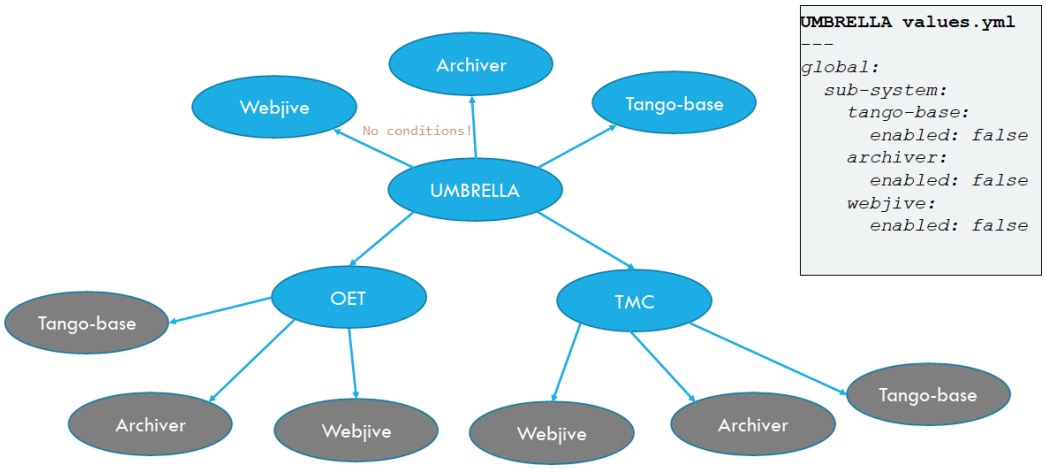


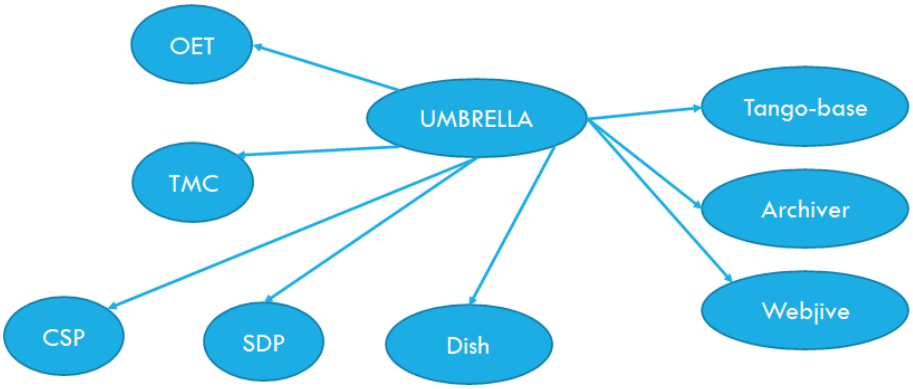
Figure 5. Umbrella chart with tmc and oet charts

Fig. **??** is the result of the refactoring of fig. **??** with the new architecture.

2.2 Pipeline

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

The entry point of the diagram is the Pipeline that is composed by many jobs (i.e. shell scripts). This has been standardised for each project in:




```
UMBRELLA values.yml
---
global:
  sub-system:
    tango-base:
      enabled: false
    archiver:
      enabled: false
    webjive:
      enabled: false
```

