CI-CD practices at SKA

M. Di Carlo, P. Harding, U. Yilmaz, D. Maia, B. Ribeiro, et al.

M. Di Carlo, P. Harding, U. Yilmaz, D. Maia, B. Ribeiro, D. Nunes, D. Regateiro, J.B. Morgado, M. Paulo, M. Santos, G. Marotta, M. Dolci, "CI-CD practices at SKA," Proc. SPIE 12189, Software and Cyberinfrastructure for Astronomy VII, 1218903 (29 August 2022); doi: 10.1117/12.2620526

SPIE.

Event: SPIE Astronomical Telescopes + Instrumentation, 2022, Montréal, Québec, Canada

CI-CD practices at SKA

Di Carlo M.[a], Harding P.[b], Yilmaz U.[b], Maia D.[d], Ribeiro B.[c], Nunes D.[c], Regateiro D.[c], Morgado J.B.[d], Paulo M.[f], Santos M.[f], Marotta G.[e], and Dolci M.[a]

[a]INAF Osservatorio Astronomico d'Abruzzo, Teramo, Italy
[b]SKA Observatory, Macclesfield, UK
[c]Atlar Innovation, Portugal
[d]CICGE, Faculdade de Ciências da Universidade do Porto, Portugal
[e]INAF Osservatorio astrofisico di Arcetri, Firenze, Italy
[f]Critical Software, Portugal

ABSTRACT

The Square Kilometre Array (SKA) is an international effort to build two radio interferometers in South Africa and Australia forming one Observatory monitored and controlled from global headquarters (GHQ) based in the United Kingdom at Jodrell Bank. SKA is highly focused on adopting CI/CD practices for its software development. CI/CD stands for Continuous Integration & Delivery and/or Deployment. This paper analyses the CI/CD practices selected by the Systems Team (a specialised agile team devoted to developing and maintaining the tools that allow continuous practices) in relation to a specific software system of the SKA telescope, i.e. the Local Monitoring and Control (LMC) of the Central Signal Processor (CSP), from now on called CSP.LMC. CSP is the SKA element with the aim to process the data coming from the receivers in order to be used for scientific analysis. To achieve this, it is composed of several instruments, called subsystems, such as the Correlator Beam Former (CBF), the Pulsar Search (PSS) and the Pulsar Timing (PST). CSP.LMC communicates to the Telescope Manager (the software front-end to control the telescope operations) all the required information to monitor the CSP's subsystems and the interface to configure them and send the commands needed to perform an observation. In other words, CSP.LMC permits the TM to monitor and control CSP as a single entity.

Keywords: CI/CD, SKA, TANGO, Continuous Integration, Continuous Delivery, Systems Team, TANGO controls framework, Software Development, CSP, CSP.LMC

1. INTRODUCTION

In software, when many parts of the project are developed independently for a period of time, it can happen that merging them into the same branch takes more than what was planned even if the classic Git Flow, also known as feature-based branching, is used (which is when a branch is created for a particular feature). In software literature this problem is called "merge hell". When there are more than one hundred developers working with over a hundred repositories with different underlying technologies conflicts might happen making it impossible to avoid delays in publishing any release. 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.

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

Further author information: Di Carlo M.: E-mail: matteo.dicarlo@inaf.it

Software and Cyberinfrastructure for Astronomy VII, edited by Jorge Ibsen, Gianluca Chiozzi, Proc. of SPIE Vol. 12189, 1218903 · © 2022 SPIE
0277-786X · doi: 10.1117/12.2620526

1.1 CI/CD

Continuous Integration (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 verified by an automated build, allowing teams to detect problems as early as possible with early feedback about the state of the integration. Martin Fowler[1] explains the best practices to implement CI which are:

- maintain a single source repository for each system's component, favouring the use of a single branch;
- automate the build (one command);
- automated testing;
- every commit should build on an integration machine;
- frequent commits;
- main branch must always be stable;
- builds must be fast;
- every software build must be tested in different environments;
- make it easy to get the latest version;
- Everyone can see what's happening: a testing environment with the latest software should be running.

Continuous delivery[2] refers to an extension of CI that corresponds to automating the delivery of new releases of software in a sustainable way. 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 to be effective. 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".[2]

There are many patterns around deployment and, nowadays, all of them are related somehow to the DevOps[3] culture. This 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 (circa 1990), 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 management so that developers can engage with the platforms developing the necessary tools, deployment processes and confidence to release on demand. 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 summarised 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).

2. CONTAINERISATION

The entire SKA project can be seen as a set of sub-systems each of them composed of a set of software modules which in turn corresponds to a git repository. All components need to be deployed and tested together so the decision taken on this aspect was to use containers. 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.

The importance of containers becomes clear with dependencies. One of the main dependencies in the SKA software is the TANGO-controls[4] framework, a middleware for connecting software processes (called device server) mainly based on the CORBA standard (Common Object Request Broker Architecture). This framework is packaged onto a set of base containers so that the final product is a containerised application that will run in a container orchestrator. Specifically, there is a SKA repository *ska-tango-images*,[5] encapsulating all its components in a set of container images. Fig. 1 shows a simplified diagram for this project. By extending one of them, TANGO-controls becomes a layer inside the base images of any SKA module solving the dependency once and for all.

