ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUBL04

CI-CD PRACTICES AT SKA

Di Carlo M.*, Dolci M., INAF Osservatorio Astronomico d'Abruzzo, Teramo, Italy
Harding P.[1], U. Yilmaz, SKA Organisation, Macclesfield, UK
Ribeiro B., Instituto de Telecomunicações Aveiro, Portugal
Morgado J. B., CICGE, Faculdade de Ciências da Universidade do Porto, 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. Continuous Integration is the practice of merging all developers' local copies into the mainline frequently. Continuous Delivery is the approach of developing software in short cycles ensuring it can be released anytime, and Continuous Deployment is the approach of delivering the software into operational use frequently and automatically. This paper analyses the decisions taken by the Systems Team (a specialized agile team devoted to developing and maintaining the tools that allow continuous practices) to promote the CI/CD practices with the TANGO-controls framework.

INTRODUCTION

When creating releases for end-users, every large software endeavour faces the problem of integrating different parts of their software solution and bring them to the production environment. When many parts of the project are developed independently for some time, an integration problem arises when merging them into the same branch, consuming more developer resources than originally planned. In a classic Waterfall Software Development process this is usual but also happens when following the classic Git Flow — also known as Feature-based Branching, which is when a branch is created for a feature. As an example, considering one hundred developers working in the same repository each of them creating one branch, merging can easily lead to conflicts becoming unmanageable, for a single developer to solve, thus introducing a delay in the releases (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.

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.

* matteo.dicarlo@inaf.it

Continuous Integration (CI)

CI refers to a set of practices requiring developers to integrate code into a shared repository often. Each commit is verified by an automated build, allowing teams to detect problems early in the process, giving feedback about the state of the integration. Martin Fowler [1] states various practices in this regard:

- maintain a single source repository for each system's component, favouring the use of a single branch;
- automate the build (possibly all in one command);
- automated test is run after build process for the software to be self-testing (this is crucial: all benefits of CI rely on the test suite being high quality);
- every commit should build on an integration machine: the more developers commit the better it is (common practice is at least once per day);
- frequent commits reduce potential conflicts: developer workflow is reconciled on short windows of change;
- main branch must always be stable;
- builds must be fast so that problems are found quickly;
- multi-stage deployment: every software build must be tested in different environments (testing, staging, etc);
- make it easy to get the latest version: all programmers should start the day by updating their local copies;
- Everyone can see what's happening: a testing environment with the latest software should be running.

Continuous Delivery & Deployment (CD)

Continuous Delivery [2] refers to a CI extension focusing on sustainably automating the delivery of new software releases. Release frequency can be decided according to business requirements, but the greatest benefit is reached by releasing as quickly as possible. Deployment has to be predictable and sustainable, irrespective of whether it is a large-scale distributed system, complex production environment, embedded system, or app. Therefore the code must always be in a deployable state. Testing becomes the most important activity, needing to cover enough of the codebase.

Often, the unsupported fact that frequent deployment equals lower levels of stability and reliability, is assumed. For software, the golden rule should be "if it hurts, do it more often, and bring the pain forward" — [2], page 26.

There are many patterns around continuous deployment related to the DevOps culture [3], "the outcome of applying

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUBL04

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". It fosters increased collaboration between development (requirements analysis, development and testing) and operations (deployment, operations and maintenance) within IT. In the era of mainframe applications was common to have the two areas managed by different teams resulting in a development team with low interest in the operational aspects (managed by a different team) and vice versa. Sharing 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: being empowered to deploy a change to production with no fear of failure. This is only possible by supplying the necessary testing/staging platform and required infrastructure tools for developers to engage with the platforms and by using applications and deployment processes that can be rolled out and reverted if required.

Automation is one of the key elements of a DevOps strategy. It allows teams to focus on what is valuable (code development, test results, etc.) instead of the deployment itself, reducing human errors. The importance of those practises is to reduce risks of integration issues, releasing new software and creating better software products. CD 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 was adopted in the initial design phase of the SKA project to reduce complexity by dividing the project into simpler elements. For every element, an initial architecture was developed, which comprises the software modules needed corresponding to a repository — each a component of the system.

