developer.skatelescope.org Documentation

Release 0.1.0-beta

Marco Bartolini

Feb 17, 2021

Home

Welcome to the Square Kilometre Array software documentation portal. Whether you are a developer involved in SKA or you are simply one of our many users, all of our software processes and projects are documented or linked to in this portal.

The portal is frequently updated as the project evolves; if you feel that something is missing, please have a look at our *guide to contributing to the developer portal*

If you're new to developing the SKA, please have a look at *our Onboarding material* and the *guideance on setting up your development environment*.

Note: Please also read the *SKA Code of Conduct*, which governs all SKA interactions.

What follows is a brief guide to the headings you'll find in the left-hand sidebar of this site. Feel free to explore!

CHAPTER 1

Getting Started and the SKA Developer Community

This section is about getting you up and running. It contains the onboarding material for all new SKA developers, the general contribution guidelines when working on SKA projects, guidance on setting up your development environment, and a list of projects, so you know what the SKA is working on. There is also a wealth of information about our development tools and practices, which you can read as you start development work.

1.1 Onboarding: Welcome to the SKA developer community

Welcome! This page is designed to help people who have just started working on SKA software find out more about the Square Kilometre Array (SKA), the way the SKA project is being run, and how to develop software for the SKA. This page consists of links to relevant information, and a suggested timeline to guide you.

If you're working in an SKA Agile Team, then your Scrum Master will assist you in getting involved in the daily development work of the SKA.

If you're working at the program or solution level, the Solution Engineer or your Release Train Engineer will take the lead.

If you're joining as a Scrum Master, either the team's existing Scrum Master or the Release Train Engineer will help you.

If you're an external contributor, then please look at the SKA Code of Conduct and the Contribution Guidelines for the GitLab project you intend to contribute to.

You may also wish to read through the relevant coding guidelines, testing policy, and some of the information on CI/CD.

Don't worry if many of the terms in the previous paragraphs don't make sense – by the time you've covered all the material on this onboarding page, you will at least know where to look up such terms, even if you can't immediately give a definition!

This page shows you things to do (or try to do!) on your first day, then things to do in your first week, first month, and first three months. The list gets less prescriptive as time goes on – we expect that you'll be working with your team to set goals, assess further training needs and so on. You may want to print this out, and also take a copy of the skills matrix spreadsheet, so that you can track what you've done and what remains to be done. Your Scrum Master may create JIRA tickets to help you and your team organise and track this work. Your Scrum Master will usually be your first port of call when you have questions; you'll learn other ways of getting your questions answered as you go through the onboarding process.

In this page, we talk about your Scrum Master doing various things – if your onboarding isn't led by a Scrum Master, then this should be done by whoever is leading your onboarding.

We welcome feedback on this page - please tell us what worked and what didn't! You can leave comments in the #proj-onboarding slack channel, or you can contribute changes to the developer.skatelescope GitLab project, or you can mail the Solution Engineer or Release Train Engineers with feedback.

1.1.1 First Day

This section is designed to get you up and running, with all the accounts you'll need and the main places to go to get further information.

Note: Scrum Master Action Needed! Scrum Masters: you should try to arrange accounts for new team members ahead of time, if you know they will be joining. You will also need to budget a fair amount of time to help new people integrate with your team; you may want to create stories or features to help with this!

- Get SKA accounts. You should set up these accounts with your institutional email address. Please provide this email address to your Scrum Master if they don't have it already! You'll need the following accounts:
  - JIRA and Confluence login (both sites are configured to authenticate you with the same username and password).
  - Slack If the person you are trying to add does not have an institutional email address, please drop by #team-slack-admins and indicate why the account needs to be done on a non-institutional email address.
  - Google Drive. You may need to create an account associated with your institutional email address. This makes it easier for us to know who is using that Google account.
  - GitLab (If you're not going to be contributing code or documentation, this step may be omitted or postponed.) Please make sure your institutional email address is associated with your GitLab ID if you've already got a GitLab account. See https://developer.skatelescope.org/en/latest/tools/git.html#use-institutional-email for instructions on how to do this.

Please be aware of the Tool Access page, where the processes for requesting access to these tools is outlined. There is also very informative Guidance page created by the SKAO IT team that you might want to consult.

Now you've got some accounts, you should log in to:

- JIRA. JIRA is a ticket tracking system; we use it to organise the SKA software work.
- Confluence. This is the SKA's internal wiki. You may want to look at the Software Engineering pages.
- Slack - visit your team channel! You will be added to the SKA's default slack channels: #announcements and #rand-chats. Reading the 'Pinned' items on the #announcements channel will help you in using the SKA Slack workspace more effectively.