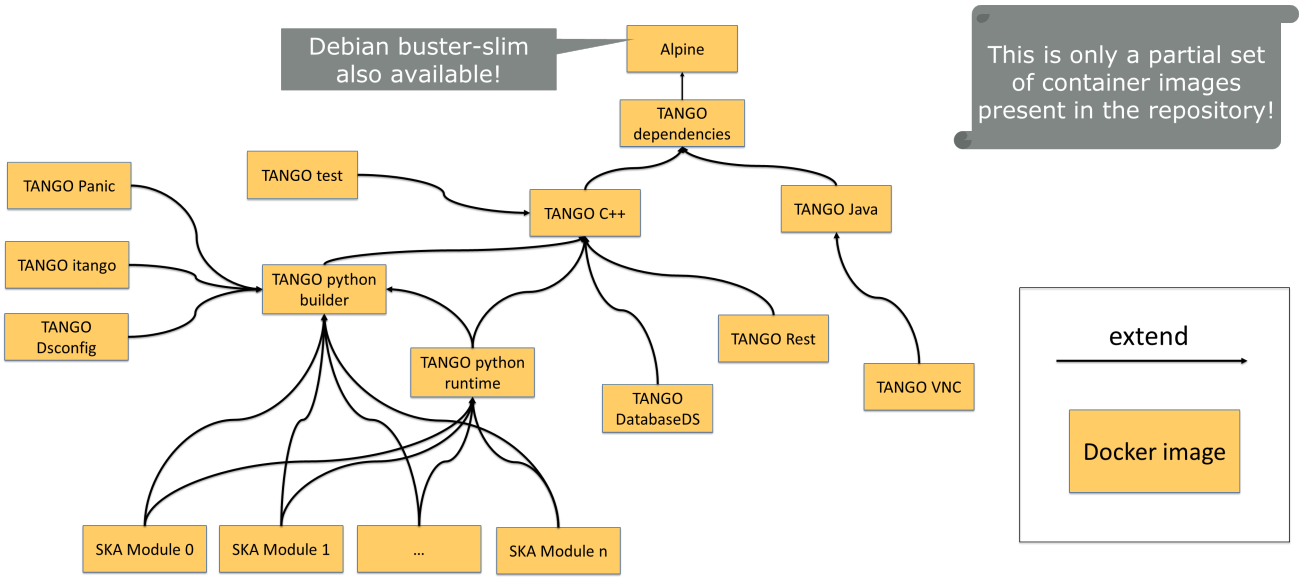


Figure 1. SKA-tango-images repository.

Kubernetes (K8s)[6] is used for container orchestration and *Helm Charts*[7] for declaring runtime dependencies for K8s applications. In K8s all deployment elements are resources abstracted away from the underlying infrastructure implementation. For example, a Service (network configuration), PersistentVolume (file-system type storage) or Pod (the smallest deployable unit of computing, consisting of containers). The resources reside in a cluster (a set of connected machines) and share a network, storage and other resources like computing power. Namespaces in k8s create a logical separation of resources within a shared multi-tenant environment. A Namespace enforces a separate network and set of access rights enabling a virtual private space for contained deployment.

Helm is a tool for managing templated K8s deployments with charts — a package of pre-configured K8s resources, tied to a run-time instance configuration. A Helm Chart contains at a minimum, information concerning the version of the container images and pull policy (image retrieval rule) for deployment. It also contains the necessary information to correctly initialize the TANGO device servers and how they are exposed to other applications for discovery whithin the cluster.

The *ska-tango-images* repository also contains the definitions of two Helm Charts that all SKA software modules use: *ska-tango-base* and *ska-tango-util*. The first one defines the basic TANGO ecosystem while the

second one is a helm library chart which helps developers define TANGO device servers through configurable template macros.

Following the path of containers, k8s and helm, each repository (a software module of a SKA sub-system) corresponds to one or more container images and, for the sub-system an helm chart must be provided. A chart can be have a dependency relationship with another chart so that it is possible to build dependencies for integrating the various SKA sub-system via a recursive include structure.

It is important to consider the operational aspects of the Helm dependencies which state that when installing or updating a chart, the K8s resources from the chart and all its nested dependencies are flattened into a single set, sorted by type followed by a name, and created/updated in that order. Due to this, a limit was imposed for single-level hierarchy with a parent chart, called the umbrella chart, which pulls together the charts of the hierarchy. In this way, it is possible to think of different umbrella charts for purposes like integration testing between a few elements. Fig. 2 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 sub-system can perform its integration testing by creating an umbrella chart with sub-systems needed for its integration.

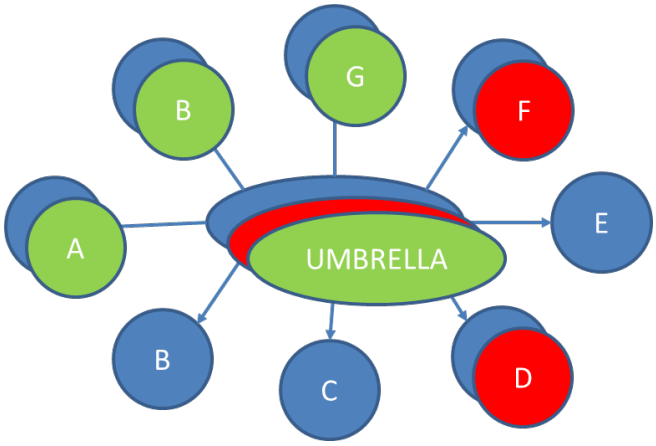


Figure 2. The umbrella chart concept.