Since all components need to be deployed and tested together, the first decision is how to package them. A container is a standard run-time unit of software that packages code and dependencies so that the component runs quickly and reliably across different computing environments. A *Docker* [4] *container image* is a lightweight, standalone, software package including everything needed to run an application: code, runtime, system tools, system libraries and settings.

One of the main dependencies in the SKA software is the TANGO-controls [5] framework, a middleware for connecting software processes mainly based on the CORBA standard (Common Object Request Broker Architecture). The standard defines how to expose the procedures of an object within a software process with the RPC protocol (Remote Procedure Call). TANGO extends the definition of an object with the concept of a Device that represents a real or virtual device to control. This exposes commands (procedures), and attributes (i.e. state) allowing both synchronous and asynchronous communication with events generated from attributes. Fig.1 shows a module view of the framework.

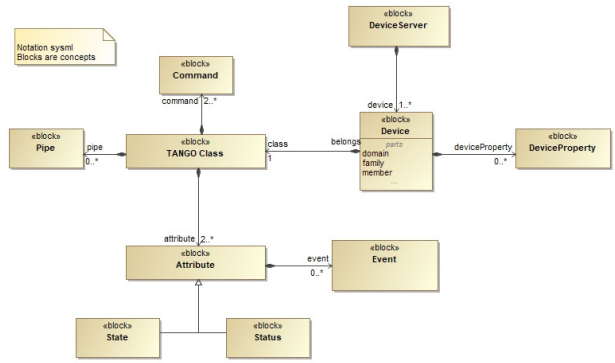

Figure 1: TANGO-controls simplified data model.

The importance of containers becomes clear with dependencies. The entire framework is packaged onto a set of containers [6] so that the final product is a containerized application that will be run in a system for managing these applications. Specifically, there is a SKA repository *ska-tango-images* [6], encapsulating all its components in a set of container images. Fig. 2 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 for all.

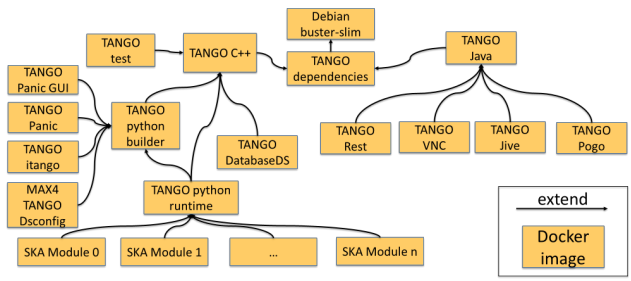

Figure 2: SKA-tango-images repository.

Kubernetes (K8s) [7] is used for container orchestration and *Helm Charts* [8] 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. Helm is a tool for managing K8s deployments with charts — a package of pre-configured K8s resources, tied to run-time instance configuration.

Namespaces create a logical separation of resources within a shared multi-tenant environment. A Namespace

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUBL04

enforces a separate network and set of access rights enabling a virtual private space for contained deployment.

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

ska-tango-base: aplication Helm Chart defines the basic TANGO ecosystem with the following K8s services:

- *tangodb*: MySQL database used to store configuration data used at the startup of a device server;
- *databaseds*: device server providing configuration information to system's components, and runtime catalogue of the components/devices;
- *itango*: interactive TANGO client;
- *vnc*: Debian environment with x11 window system together with vnc [9] and noVNC [10] installed on it;
- *tangorest*: rest api [11] for TANGO eco-system;
- *tangotest*: TANGO test device server [12].

ska-tango-util: library chart helper for defining TANGO device servers on applications. Defines the following Helm templates:

- *multidevice-config*: K8s ConfigMap defining dsconfig JSON configuration file, bootstrap script for dsconfig, and python script for multi-class device server startup;
- *multidevice-job*: job for dsconfig application to apply a configuration JSON file set into the values file;
- *multidevice-sacc-role*: K8s service account, role and role binding that waits for configuration job to finish;
- *multidevice-svc*: K8s service and a K8s StatefulSet for a device server tag specified in the values file.