Figure 6. Umbrella chart for skampi: initial model refactored

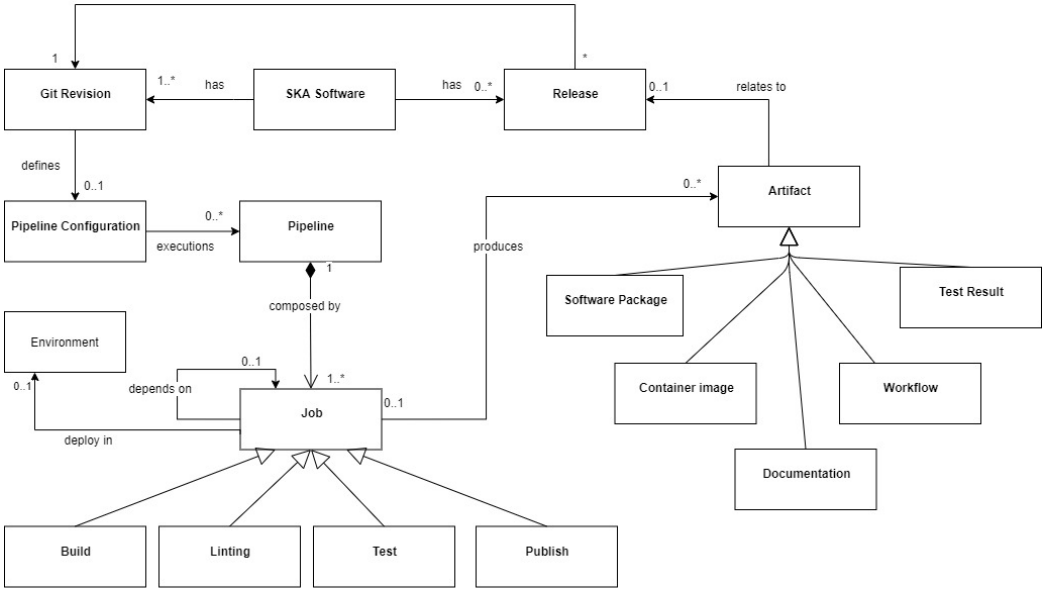


Figure 7. Pipeline definition data model.

- Build, where code is compiled and a docker image is created;
- 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;
- Test, where the compiled package (and docker image) are tested; tests are grouped into Fast / Medium / Slow / Very Slow categories.
- Publish, where the coding artifacts are published;
- Pages, where test results, documentations and logs are published (the name comes directly by the Gitlab technology).

At run time, for every change of the SKA software (i.e. a commit in the source code repository - Gitlab) one or more executors (i.e. Gitlab runners) execute all the job for the specific pipeline defined in a yaml file and part of the git revision. Each job can deploy the software in the repository into different environments like testing/staging/production and can produce artifacts (which can be a container image, documentation or test result) stored in an artifact repository. The following diagram, Fig. **??** shows the above concepts.

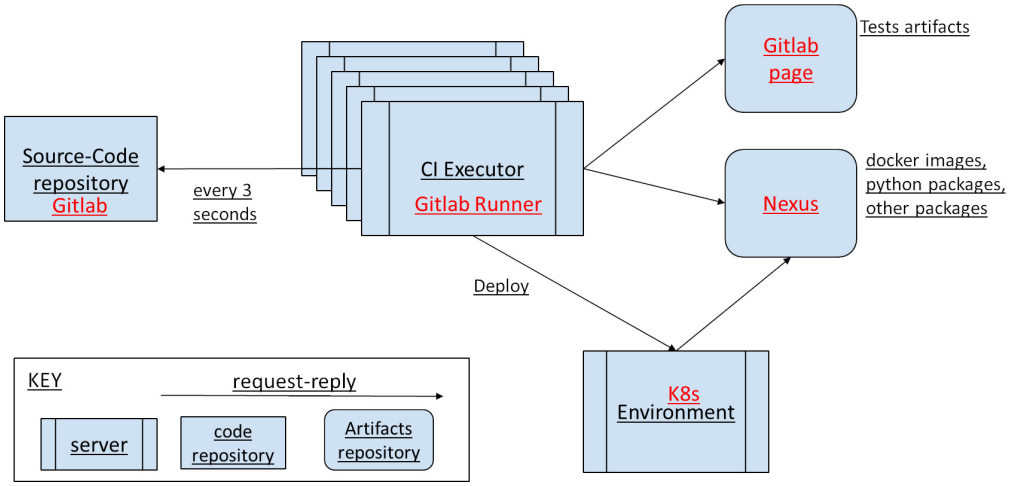


Figure 8. CICD at run time

2.3 Testing

The most important best practice for CI is testing so the main question now is how a generic component of SKA can be tested with the above architecture? In SKA, testing has been split into two distinct types: pre-deployment and post-deployment tests. The deployment happens when a runner execute a job with an environment keyword. By doing so, job is linked to a kubernetes cluster defined for every project in the ska telescope group (and called *syscore*). While the pre-deployment tests (namely unit tests) are all made without the real system to be online, the other ones (namely integration and system tests) needs more than one component to be up and running. SKA is composed by 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 requirements. In specific, to enable the GitLab pipeline to deploy and test the specific component each repository must:

- contain at least one helm chart
- have an environment
- have a Makefile for k8s testing

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

- install: installs the chart 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 tests results
- post test: delete all resources allocated for the tests

The artifacts 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.