3. PIPELINE

In order to bring everything together for a complete CI/CD toolchain, GitLab[8] has been selected. The gitlab model is shown in figure 3. The entry point of the diagram is the Pipeline which is defined in a file (called gitlab template or Pipeline configuration in the figure) which corresponds to a revision of a gitlab repository. The configuration describes a set of jobs (shell scripts) which basically compose the pipeline. Each job can produce artefacts which normally are stored in the gitlab registry storage. An artefact can be container images, software packages, helm charts or any other kind of product (including documentation).

Within the SKA, each job has a stage and the global set of stages has been standardised for all projects regardless of the artefact types each project delivers so that the same steps for code/configuration, helm charts, documentation and so on are followed. In specific, the stages are:

- Lint, where code is analysed against a set (or multiple sets) of coding rules in order to check if it follows the SKA best practices;
- Build, where code is compiled and docker images are created;
- Test, where the compiled packages are tested;
- Deploy, where developers have the ability to deploy their applications on a namespace in the k8s cluster for development (realised with the use of GitOps[9]);

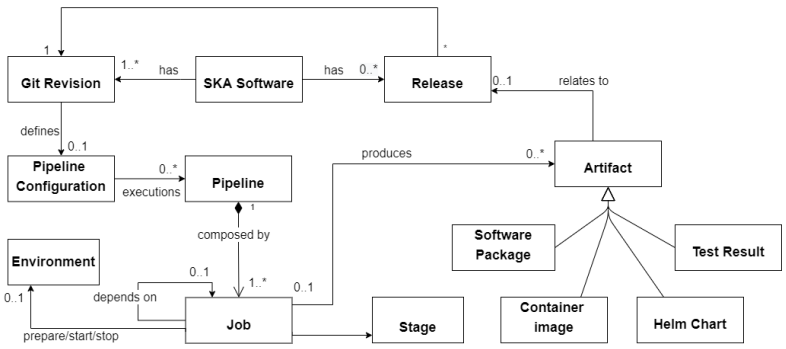


Figure 3. Pipeline definition data model.

- Page, where test results, documentation and logs are published (note: the name comes directly from the GitLab technology);
- Publish, where the repository artefacts are published;
- Scan, where container images and python modules are checked against vulnerability;
- .Post, where badges are created according to the coverage report artefact produced in the Test stage.

The pipeline machinery is a set of gitlab templates[10] and make targets[11] to track the development to delivery and retain all the released software forever. In short they defines the jobs composing the pipeline of every SKA repository described in terms of their stages above. The main motivations for doing this are listed below:

- Standards and Documentation: Ensuring the development guidelines and standards described by SKAO are followed with their documentation in place.
- Shift-left consistency: This requires the operational aspects of software support to move earlier in the Supply Chain using collaborative methods.
- Reference implementations: Ensuring the things that are committed are not broken so that we have bug-free and stable releases.
- Quality checks: Ensuring the software artefacts are complying with quality standards described by SKAO.
- Automated processing of self-describing elements at all phases of the Software Development Life Cycle (SDLC): Ensuring that each self-describing element at any stage of SDLC can be automated without difficulties.

The rationale for the use of *Makefiles* in each project (included as a git sub module) is to simplify the containerisation work and all the automation for code building, testing and packaging so that it is possible with a single command to compile the project, generate container images and test them by dynamically installing the related Helm Chart in a K8s environment. By using the same make targets, all repository will share the same folders and structure promoting a common understanding, best practices and simplifying maintainance.

The make targets also enable publishing of container images and Helm Charts to SKA artefact repository and promotes reusability of the same build toolchain in different environments such as local development and CI/CD lifecycle.

The CICD infrastructure has been realised with the help of Gitlab software and, in specific, with the Kubernetes Runner.[12] Together with the kubernetes runner, MinIO[13] has been installed as the object storage cache solution used by the gitlab job runners.

4. INFRASTRUCTURE

To support the integration's architecture a platform infrastructure was built consisting of a standard footprint of VPN/SSH JumpHost gateway (called Terminus), Monitoring, Logging, Storage and K8s services to support the GitLab[8] runner architecture, and MVP testing facilities as shown in Fig.4 used to support DevOps and Integration testing facilities. In specific, the logging solution selected is Elasticsearch,[14] the storage solution is Ceph[15] and the (central) artefact repository (CAR) is Nexus Repository Manager.[16] In relation to CAR, it is important to notice that only artefacts produced from the gitlab pipeline are allowed to be stored into the repositories and only if the pipeline has been triggered for a git tag. In all other cases, the gitlab artefact repository is used. The selected monitoring solution is Prometheus.[17]