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 database (device configuration) and how is exposed to other applications for discovery whiting the cluster.

Other SKA repositories *Makefiles* are selected as an abstraction and organisation layer eliminating language-specific scripts for building, testing, deployment and promoting ease of use in CI/CD. The use of a *Makefile* in each project simplifies containerisation work and automation of code building, testing and packaging processes, making it 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.

The Makefile also enables 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.

ARCHITECTURE FOR INTEGRATION

In the previous section, was highlighted how the SKA project can be seen as a set of elements composed by a set of software modules corresponding to a repository. For each repository, one or more container images are built and pushed into the artefact repository (Nexus [13] box shown in fig. 5) while for each element, a Helm Chart is published into the same storage solution.

Since a 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 [14]) in a composable way representing the bulk of the effort for integrating all SKA's software sub-systems. Fig.3 shows a simplistic view of this above concept and its hierarchy.

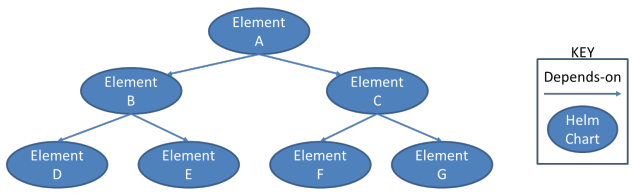

Figure 3: SKAMPI.

It is important to consider the operational aspects of the Helm dependencies which state that when Helm installs/updates a chart, the K8s resources from the chart and all its dependencies are aggregated 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. 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. 4 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 by creating an umbrella chart with sub-elements needed for its integration.

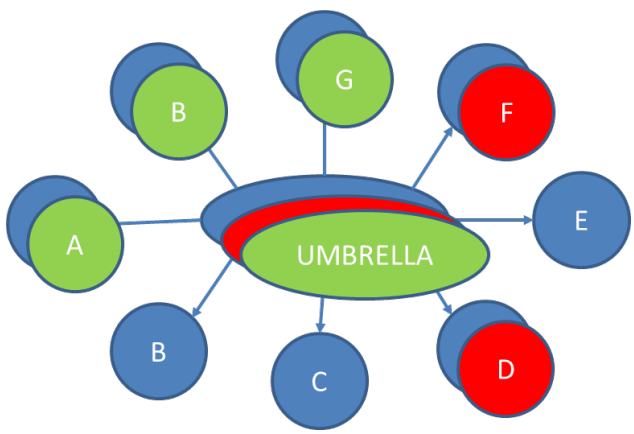

Figure 4: The umbrella chart concept.

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUBL04

SKA INFRASTRUCTURE

To support the integration's architecture an 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 [15] runner architecture, and MVP testing facilities as shown in Fig.5 used to support DevOps and Integration testing facilities.

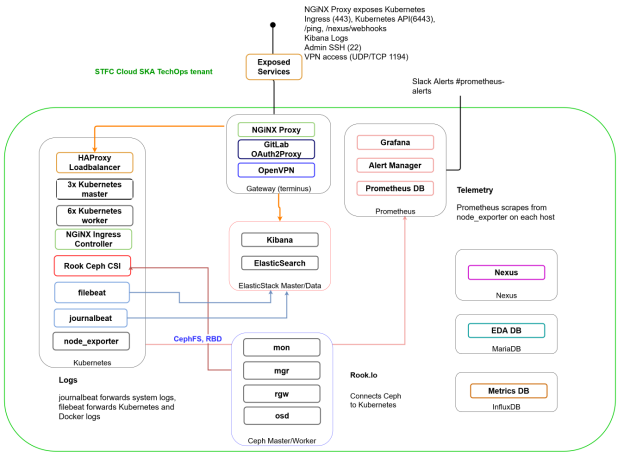

Figure 5: STFC cloud components.