You should read the SKA Code of Conduct. The SKA is an inclusive organisation, and we want to make sure that everyone is respected and has a pleasant experience working for SKA. We therefore have a Code of Conduct, and reading, understanding, and adhering to that Code is a key part of your onboarding and your future work at SKA.

And you should visit the developer portal! (You may be reading this on the developer portal; this is a reminder to Scrum Masters that they should provide you with the link!)

1.1.2 First Week

This week is about providing you with some context about the project, and working out what your immediate training needs are. You will learn more about the SKA in the following weeks and months: some of this is just a brief introduction. You can use the links provided here as a way to find information in the future, should you wish to revisit some of this material.

Information about the SKA

- What is the SKA?
  - Have a look at the SKA website.
  - Read about the SKA and its history.
- How does the SKA operate? How is it meant to work?
  - Read about the SKA Operational Programme.
- What is Radio Astronomy?
  - Our Senior System Scientist, Robert Laing, recorded some lectures. We suggest that you listen to the first and possibly the second lecture now. If you're interested or need to know more about radio interferometry, you can look at the other lectures in the next few months. Or come back later on – you may discover a need to know more about how radio telescopes work in a few years!
- What is SAFe?
  - This is a very good question! We have some slides to tell you what SAFe is.
  - You should also visit the Training Events Confluence page, and discover when the next appropriate SAFe training sessions will happen. You will probably want to attend the SAFe for Teams training, but please discuss this with your Scrum Master. The training could happen at any point in the next three months, and some of it will be revision of what you've learnt from reading the slides.
- What are we building? * The Solution Intent Confluence pages describe the software architecture of the SKA. It also describes the direction in which we intend the architecture to evolve. We suggest you look at the top-level views, and then look at the specific views for the part of the system that your team is working on. Again, at this stage, you're only looking at a small section of the information. You can return later to get a better idea of the wider context. Your Scrum Master will help you identify the relevant pages.
- How are we building it?
  - We have a guide to the way SKA currently works to develop software. This tells you a bit about what teams we have and what we are doing.
  - You'll also want to look at the Operations Context, to find out how the software fits in to the operational environment of the running telescope.
  - Check out the timeline of the software project, so you know roughly what the SKA is planning to do when, and where we are in the process of building a world-leading Radio Astronomy Observatory.
  - Coming soon: the SKA Software Security policy!
  - We encourage you to use ssh to push your changes GitLab. GitLab tells you how to set up ssh keys.
  - We also expect you to sign your commits. GitLab provides instructions on how to create a GPG key and use it to sign your commits. If you already have a GPG key, the same page tells you how to associate it with your GitLab account.
- Finally, there's a Glossary. This lists many of the terms and acronyms in use in SKA. Also, don't be afraid to ask your team on Slack, or ask questions in meetings if you don't understand.

Information about People

Your Scrum Master should introduce you to your team, and other people you'll meet in the course of your work. You can find out about the people who work for the SKA Organisation from the SKA website. You'll find out more about the people working on the software later on.

Socialising

You'll get to know people a bit through the various meetings SKA holds, and your own institution probably has some social events that you can participate in. For SKA, we currently have the #social-boardgames slack channel, the #rand-chats channel, and a lunchtime speaker series.

Information about your Team

Your Scrum Master should give you links to:

- Your team's Google Drive space
- Your team's Confluence area. Each team has a space in Confluence. All of the teams are listed in the Agile Release Train pages.
- Key SKA Confluence Calendars and instructions on how to copy them to your own calendar.
- The main Slack channels in use. You should join your team's slack channel. We also suggest that you join some of the help channels (they all start #help-) , #announcements, #system-demos-buzz and #rand-chats. You may find other channels to join later on!

If you are not employed by the SKA Organisation, you'll probably want to set up a Zoom account. While you *can* use Zoom from your web browser, we use Zoom so much that you'll probably find it easier to have your own account. We also sometimes use Slack for conversations between individuals, but most major SKA events are conducted using Zoom.

If you are employed by the SKA Organisation, you may have a Zoom account associated with your SKA email address. The SKA IT team should be able to assist.

We also recommend that you set up a Miro account. You can use it as a guest for many applications, but it's sometimes useful to sign up with your institutional email address. If you are a new Scrum Master or Product Owner, you will definitely need a Miro account! You can familiarise yourself with Miro by playing in this sandbox.

Training

This may be the first time you've used JIRA or Confluence. Both of these have extensive help pages, which can be accessed by clicking on the question mark in the top right of the screen. The links change every time the software is updated, but the question mark icon will always link to the latest version. As a rough guide, you should be comfortable editing and creating new Confluence pages, and creating and updating JIRA tickets. If you're not, then spend some time with the documentation. SKA Confluence has a dedicated Demonstration space for you to test things out.