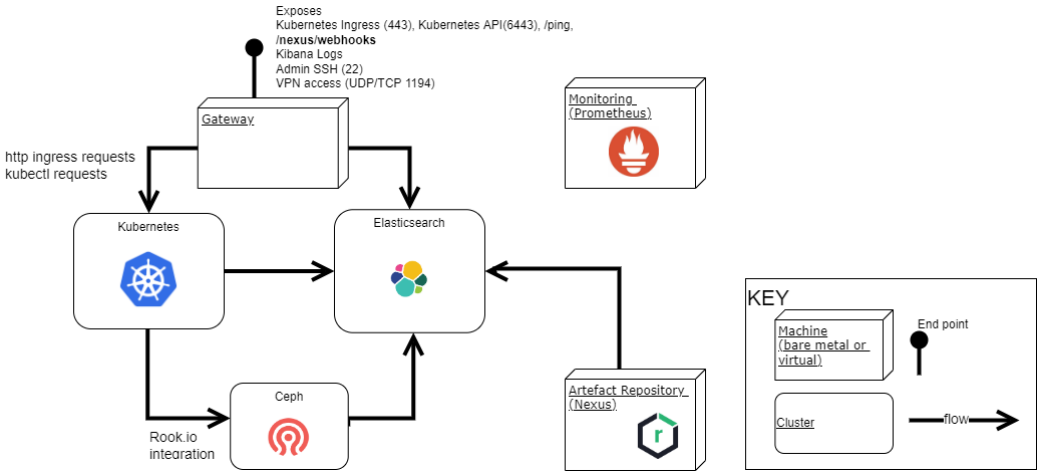


Figure 4. Simplified infrastructure

Fig.5 illustrates how the LoadBalancer ties K8s services together exposing deployed applications to the outside world. The K8s API Server is exposed externally from the Gateway using TCP pass-through, and NGiNX[18]'s Ingress Controller is SSL terminated for external access. These services are exposed using an NGiNX reverse proxy. Ingress access on port 443 is password protected using oauth2-proxy[19] integration with Azure AD.

For the creation of the infrastructure a set of repositories[20] has been created in order to support the deployment and maintenance of SKA Services. Every repository of this group is intended to do one thing very well: deploy the elasticsearch cluster, the prometheus platform and so on so forth. For each infrastructure, a repository is created which includes the services required as git sub-modules. An example of this repository can be found in the *SKA CI CD Deployment on STFC Cloud* repository.[21] At the moment 3 complete infrastructures have been deployed in three different location: Australia, Portugal and UK.

In fig. 6 it is shown how the gitlab pipelines are executed with the help of the K8s runner.[12] It works as a multiplexer receiving requests from GitLab for jobs and launching their respective Pods up to a configured scaling limit. GitLab's runners use an intermediate cache to speed up jobs by passing dependencies between them. This cache is based on Minio[13] with S3[22] compatible buckets for storage.

5. TESTING

One of the most important best practice for CI is testing and the relentless improvement of the quality of testware is a primary goal. There are different kinds of tests, mostly depending on the system-under-test (SUT):

- Acceptance Tests. Cover formal testing of the deployed system. Can involve tests done manually by stakeholders.
- System Tests. Check that the integrated system can perform certain functions.

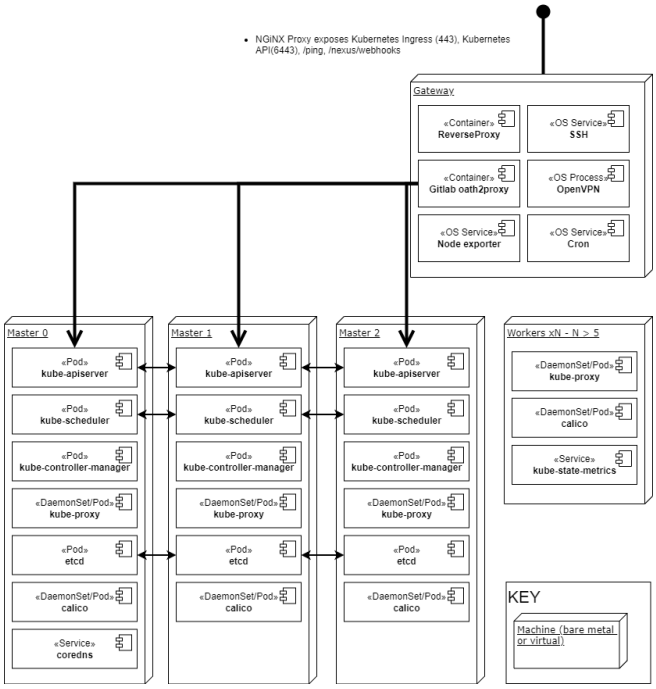


Figure 5. Kubernetes infrastructure

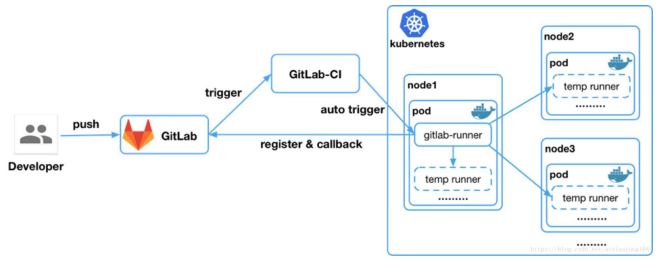


Figure 6. GitLab runner

- Component and Integration Tests. Test specific parts of the system in isolation, typically by focusing on one interface and mocking as much of the remaining system as posible.

Tests are fully integrated into the pipeline machinery developed. In order for them to work in the pipeline in the expected fashion there should be a tests folder in the root of the Gitlab repository. Inside the test folder it is important to follow pytest[23] conventions. The pytest framework supports execution of most of SKA test cases and can scale to support a very large number of tests.

When the tests are related to the system (as opposed to individual components or services) it is important that they act as living documentation, that is, test cases can be used as documentation of the business logic and policies implemented by the system. Those tests should be defined using simple natural language formulations that allow for a good mapping between software and requirements. In turn, this will allow a clear representation of the capabilities of the system to stakeholders. In practice what this means is that those tests can be understood by people who aren't developers. Behaviour Driven Development (BDD) tests are defined using "given", "when", and "then" steps using the Gherkin specification language.[24]

Every Ansible playbook included in the infrastructure repositories shall be tested with ansible molecule.[25] Molecule is a project that helps with development and testing of Ansible roles.

