CI-CD PRACTICES WITH THE TANGO-CONTROLS FRAMEWORK IN THE CONTEXT OF THE SQUARE KILOMETRE ARRAY (SKA) TELESCOPE PROJECT*

M. Di Carlo†, INAF Osservatorio Astronomico d'Abruzzo, Teramo, Italy
M. Bartolini, SKA Organisation, Macclesfield, UK
K. Madisa, A. J. Venter, A. de Beer, SKA South Africa, Cape Town, South Africa
J. B. Morgado, D. Bartashevich, D. Nunes, Instituto de Telecomunicações (GRIT)
Universidade de Aveiro, Portugal

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) in order to promote the CI-CD practices with TANGO controls framework.

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, then it is easy to understand what is called the "merge hell" that is when every merge has to deal with conflict parts and it is impossible, for a single developer, to solve all the conflicts thus creating delay in publishing any release. 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.

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. 1 shows a module view of the framework.

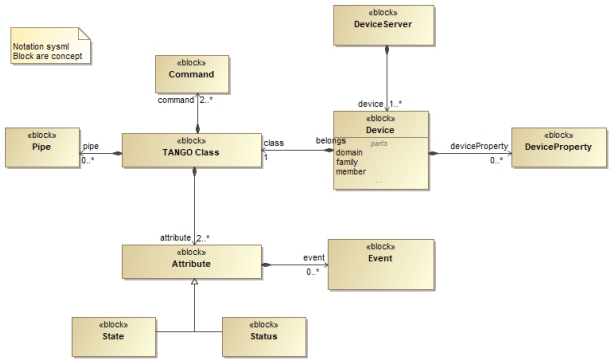

Figure 1: TANGO-Controls simplified data model.

* Work supported by Italian Government (MEF - Ministero dell'Economia e delle Finanze, MIUR - Ministero dell'Istruzione, dell'Università e della Ricerca)
† matteo.dicarlo@inaf.it

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 early. According to Martin Fowler [2], 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.

CONTINUOUS DELIVERY AND CONTINUOUS DEPLOYMENT (CD)

Continuous delivery [3] 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" ([3], page 26). There are many patterns for deployment such as the blue-green deployment (Fig. 2, ([3], page 261)) and, in general, all of them are related somehow to the DevOps culture. According to [4], "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 is an increased collaboration between development (intended as requirement analysis, development and testing) and operations (intended as deployment, operations and maintenance). In fact, it is very common, when those two areas are managed by different teams, that the development team loses interest in the operations aspects, when the software is 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

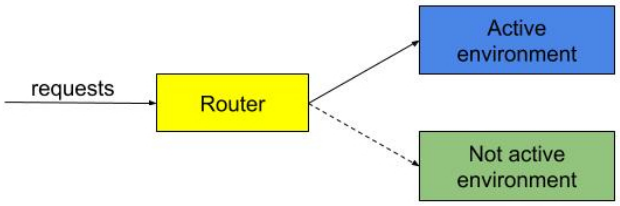

Figure 2: Blue-Green deployment: If the active environment in production is the blue box, it deploys on the green and if everything goes well the router can switch the incoming requests; if an error is discovered is always possible to rollback.

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.

CONTAINERIZATION

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 compo-

nent 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. In order to allow all the teams to work in the same environment, it has been created a containerized Tango environment [5] is summarized in Fig. 3. Every tango project

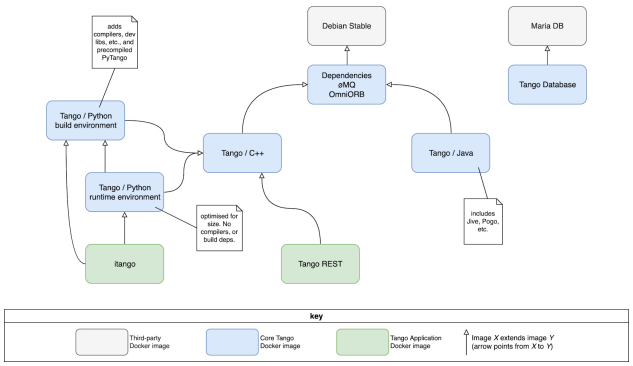

Figure 3: SKA-docker project module view.

comes with a docker-composed based orchestration (docker-compose.yml file) that includes:

- MariaDB service container,
- DatabaseDs service container,
- For each device server in the project, one container defined by the developer.

Each container includes one application (in TANGO terms one device server) and testing is also made within a container (created on the fly) with a standardized (for SKA) metrics file output that states for each project/component the following information:

- Build status: true or false,
- Test report,
- Coverage,
- Linting information.