A K8s cluster was deployed with 1 LoadBalancer, 3 Master, and 6 Worker configuration. Fig.6 illustrates how the LoadBalancer ties K8s services together exposing deployed applications to the outside world. The K8s API Server is exposed externally from Terminus using TCP pass-through, and NGiNX [16]'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 [17] integration with GitLab.

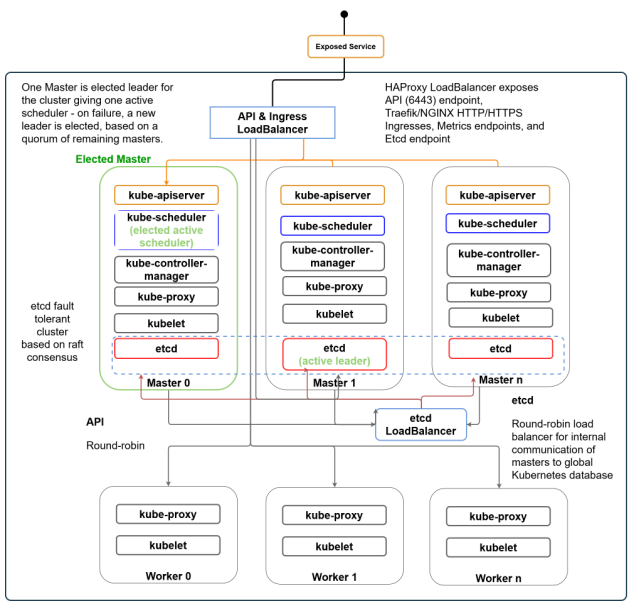

Figure 6: Kubernetes components.

In Fig.7 it is hown the K8s runner [18] works as a multiplexer receiving requests from GitLab for jobs and launching their respective Pods up to a configured scaling limit (15 cur-rently). GitLab's runners use intermediate cache to speed up jobs by passing dependencies between them. This cache is based on Minio [19] with S3 [20] compatible buckets for storage.

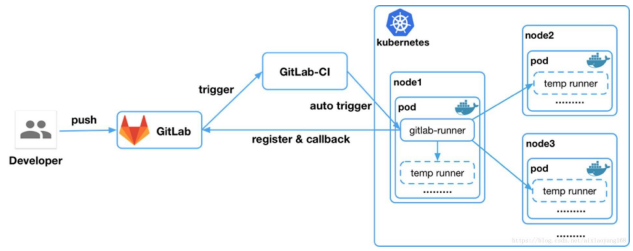

Figure 7: GitLab runner.

PIPELINE

To bring everything together for a complete CI/CD toolchain, GitLab [15] was selected. The data model for a generic SKA software is shown in Figure 8.

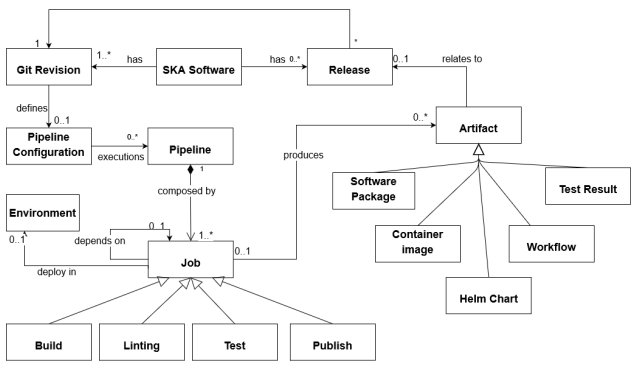

Figure 8: Pipeline definition data model.