The make targets use many scripting tools such as bash, sed, awk, jq, yq, and UNIX commands. These are tested as well with the use of the BATS framework.[26]

In the SKA, testing has been split into two distinct types: pre-deployment and post-deployment tests. While the pre-deployment tests (namely unit tests) are all made without the real system online (using stubs and mocks), the post-deployment tests (namely integration and system tests) need more than one live system component to be up and running to facilitate interactions with 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.

The test stage, introduced in section 3, is composed by those types of tests and the post-deployment is done in the following way:

- Install (deploy) the application (umbrella chart) in a new namespace (using the gitlab registry)
- Wait for all containers to be running
- Run a new container which will run integration tests on the deployed application
- Uninstall the application and delete all resources created

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 the next steps (mostly packaging and releasing) in the pipeline can be run.

6. MARVIN

To ensure that all best practices are followed by SKA developers an AI called Marvin, has been implemented in order to perform quality checks on Gitlab merge requests and validation on artefacts published to CAR. In fact, using webhooks from Gitlab and Nexus, Marvin performs its checks and also informs the developers of what is happening such as new releases, or quality issues.

From a technical point of view it is composed by three main software packages: a FastAPI[27] application, a celery[28] application and a library which comprises different APIs to access external services. The two applications work together and both are realised with a plugin architecture (where a plugin is a checks or a validation). In specific:

- **ska-cicd-services-api**[29] which includes all APIs that will be used in the other two packages. These APIs allow to communicate with GitLab (i.e merge request creation or for project information), Slack[30] (i.e. sending messages to channels), Jira[31] (i.e. project information) and Nexus (i.e. obtaining component information);
- **ska-cicd-automation**[32] which uses FastAPI[27] to build a Python web application with a plugin architecture. Three plugins have been created: **gitlab_mr** used for merge request quality checks and providing feedback to developers directly on GitLab; **jira_support** to handle jira operations and **nexus_webhook** to trigger webhooks every time a new artefact is published;
- **ska-cicd-artefact-validations**[33] based on Celery[28] containing a main server pulling messages from Redis,[34] transforming them into artifact validation tasks and storing the results into a MongoDB[35] database.

