Proposal for Team Badger’s new Testing Repos Setup

Proposal for Team Badger’s new Testing Repos Setup

Emanuele Lena, Giorgio Brajnik (review)

Structure Overview

https://app.diagrams.net/#G1WSbiwjthHvYx1vKMlynJYJIPx5LkmFPk#%7B%22pageId%22%3A%22WiD_wVFLmC4MBV2jh7vP%22%7D

General Idea (short)

Inspired to what is the SKA standard: https://gitlab.com/ska-telescope/ska-tango-examples

But taking into account the 2 requirements of:

1. versioning separately the SUT and the tests
2. Having mid and low tests in the same repo

NOTE: this will be in replacement of the current solution used in https://gitlab.com/ska-telescope/ska-sw-integration-testing (and in the related deployment repos)

Solution overview:

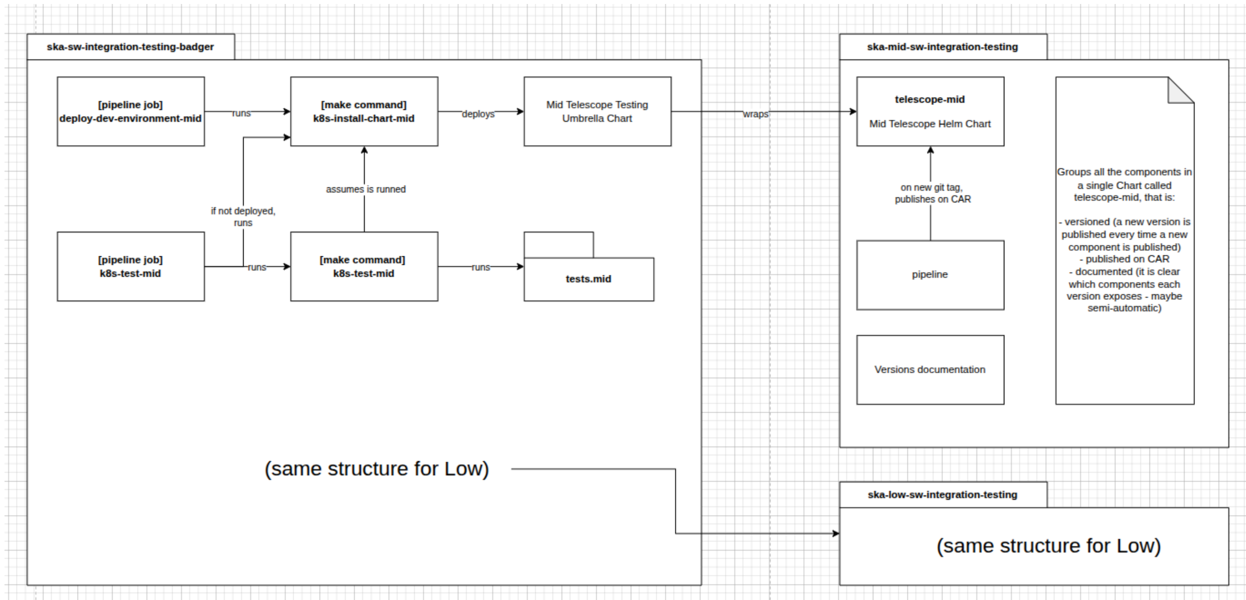
- We keep 3 repos
- Two repos (one for Mid, one for Low) define and version what the SUT is
    - https://gitlab.com/ska-telescope/ska-mid-sw-integration-testing
    - https://gitlab.com/ska-telescope/ska-low-sw-integration-testing
- These Two repos contain the charts with the combination of components with their versions (as the current Release Manager repos are doing now)
    - (No tests here, as per requirement)
- Differently from now, these repos use the standard commands and pipeline jobs to:
    - Build the chart
    - **Release it as a versioned CAR artefact, named something like “telescope-low”** → *we track explicitly which are the various deployments that could be tested*
    - (maybe) document which components versions are used by each release
- The possibility to manually deploy it remains but:

- The deployment is done in a branch-dependant namespace
    - It is **not** necessary for testing!
- A single repo contains the tests, both for Mid and for Low
    - https://gitlab.com/ska-telescope/ska-sw-integration-testing-badger
- Differently from now, this repo:
    - Defines **two umbrella charts**, that point to the released “telescope” version from the CAR for mid and low→ *one can update the pointed SUT in a single code line, it’s easy to switch and it’s not technical*
    - Defines **two tests jobs** that **extend the standard ones and will apply on on-the-fly installations of respectively mid and low**, as the standard process does → *every test run from the pipeline points on an independent SUT deployment, handled transparently as it’s done in most of SKAO repos*