The entry point of the diagram is the Pipeline box, composed of several jobs. This was standardised for each project regardless of its artefacts so that the same standardised steps for code/configuration and Helm Charts are followed:

- **linting**: code is analysed against sets of coding rules to check if it follows the agreed best practices;
- **build**: code is compiled and a container image created;
- **test:** compiled package (and image) are tested;
- **publish**: code artefacts are published;
- **pages**: test results are published (GitLab's naming).

The pipeline is respected as the main hub of software development in which code is built, tested, verified, published and integrated. These steps are used in local development (where the same shell scripts are available thanks to the *Makefile* targets), merge workflow, QA, integration and release. Also, having an almost identical platform environment

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUBL04

for different stages of the software lifecycle, significant differences between development and operations are eliminated.

Fig.9 shows a simplification of the run-time behaviour of the selected technologies working together. At the centre part, there is a K8s cluster defined for every project in SKA's telescope. Outside the K8s cluster, are GitLab code repositories and pages, Nexus Artifact repository [13] (store packaged code artefacts), ELK stack (Elasticsearch, Logstash and Kibana) [21] for logging, prometheus [22] for monitoring (metric collection) and Ceph [23] for distributed storage.

Inside the cluster, in an isolated K8s Namespace, are the GitLab runners related K8s resources, checking every 30 seconds if there are pending pipelines triggered manually or pulled by resources. If the runner finds a pipeline, it creates a K8s Pod for each job defined in the configuration file. Each created Pod is capable of deploying an umbrella chart needed for specific testing of the repository in an isolated Namespace. The installed deployment can then be tested and the job's result will be reported to GitLab producing artefacts, to be stored in the correct artefact repository.

During any stage of the pipeline, jobs can download required dependencies from the artefact repository. Depending on the type of job, the pipeline is used for deploying the permanently running version of SKAMPI or other resources needed. The K8s cluster is equipped with monitoring solutions to examine the cluster's health and performance and any resources that are deployed in it. Storage and logging solutions are integrated to provide a consistent logging and distributed storage framework for the resources. Finally, this architecture for creating temporal K8s resources for pipeline steps (testing, building, packaging, etc.) ensures that necessary environments for the jobs are always clean — i.e. not affected by previously run pipelines.

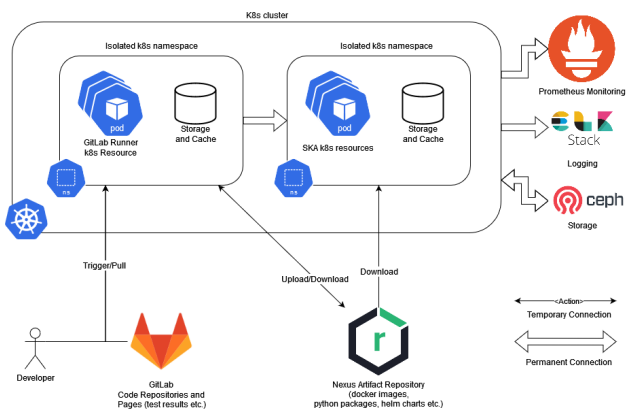

Figure 9: CICD at run time.

TESTING

The most important in CI is testing, so we need to question ourselves how a generic component of the SKA can be tested with the above architecture. At the SKA, testing was split into two distinct types: pre-deployment and post-deployment. Deployment happens when a runner executes a job with a GitLab environment keyword. By doing so, the job is linked to the K8s cluster through GitLab configuration. While pre-deployment tests (unit tests) are made without the real system online (using stubs and mocks), other tests (integration and system tests) need more than one live system component to be up and running as they will be using other services and applications. The SKA is composed of several distinct modules, each of them with its repository and different requirements for the components needed for integration and system testing. For each, an umbrella chart was 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:

- contains at least one Helm Chart;
- has an environment;
- has a Makefile for K8s testing.

A set of templates and standardized Makefiles was developed by the System Team, so it is only necessary to include them in the repository. The post-deployment test job is then composed of the following steps defined in those Makefiles:

- **install**: installs chart (and sub-charts) in the Namespace specified in the environment;
- **wait**: wait for every container to be running;
- **test**:
  - create container in the Namespace specified in the environment;
  - run PyTests inside this container;
  - return test results.
- **post test**: delete all resources allocated for tests.