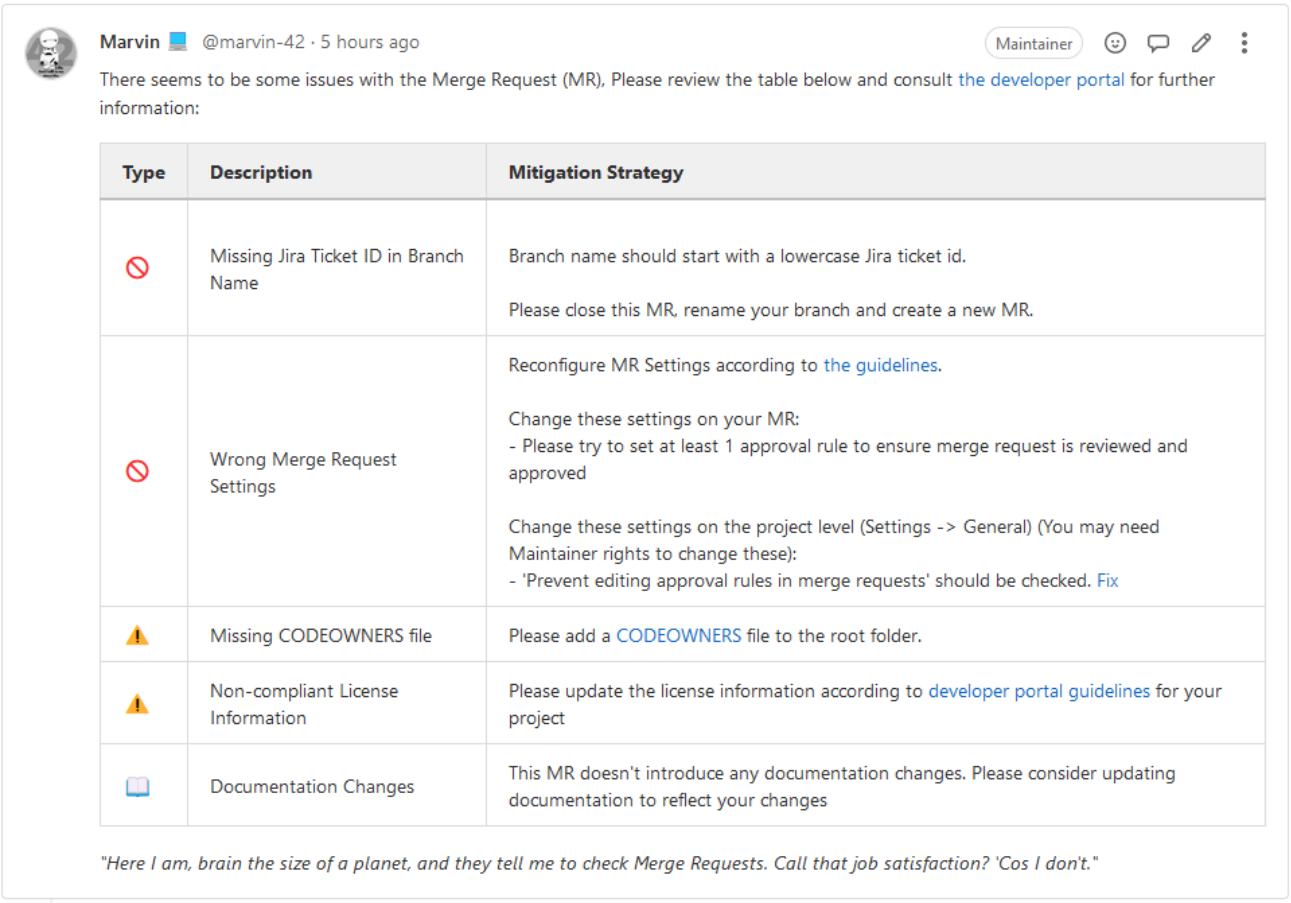
6.1 Merge Request Quality Checks

To ensure that every developer follows the development workflow and best practices, automated checks are performed on Merge Requests. A webhook was added to the ska-telescope GitLab group, which triggers a service (*ska-cicd-automation*[32]) every time a new Merge Request is created. Among other quality checks, it will then verify if:

- the Merge Request Settings were set correctly;

- the branch name, the commit messages and the merge request have a Jira Ticket ID;
- the project has a proper license;
- the project as documentation on it and if it was updated;
- the project has pipelines with the needed jobs;
- the project has any references to docker compose;
- the project slug is complaint with SKA standards;
- the correct CI/CD Makefile templates are included.

After performing the checks, the results are reported back to the developers on GitLab's main Merge Request page via a comment (see fig. 7 for an example of the comment, including a table with the severity of the failed check, description about the check and mitigation strategy).



Marvin @marvin-42 · 5 hours ago — Maintainer

There seems to be some issues with the Merge Request (MR), Please review the table below and consult the developer portal for further information:

Type	Description	Mitigation Strategy
🚫	Missing Jira Ticket ID in Branch Name	Branch name should start with a lowercase Jira ticket id. Please close this MR, rename your branch and create a new MR.
🚫	Wrong Merge Request Settings	Reconfigure MR Settings according to the guidelines. Change these settings on your MR: - Please try to set at least 1 approval rule to ensure merge request is reviewed and approved Change these settings on the project level (Settings -> General) (You may need Maintainer rights to change these): - 'Prevent editing approval rules in merge requests' should be checked. Fix
⚠️	Missing CODEOWNERS file	Please add a CODEOWNERS file to the root folder.
⚠️	Non-compliant License Information	Please update the license information according to developer portal guidelines for your project
📖	Documentation Changes	This MR doesn't introduce any documentation changes. Please consider updating documentation to reflect your changes

"Here I am, brain the size of a planet, and they tell me to check Merge Requests. Call that job satisfaction? 'Cos I don't."

Figure 7. Checks results table

6.2 Nexus Artefact Validation

There are many packaged code artefacts of multiple formats being created in the SKA project hosted on GitLab and then published to the Nexus Artifact repository. These should follow SKAO's conventions: artefact names should be compliant with ADR-25,[36] versions compliant with semantic versioning 2.0.0,[37] must include associated metadata with the required information, such as who published it, from which GitLab repository it originated from and other useful information, and python artefacts and oci images must not have any critical vulnerabilities.

To ensure that the guidelines and policies described are followed for consistent, compliant and robust artefact management, there are a series of automated validations in place at *ska-cicd-artefact-validations*.[33] If an artefact fails validation, it is moved to a quarantine state and the results of the checks are reported back to developers that triggered the pipeline that published it. This report is made by creating a new Merge Request where the developer is made assignee and its description contains a table composed of the failed validations and instructions on how to mitigate them.

The execution of artefact validations happens following the Celery architecture with a server that pulls messages from a queue (Redis) and creates tasks (processes) to perform the specific validation. Every task can then create other tasks as needed to perform other activities (i.e. quarantine the artefact or create merge request on GitLab). The result validation is stored in a MongoDB database.

When an artefact is published via a GitLab job the following tasks are performed:

- **validation**: performs the validation checks;
- **get metadata**: extract existing metadata from artefacts;
- **container scanning**: scans for container vulnerabilities using Trivy;
- **python scanning**: scans for python modules vulnerabilities using Gemnasium;
- **quarantine**: quarantine artefacts if any checks fail;
- **create MR**: create MR to report failures to developers;
- **insert DB**: insert metadata into MongoDB about the validation performed.

With the above tasks, it is possible to keep and maintain a clean and organized repository, where all artefacts follow the guidelines and policies defined on the project.

7. APPLICATION OF SKA-CICD: CSP.LMC

In this section we analyze the application of CI/CD practices to a specific software system of the SKA telescope, the CSP.LMC, i.e. the Local Monitoring and Control (LMC) software for the Central Signal Processor (CSP).

CSP is the SKA element that processes the data coming from receivers to let them be used for scientific analyses. It is composed of three main instruments, from now on called "subsystems". They are:

- the Correlator and Beam Former (CBF), that creates the visibility from raw data coming from the antennas;
- the Pulsar Search (PSS), to retrieve suitable candidates for the research of pulsars;
- the Pulsar Timing (PST), that measures the frequency of the radiation emitted from pulsar candidates.

CSP.LMC represents the interface of the entire CSP instrument, exposing it as a single entity to the clients that are working in the SKA TANGO environment. In the integrated software for SKA, CSP.LMC communicates to the Telescope Manager all the required information to monitor the CSP's subsystems. It also provides the interface to configure the subsystems and to send all the commands required to perform an observation. In the SAFe development process of SKA software, CSP.LMC code is developed by a specific Agile team, the CREAM team.