- *(eventually - if needed - we set up mechanisms to deploy once and repeat the tests on that deployment, but the deployment happens in the testing repo through the umbrella chart and is done semi-automatically by the same pipeline that runs the test)*

- *(eventually - if needed - the tester may run the umbrella chart manual deployments in the cloud and connect to them through Coder or the VPN; again, without switching repo)*

Diagram



Pipeline Jobs

Pipeline Jobs in the SUT definition repos:

- Lint
  - k8s-lint
- Build
  - Docs-build
  - Helm-chart-built
  - ~~oci-image-build (?) <-- not sure if needed~~
- deploy (manual)
  - deploy-dev-environment (manual)
  - info-dev-environment (manual)
  - stop-dev-environment (manual)
  - test-dev-environment (manual)
- Publish
  - (?) <-- what exactly publishes a chart on the CAR? When is it triggered?

Pipeline jobs in the testing repo:

- lint
  - python-lint

    - k8s-lint (?) <-- do we need it? Do we need two distinct, one for mid and one for Low?
- build
    - docs-build
    - helm-chart-built (?) <-- do we need it? Do we need two distinct, one for mid and one for Low?
    - oci-image-build (?) <-- not sure if needed
- deploy (manual) <-- maybe we need to duplicate for mid and low... (sounds not that good tho)
    - deploy-dev-environment (manual)
    - info-dev-environment (manual)
    - stop-dev-environment (manual)
    - test-dev-environment (manual)
- test
    - k8s-test <-- duplicate for sure for mid and low

Usage Scenarios

Usage Scenario #01: I am a Tester, I write a new test and I want to run it using the pipeline

Preconditions:

- a combination of components is already published in the CAR as chart

Steps:

1. The tester creates a branch in the testing repo
    1. OPTIONAL: the tester changes a flag to run just Mid or just Low to avoid overhead, he will have to remember to change that flag before completing the MR
2. The tester writes a new test
3. The tester pushes the changes
4. The pipeline (transparently) deploys the SUT on-the-fly
5. The pipeline runs the tests
6. The pipeline (transparently) tears down the on-the-fly deployment
    1. or maybe not and the deployment stays on for the next run (?)

Usage Scenario #02: I am a Tester and I want to test a new component version

Preconditions:

- some tests are defined and available

Steps:

1. The tester creates a new branch in the repo with the SUT chart
2. The tester updates a component version in the chart
3. The tester ships the new combination components as a new version of the telescope chart in the CAR (new tag)
4. The tester creates a branch in the testing repo
5. The tester runs the pipeline, configuring an ENV variable so the new chart version is used
6. The pipeline creates a JIRA XRAY report that automatically references the new chart version (and with a link is easy to see which were the tested components)

Usage Scenario #03: I am a tester and I want to debug a tedious test (or a part of the test harness) with CLI-triggered test runs on a unique deployment

Preconditions:

- a combination of components is already published in the CAR as chart
- the tester already has a branch in the testing repo
- the tester has access to SKAO Coder *or* to SKAO VPN

Steps:

1. The tester opens his testing repo work branch in a suitable Coder dev environment
    1. ALTERNATIVE: the tester connects activates SKAO VPN in his machine
2. The tester triggers the pipeline in his branch (in the testing repo)
3. The tester runs a manual job that deploys the SUT in the cloud (in the testing repo)
    1. the pipeline deploys the SUT in a branch-dependant kube namespace, accessible in the SKAO cloud
4. The tester configures from his dev environment a single ENV variable to make the test job "point" to the deployment
5. The tester runs the cli test command (make k8s-test)
    1. the command deploys the testing pod in the cloud deployment (that already contains the SUT)
    2. the tests runs against the cloud deployment
6. [OPTIONAL] we can repeat the process without re-deploying
7. [OPTIONAL] the tester connects to the deployment using notebooks to inspect the SUT

Advantages

Advantages:

- Way closer to the standard process used in many other repos
  - *→ more reliable, more maintainable*
  - *→ easier to obtain support from system team when needed*
- Components combinations are formally versioned and the versions are published in the CAR
  - *→ the process can even be automated with some kind of triggers in future (if we need)*
  - *→ a tester can pick one version, know exactly which components were involved and which version they have and test it with existing tests, eventually creating a formal report that include an exact direct reference*
- A Tester can work on a single repo, without worrying about manually deploying. He just need to point to the SUT chart name with a single variable that defines the version
- Potentially, release managers can work separately from testers and they don't need to actually deploy anything
  - *→they can formally define which combination of versions should be "packaged" and tested*