The artefacts are the output of the tests, containing reports both in XML and JSON and other (i.e. PyTest's) output so that consequent pipeline steps (mostly packaging and releasing) can be run.

DEVELOPMENT WORKFLOW

There are two important assumptions for understanding SKA's development workflow: the master branch shall always be stable, and branches shall be short-lived. Stable 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. As a result, the selected development workflow for SKA is the following:

- developer works on current code base's copy;
- work on a story starts on a new branch named after it;
- developer frequently commits changes to local git repo;

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUBL04

- developer creates unit tests to be run on local environment until they successfully pass;
- once tests pass, changes are pushed to a remote branch;
- CI server (GitLab):
  - checks out changes when they occur;
  - runs static code analysis providing feedback;
  - builds system and runs unit and integration tests on the branch;
  - provide feedback to developer about test status;
  - provide feedback about coverage metrics.
- once all branch's tests execute successfully, the developer opens a pull request (*i.e. GitLab's Merge Request*) for merging changes onto master;
- code is reviewed and approved by other developers;
- code is merged into the master branch;
- CI server (GitLab):
  - runs whole pipeline again including the tests on master branch;
  - releases deployable artefacts for testing (reports, code analysis, etc.);
  - assigns build label the code's version built (i.e. docker image version);
  - alerts team if build or tests fail for them to fix issue ASAP;
  - publishes the successful build artefacts to artefact repository.

CI-CD AUTOMATION AND QUALITY

To verify if all best practices are followed, plugins and tasks were implemented to perform quality checks on GitLab's merge requests and artefacts published to Nexus. Fig.10 shows a module's view of the frameworks for those.

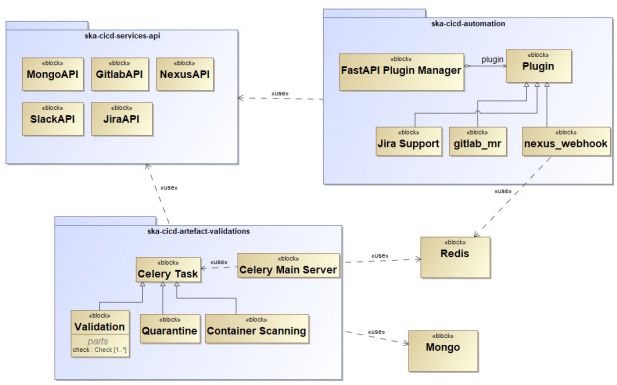

Figure 10: CI-CD automation framework.

The above diagram shows three main packages:

- **ska-cicd-services-api** [24] 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 [25] (i.e. sending messages to channels), Jira [26] (i.e. project information) and Nexus (i.e. obtaining component information);
- **ska-cicd-automation** [27] which uses FastAPI [28] 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** [29] based on Celery [30] containing a main server pulling messages from Redis [31], transforming them into artifact validation tasks and storing the results into a MongoDB [32] database.

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 (fig. 10 *ska-cicd-automation* [27]) every time a new Merge Request is created. The quality checks 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.

After performing the checks, their result is reported back to the developers on GitLab's main Merge Request page via a comment (see Fig. 11 for an example of the comment, includ-ing a table with the severity of the failed check, description about the check and mitigation strategy).

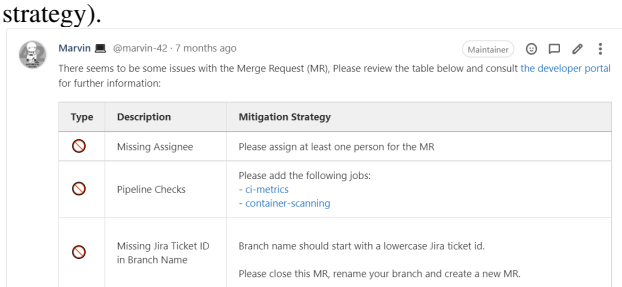

Figure 11: Checks results table.