CSP.LMC have two different implementations that are specific to the Mid range and Low range frequency SKA telescopes. For each implementation a Gitlab project is provided: *ska-csp-lmc-mid* and *ska-csp-lmc-low*. However, since most functionalities are common to both telescopes, the most of the code is contained in a third repository, the *ska-csp-lmc-common*, that implements the base classes for Mid and Low specialization. Moreover, ska-csp-lmc-common is developed on the base of the ska-tango-base code, the "basic TANGO ecosystem", as mentioned in section 2. Figure 8 show a simplified UML diagram for the inheritance structure of CSP.LMC.

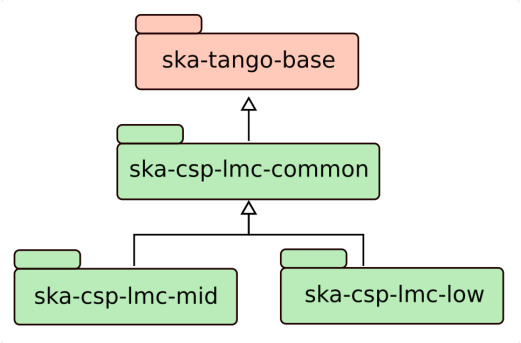


Figure 8. Simplified UML diagram for CSP.LMC inheritance. Green boxes are related to the Code developed by CREAM team

The use of SKA common templates has been a well consolidated practice in the development of the CSP.LMC code. This permits the code developers to focus on the business logic of the instruments. In this case, the use of *ska-tango-base*[38] provides a tested and reliable foundation for the interface between the LMCs features, written in Python, and the TANGO controls Framework. On the other hand, the *ska-tango-util*[5] project takes care of the definition for the TANGO device server in the k8s environment.

In this context, the whole CI/CD machinery developed by the system team ensures to the three CSP.LMC Gitlab projects a predictable context for delivering the artifacts needed by the SKA software community. Furthermore, it gives the CREAM developers an environment that is sufficiently customizable.

For the three projects, all the jobs defined in section 3 are provided into the Gitlab pipeline. They are all based on the *Makefiles* provided by the system team and included as a git submodule. Furthermore, most of the jobs are imported from a template repository. With this situation, Cream Team developers don't need to maintain neither the pipeline configuration file, nor the logic of the Makefiles' entries. In this case the only effort required is to keep the code compliant with linting and to provide the tests for the code. However, the testing infrastructure of CSP.LMC requires some customization for the CI/CD.

Tests for CSP.LMC are provided as described in section 5. There are some pre-deployment tests, namely unit tests, that check the functionality of the python-code mocking all the external connections to the TANGO layer. On the other side, post-deployment tests, namely integration tests, are checking the TANGO device behavior when it connects to the subsystems. They are written in BDD format. However, integration tests are provided with both real subsystems, and with simulated ones, where the behavior of the devices is externally driven. On one side, having tests with simulated subsystems permits testing CSP.LMC with all the three ones (at the present moment, only CBF is implemented), while on the other side allows testing the component against several fault conditions derived from the subsystems. To run real or simulated subsystems, different umbrella charts are provided

The two different kinds of post-deployment tests are both implemented in the Gitlab Pipeline of *ska-csp-lmc-mid* and *ska-csp-lmc-low*. The templates provided by System Team make it simple to have two different test jobs with different umbrella charts. In fact, the only thing needed is overriding the variable that contains the path umbrella chart as well as the one containing the path of the test definition. A different situation regards the case of ska-csp-lmc-common project, where only simulated subsystems are deployed and integration tests are provided only with them.

Another essential feature of the CI/CD is the publishing of the artifacts. Even in this case the possibility to customize the pipeline has helped the Cream Team to be as efficient as possible. In fact, the *ska-csp-lmc-common* project only provides the code to be included in the python package of the same name. The related oci-image and the helm chart are only used for integration tests and not needed to be published in the Central Artefact Repository. For this reason, the corresponding pipeline jobs are skipped. A different situation is for *ska-csp-lmc-mid* and *ska-csp-lmc-low*: python packages, oci-images and helm charts are published to permit testing and integration with other software components developed by different teams.

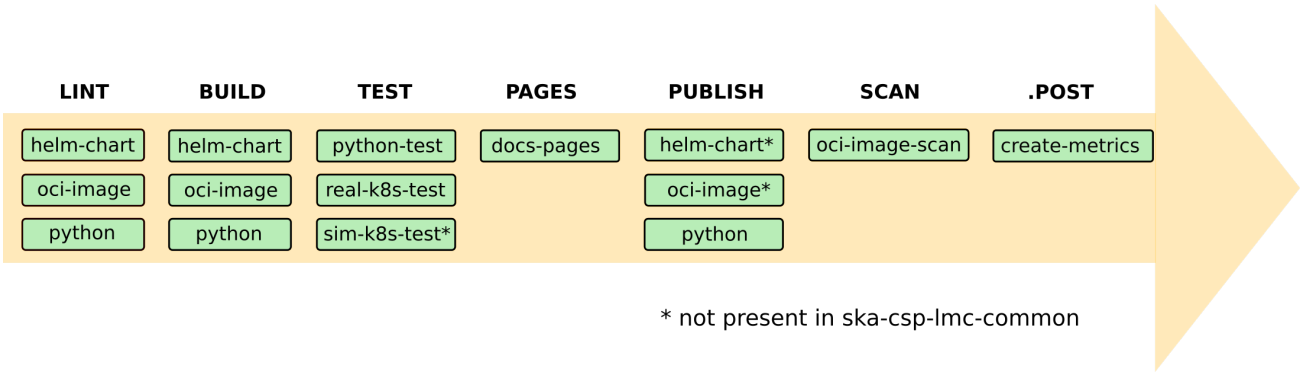


Figure 9. Schematics of the CSP-LMC pipelines

Figure 9 shows a schematic of the CSP.LMC pipelines. Jobs marked with the asterisk are not present for *ska-csp-lmc-common* project.