For the integration of all the docker images (coming from the different repositories), the integration environment is based on Kubernetes (k8s) orchestration [6] and Helm [7]. Helm is a tool for managing Kubernetes charts where a chart is a package of pre-configured Kubernetes resources that are a set of information necessary to create an instance of a Kubernetes application. In specific every chart contains at least information concerning the version of the docker image and the pull policy for the orchestration. In specific every component has a helm chart that contains 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). Fig. 4 summarize the module view of the structure of the repository for the integration of the SKA components.

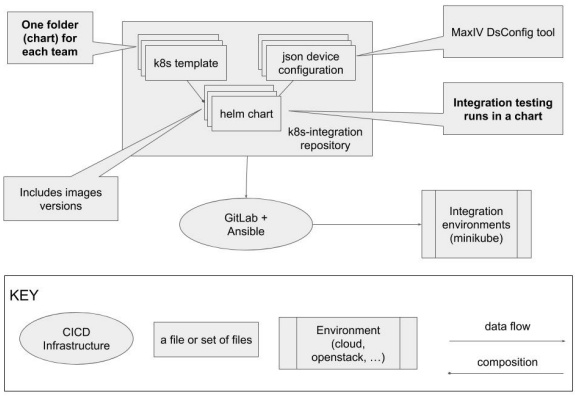

Figure 4: K8s Integration repository module view.

INFRASTRUCTURE

The infrastructure described in this section is possible thanks to 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.

Runtime view

Fig. 5 describes the runtime view of the CICD infrastructure in place for the SKA project. The starting point is the source-code repository server where every team put their code (for instance it can be Github). The CICD Server requests the code from the source code repository periodically and keeps information about the execution of the pipeline (see next section). The CI executor takes the pipeline configuration information and executes it pushing artifacts to the Artifacts repository. If tests are successful the executors starts the deployment of the software artifacts to various environments (like integration, stage and so on) using ansible scripts [8].

Data Entity

The selected tool for CICD is Gitlab where the CI executor is called “runner” and the CICD server is a code repository that includes a description (in a yaml file) of the execution that has to be done by the runners. Gitlab includes also an artifacts repository called “pages” for each repository. Fig. 6 summarize the data model of the SKA CI-CD infrastructure in a UML diagram.

The entry point of the diagram is the Pipeline that is composed by many jobs. This has been standardised for each project in:

- Build, where code is compiled and a docker image is created;

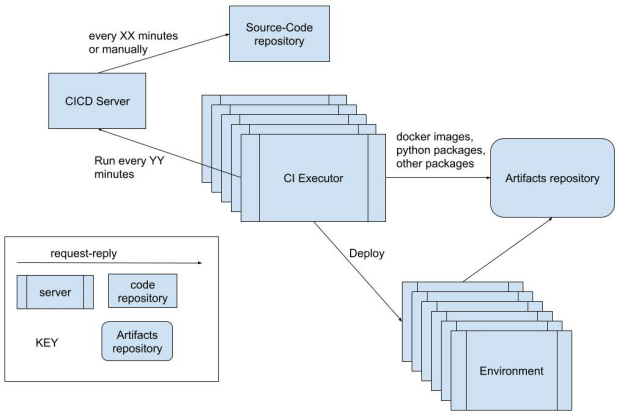

Figure 5: Runtime view of the SKA CI-CD infrastructure.

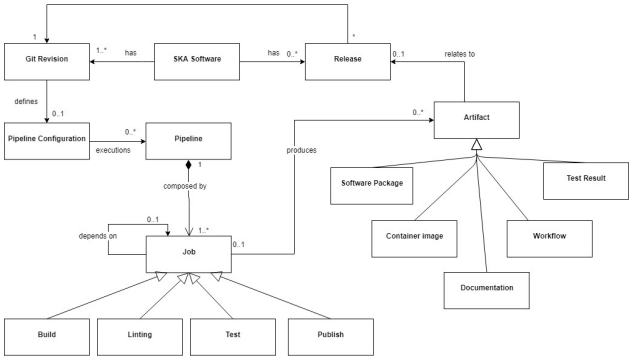

Figure 6: Data Model of the SKA CI-CD infrastructure.

- 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 and logs are published (the name comes directly by the Gitlab technology).

In the Gitlab cloud space the artifacts stored are the logs of each job executed and the test results obtained by the Test job. Other artifacts (python packages, docker images and so on) are stored in a different repository based on the Nexus tool. For each git revision there is a version of the pipeline configuration (stored in the repository with the name ".gitlab-ci.yml").

DEVELOPMENT WORKFLOW

There are two possible workflow in order to make a change to a git repository whether there is branch or not. In general, two concepts are important to the SKA way of using GIT:

- The master branch of a repository shall always be stable.
- Branches shall be short lived, merging into master as often as possible.