Nexus Artefact Validation

There are many packaged code artefacts of multiple formats being created in the SKA project hosted on GitLab

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUBL04

and then published to the Nexus Artifact repository. Those should follow SKAO's conventions: Artefact names should be compliant with semantic versioning 2.0.0 [33] and must include associated metadata with the required information, such as who published it, from which GitLab repository is they originate from and other useful information.

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* [29]. 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 on the 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;
- **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.

CONCLUSION

The majority of decisions taken by the Systems Team follow the workflow described by the Continuous Integration process outlined in Martin Fowler's paper inspired by the state-of-the-art industry practices of [1–3]. In particular:

- for each component of the system, there is only one short-lived repository with minimal use of branching;
- artefact build, tests and publishing are automated with the use of few commands (Makefile targets);
- every commit triggers a build in a different machine;
- once artefacts are built, the SKAMPI will automatically create a new deployment of the system and more tests will be done at that level (i.e. system tests);
- having a common repository (Nexus and GitLab pages) for the code artefacts and test results simplifies downloading the latest changes from every team and for each component to enable fast development;
- the integration environment is accessible to all developers and deployed in a unique Namespace on the cluster.

In addition, every artefact is validated in terms of quality so that a common standard across the project is maintained.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] Martin Fowler, Continuous Integration, https://martinfowler.com/articles/continuousIntegration.html

[2] J. Humble, D. Farley, "Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation", 2010, ISBN (0321601912, 9780321601919), Pub. Addison-Wesley Professional

[3] G. Kim, P. Debois, J. Willis, J. Humble, "The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations", ISBN (1942788002 9781942788003)

[4] Docker, https://www.docker.com/

[5] TANGO-controls, https://www.tango-controls.org/

[6] ska-tango-images repository, https://gitlab.com/ska-telescope/ska-tango-images

[7] Kubernetes, https://kubernetes.io/

[8] Helm, https://helm.sh

[9] v11vnc, https://github.com/LibVNC/x11vnc

[10] noVNC, https://github.com/novnc/noVNC

[11] TANGO-controls REST API, https://gitlab.com/tango-controls/rest-api

[12] TANGO-controls test device server, https://gitlab.com/tango-controls/TangoTest

[13] Nexus, https://www.sonatype.com/nexus/repository-pro/

[14] SKAMPI - SKA Mvp Prototype Integration, https://gitlab.com/ska-telescope/ska-skampi

[15] GitLab, https://gitlab.com/

[16] NGINX, https://www.nginx.com/

[17] OAuth2 Proxy, https://github.com/oauth2-proxy/oauth2-proxy

[18] GitLab Runner, https://gitlab.com/ska-telescope/sdi/deploy-gitlab-runners/

[19] MinIO, https://operator.min.io/

[20] Amazon S3 buckets, https://aws.amazon.com/it/s3/

[21] Elasticsearch, https://www.elastic.co/

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUBL04

[22] Prometheus, https://prometheus.io/

[23] Ceph Storage, https://ceph.io/

[24] ska-cicd-services-api, https://gitlab.com/ska-telescope/sdi/ska-cicd-services-api

[25] Slack, https://slack.com

[26] Jira, https://www.atlassian.com/software/jira

[27] ska-cicd-automation, https://gitlab.com/ska-telescope/sdi/ska-cicd-automation

[28] FastAPI, https://fastapi.tiangolo.com/

[29] ska-cicd-artefact-validations, https://gitlab.com/ska-telescope/sdi/ska-cicd-artefact-validations

[30] Celery, https://docs.celeryproject.org/en/stable

[31] , Redis, https://redis.io/

[32] MongoDB, https://www.mongodb.com

[33] Semantic versioning 2.0.0 https://semver.org/