8. 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.[1–3] In particular:

- For each component of the system, there is only one repository with minimal use of branching that is short-lived;
- build, test 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.

Furthermore, the experience of the CSP.LMC demonstrates that all the tools provided by System Team for the CI/CD are easy to use and can be customized to meet the developer's needs. On the other hand, they give a solid and reliable infrastructure to deliver artifacts compliant to the software guidelines, that is essential for the integration of a complex system, such as SKA.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Fowler, M., "Continuous integration." https://martinfowler.com/articles/continuousIntegration.html. (Accessed: 5 October 2020).

[2] J. Humble, D. F., [*Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*], Addison-Wesley Professional (2010).

[3] at all, G. K., [*The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*] (2016).

[4] "Tango-controls framework." https://www.tango-controls.org/. (Accessed: 5 October 2020).

[5] "Ska-tango-images repository." https://gitlab.com/ska-telescope/ska-tango-images. (Accessed: 5 May 2022).

[6] "Kubernetes." https://kubernetes.io/. (Accessed: 5 May 2022).

[7] "Helm." https://helm.sh. (Accessed: 5 May 2022).

[8] "Gitlab." https://gitlab.com/. (Accessed: 5 May 2022).

[9] "What is gitops?." https://about.gitlab.com/topics/gitops/. (Accessed: 5 May 2022).

[10] "Ska gitlab templates repository." https://gitlab.com/ska-telescope/templates-repository. (Accessed: 5 May 2022).

[11] "Ska cicd makefile repository." https://gitlab.com/ska-telescope/sdi/ska-cicd-makefile. (Accessed: 5 May 2022).

[12] "Gitlab kubernetes runner." https://docs.gitlab.com/runner/install/kubernetes.html. (Accessed: 5 May 2022).

[13] "Minio operator." https://github.com/minio/operator. (Accessed: 5 May 2022).

[14] "Elasticsearch." https://www.elastic.co/. (Accessed: 5 May 2022).

[15] "Ceph storage." https://ceph.io/. (Accessed: 5 May 2022).

[16] "Nexus." https://www.sonatype.com/nexus/repository-pro. (Accessed: 5 May 2022).

[17] "Monitoring the performance of the ska cicd infrastructure," International Society for Optics and Photonics, SPIE (2022).

[18] "Nginx." https://www.nginx.com/. (Accessed: 5 May 2022).

[19] "Oauth2 proxy." https://github.com/oauth2-proxy/oauth2-proxy. (Accessed: 5 May 2022).

[20] "Software defined infrastructure." https://gitlab.com/ska-telescope/sdi. (Accessed: 15 June 2022).

[21] "Ska ci cd deployment on stfc cloud." https://gitlab.com/ska-telescope/sdi/ska-ci-cd-deployment-on-stfc-cloud. (Accessed: 15 June 2022).

[22] "Amazon s3 buckets." https://aws.amazon.com/it/s3/. (Accessed: 5 May 2022).

[23] "pytest." https://docs.pytest.org/. (Accessed: 5 May 2022).

[24] "Gherkin specification language." https://cucumber.io/docs/gherkin/. (Accessed: 5 May 2022).

[25] "Ansible molecule." https://molecule.readthedocs.io/en/latest/. (Accessed: 5 May 2022).

[26] "Bats framework." https://github.com/sstephenson/bats#readme. (Accessed: 5 May 2022).

[27] "Fastapi." https://fastapi.tiangolo.com/. (Accessed: 5 May 2022).

[28] "Celery." https://docs.celeryproject.org/en/stable. (Accessed: 5 May 2022).

[29] "ska-cicd-services-api repository." https://gitlab.com/ska-telescope/sdi/ska-cicd-services-api. (Accessed: 5 May 2022).

[30] "Slack." https://slack.com. (Accessed: 5 May 2022).

[31] "Jira." https://www.atlassian.com/software/jira. (Accessed: 5 May 2022).

[32] "ska-cicd-automation repository." https://gitlab.com/ska-telescope/sdi/ska-cicd-automation. (Accessed: 5 May 2022).

[33] "ska-cicd-artefact-validations repository." https://gitlab.com/ska-telescope/sdi/ska-cicd-artefact-validations. (Accessed: 5 May 2022).

[34] "Redis." https://redis.io/. (Accessed: 5 May 2022).

[35] "Mongodb." https://www.mongodb.com. (Accessed: 5 May 2022).

[36] "Architectural decision register 25 - general software naming convention." https://confluence.skatelescope.org/display/SWSI/ADR-25+General+software+naming+convention. (Accessed: 30 May 2022).
[37] "Semantic versioning 2.0.0." https://semver.org/. (Accessed: 5 May 2022).
[38] "ska-tango-base repository." https://gitlab.com/ska-telescope/ska-tango-base. (Accessed: 15 June 2022).