Stable means that the master branch shall always compile and build correctly, and executing automated tests with success. 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.

Every commit triggers a pipeline build and there may be different rules applied to determine which stages are executed in the pipeline based on factors like the branch name. According to the data model, every pipeline job is associated with its git commit (including tag commits) and developers have to ensure that the stages complete as fast as possible. For this reason it is possible to parallelize jobs, for example unit tests and static analysis could be run in parallel. Developers needs to keep all tests working on the "master" branch that must be kept stable. When working on a development project, it is important to stick to these simple commit rules:

- Commit often.
- Git logs shall be human readable in sequence, describing the development activity.
- Use imperative forms in the commit message.

Workflow for master based development

- A developer takes a copy of the current code base on which to work
- Work is started on a local copy
- As the developer advances in the implementation commits are done on the local git repo.
- Unit tests are written and run in the development environment until successfully executed
- Once the tests pass the developer pushes the changes into a remote repository
- The CI server
  - Checks out changes every X minutes (or when they occur)
  - Runs static code analysis and provide feedback to the developers
  - Builds the system
  - Runs all tests
  - Releases deployable artefacts for testing (reports, code analysis, etc.)
  - Assigns a build label to the version of the code it just built (i.e. docker image version)
  - Alerts the team if the build or tests fail which fixes the issue asap
  - Provide feedback about coverage metrics

Workflow for story based branching

- A developer takes a copy of the current code base on which to work
- Work is started on a new branch based on the story being implemented
- As the developer advances in the implementation commits are done on the local git repo.

- Unit tests are written and run in the development environment until successfully executed
- Once the tests pass the developer pushes the changes into a remote branch
- The CI server
  - Checks out changes when they occur
  - Runs static code analysis and provide feedback to the developer
  - builds the system and runs unit and integration tests on the branch
  - Provide feedback to the developer about test fail or success
  - Provide feedback about coverage metrics
- Once all tests execute successfully on the branch, the developer makes a pull request for merging the changes into master.
- As part of the pull request the code is reviewed by other developers.
- Code is merged into master branch
- The CI server
  - Runs all tests on the master branch
  - Releases deployable artefacts for testing (reports, code analysis, etc.)
  - Assigns a build label to the version of the code it just built (i.e. docker image version)
  - Alerts the team if the build or tests fail which fixes the issue asap

DASHBOARD

From a management point of view it is very important to monitor the projects in SKA with at least the following information:

- Latest build status and date
- Latest green build date
- Test coverage
- Testing report

In order to do that, a dashboard has been created as a progressive web app (PWA). this kind of web application are loaded like regular web pages or websites but can offer user functionalities such as working offline, push notifications, and device hardware access traditionally available only to native applications. In specific, PWAs combine the flexibility of the web with the experience of a native application. In particular progressive means that it works for every user, regardless of browser choice because they're built with progressive enhancement as a core tenet. Other important qualities are:

- Responsive, it works for desktop, mobile, tablet;.
- Connectivity independent: it allows work offline
- App-like: feel like an app to the user with app-style interactions and navigation;
- Fresh: always up-to-date thanks to the service worker update process
- Safe: HTTPS

The deployment are based on a pipeline with two main jobs:

- Retrieve data, from Gitlab rest api download a set of json files (one per each page)
- Publish data, that generate the web page.

CONCLUSION

All the decisions taken by the system team try to follow 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;
- The build of each component is automated together with tests that allows to push only the correct artifacts into the artifact repository;
- Every commit triggers a build in a different machine (not the local developer machine);
- Once the artifacts are built, they are transferred in an integration repository which run a kubernetes cluster and more tests are done;
- Having a common repository 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.

There is some work still to be done, the first and most important is improving the performance of the tests. At the moment, tests are executed after the code is compiled and the docker images are built. A possible improvement is building the docker images after the tests are executed. Besides, more environments have to be created. For this purpose a kubernetes cluster is going to be used so that it will be possible to isolate each environment adding for each of them the specific resources needed in terms of GPUs or other resources. Having kubernetes will also make very easy to implement the blue-green deployment pattern.

ACKNOWLEDGEMENTS

This work has been made possible thanks to the financial support by the Italian Government (MEF - Ministero dell'Economia e delle Finanze, MIUR - Ministero dell'Istruzione, dell'Università e della Ricerca). This research is also supported by the project Enabling Green E-science for the SKA Research Infrastructure (ENGAGE SKA), reference POCI-01-0145-FEDER-022217, funded by COMPETE 2020 and FCT, Portugal.

REFERENCES

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

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

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

[4] 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)

[5] SKA-docker repository, https://gitlab.com/ska-telescope/ska-docker

[6] kubernetes, https://kubernetes.io

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

[8] Ansible, https://www.ansible.com/