How to use Badger Repo

How to use the Badger Repo

Edit, run and debug tests from Coder

- Run the pipeline and deploy the system
    - To deploy the system you click at the manual stage “Stage: deploy” (the one with “>>” symbol), then run the manual job “deploy-dev-environment”
    - It will likely take some minutes, so go on with the rest, meanwhile
- Open https://coder.k8s.stfc.skao.int/workspaces and login with Gitlab
- Create a workspace (or open an existing one you already set up):
    - Template: Kubernetes
    - You likely can use default settings
- If you created a new workspace, to set it up do the following:
    - Open it with the terminal, or code-server or Visual Studio Code Desktop (whatever will let you to have a terminal)
        - *Tip: if you work with some kind of VSCode environment, open a* ***bash*** *terminal instead of an* ***sh*** *terminal. Much better to use (you will have all the very useful keyboard shortcuts/controls you are probably used to)*
    - Install ssh utils (to create a private key and configure it for Gitlab)
        - sudo apt update
        - sudo apt install ssh
    - Create a SSH key (accept default settings) and inspect public key
        - ssh-keygen -t rsa # create ssh key
        - chmod 600 /home/tango/.ssh/id_rsa # ensure the private key is protected
        - cat .ssh/id_rsa.pub
    - Copy the key, open https://gitlab.com/-/user_settings/ssh_keys and add it to your keys (this will permit you to push your commits)
    - TODO: check if the second time you open Coder ssh is still installed, otherwise add a note to tell to re-install
    - TODO: check if we can avoid to install SSH and instead just run git clone, let it fail and then use the key it gives me
- Clone the repository, ensuring you do it using the ssh name and open it with
    - The repo handler to use: git@gitlab.com:ska-telescope/ska-sw-integration-testing-badger.git
    - From a terminal, you can use:
        - git clone git@gitlab.com:ska-telescope/ska-sw-integration-testing-badger.git
    - Or directly from code-server or Visual Studio Code Desktop (better, because then you will have to open the project with that)
- Open the repository in coder, through code-server or Visual Studio Code Desktop and do the following usual setup operations.
    - Init all submodules (essentially, the make submodule that gives you access to)

        - `git submodule update --init --recursive`
    - Pull and checkout to the branch you are using to work (if different than **main**)
        - git fetch -p # check if remotely some branches have been deleted
        - git branch -r # print remote branch names
        - git switch -t origin/branch-name # connect to the branch
- Configure a few key env variables to make your coder instance communicate with the deployment and verify the deployment is reachable
    - Inspect the deploy job and search for the Kubernetes Namespace and Tango Host. You should see some lines like these:

      SKA Deployment completed. To connect to the deployed Tango system use:
      export KUBE_NAMESPACE='ci-dev-ska-sw-integration-testing-badger'
      export
      TANGO_HOST='tango://tango-databaseds.ci-dev-ska-sw-integration-testing-badger.svc.techops.internal.skao.int:10000'

    - Export them in the coder Visual Studio bash:
      export KUBE_NAMESPACE='ci-dev-ska-sw-integration-testing-badger
      export'
      TANGO_HOST='tango-databaseds.ci-dev-ska-sw-integration-testing-badger.svc.techops.internal.skao.int:10000'
    - Check the deployment is active (it should work as if called after deploying a system locally)
        - make k8s-wait
- When a deployment is active, to inspect it you can:
    - List the pods:
        - kubectl get svc -n ci-dev-ska-sw-integration-testing-badger
    - Get the log of a specific pod:
        - kubectl logs POD_NAME -n ci-dev-ska-sw-integration-testing-badger
- If everything is okay, you can directly run the tests via pytest
    - Export further needed env variables (for now)
      export MCCS_SIMULATION_ENABLED="false"
      export SDP_SIMULATION_ENABLED="false"
      export CSP_SIMULATION_ENABLED="false"
    - Run the tests directly via pytest:
        - poetry run pytest -m "system_level_tests" -x ./low/tests/
    - If everything is configured correctly you should be able to make modifications to your test code and repeat the test run
- If everything is configured correctly, you should be able commit and push your modification
    - NOTE: if you push you may need to repeat the deployment.