2.4 SKA-docker

One of the most important ska repository is the one called *SKA-docker*[?] and contains the definition of a containerized TANGO environment and two helm charts for installing it in k8s: tango-base and webjive.

2.4.1 TANGO-util library chart

A library chart is a type of Helm chart that defines chart primitives or definitions which can be shared by Helm templates in other charts. In SKAMPI, many charts are a collections of TANGO device servers so it is possible to harmonize their definition with a library. Fig. **??** shows a data model diagram for the harmonized values file.

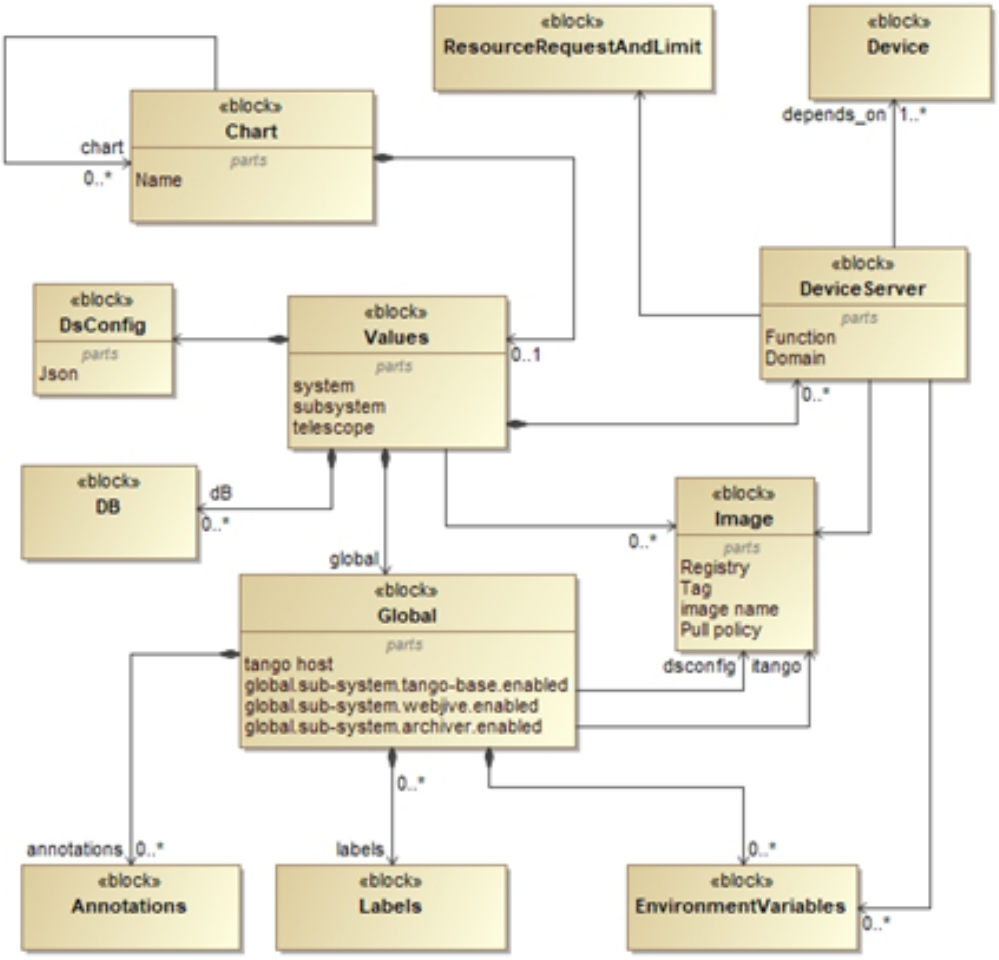


Figure 9. Data model for the values file

3. CONCLUSION

All the decisions taken by the system team try to follow some of the continuous integration suggestion from Martin Fowler's paper. In particular:

- For each component of the system, there is only one repository with minimal use of branching that are short lived;
- build, tests and publish of artifacts 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 artifacts 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 artifacts and for the test results artifacts make it very easy to download the latest changes from every team and for each component;
- The integration environment is accessible for every developer and, in specific, is a specific namespace in a k8s cluster.

Besides, with the new subcharts architecture integration testing is done within the repositories of teams and brought in the SKAMPI repo when a new version is available making developers free to tests not only their work but also their work in conjunction of the one of another team.

ACKNOWLEDGMENTS

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