If you've already used JIRA or Confluence before, we recommend reading the JIRA and Confluence Usage Guidelines to find out how we're using them specifically in the SKA. It's also worth talking to your Scrum Master to find out how your team is using JIRA and Confluence. In general, developers are empowered to raise issues, but there is then a process to prioritise that activity, so that we're working on the most critical issues first. But that's just common sense.

You'll probably have a lot of questions at this point. Your Scrum Master is your first port of call, but they may also encourage you to talk to someone else on the team, or someone else in the SKA. If you're having trouble with particular tools, the slack help channels may be of use as well.

1.1.3 First Month

This month is about getting you to the stage where you're able to contribute to your team's work. In each subsection, items are approximately ordered by priority, so things earlier in the list should usually be done before things later in the list.

First of all, you should familiarise yourself with the *SKA Definition of Done*

Understanding more about the SKA

- Have a look at the SKA organisation chart (it's linked on the bottom of the right-hand sidebar on the staff page, and find out where you fit in.
- Find out about your ART (Agile Release Train). An overview of the structure will give a general picture. Then you should look at one or other of
  - the DP ART
  - the OMC ART.
  - Have a look at the pages on Program Increment and Cadence; they will tell you about the regular planning and evaluation cycles of the SKA.
  - And look at the operational flow. The goal here is to find out where your team fits in the organisation, but with a bit more detail than we had time for in week one.
- Look at the Module Decomposition of the SKA, and learn how this maps to the different GitLab project.
- Read the *Architectural Decision Process </policies/decision-making>*. This process is how we can change and update our architecture, as we find out more about the system we're implementing, or as we need to adopt new technology. All developers are able to reason about the architecture of the system; you'll need to know the process.

What are your skills?

Now you know more about what your team does, and where it fits in the organisation, we suggest you look at the SKA skills matrix. You'll now work out with your Scrum Master which skills you need to do your job. We recommend taking a copy of the skills spreadsheet and putting it in your team area in Google Drive.

The skills are approximately grouped by difficulty and how frequently you might need to do the activity. The "Advanced" sections often require using different skills together to produce the desired result. Then assess whether you need to do some training or learning so that you can do your work confidently. Your Scrum Master may create some JIRA tickets to help manage this. You can return to this matrix at various points in your SKA work, to use it as a guide when you need to learn new topics.

The skill gradation is only approximate. Some frequently-needed activities may be classed in the "basic" section of the skills matrix, even if they're conceptually a bit more difficult, simply because we expect you'll need to use them very frequently to work in that area. The more advanced tasks may require knowledge across multiple domains. We've tried to arrange these topics in a moderately logical order, leading from skills everyone needs, through to more specific and/or complex skills that may not be needed by everyone. Then there are a few sections on general programming skills. This arrangement can only be approximate; there are many ways to arrange this, and the order in which you tackle these is something you should discuss with your Scrum Master. We do recommend that everyone makes sure they can do the basic tasks in JIRA, Confluence, and Zoom.

You should work through the skills specified by your Scrum Master, and see wheter you can do the associated activity. Even if you can do the activity, you may need to do some reading to find out how the SKA does things. You can also sign up for training on the Confluence training pages.

Suggested Activities

These are some things we think you might want to do. Discuss this with your Scrum Master to see which ones are most appropriate for you.

- Join a Community of Practice (CoP). CoPs span the two Agile Release Trains (ARTs), and are a good way of sharing expertise, connecting with the wider community and making a contribution.
- Continue watching the Radio Interferometry lectures.
- Get involved in a team's feature. This may be as a developer, reviewer, tester, by shadowing a Feature owner, helping with a demo, or something else!
- Learn about (or get!) access to the *EngageSKA Cluster </tools/test-infrastructure>*, or access to HPC facilities for testing, prototyping and performance testing. People on the DP ART are more likely to need to access the HPC facilities for performance testing; most developers will need to be aware of how the EngageSKA cluster is used for testing. You may also need to arrange access to the SKA Data Store.
- Create or amend some SKA documentation, whether on the Developer Portal, Confluence, or in a specific GitLab project.
- Attend a system demo. You can find out more about demos in the #system-demos-buzz Slack channel, or in the Demos pages in Confluence.
- Sign up for some SKA-organised training. We expect that you'll need to attend some SAFe training; now is a good time to sign up!
- Watch some parts of videos of recent demos that describe the part of the system you're working on; your Scrum Master should be able to recommend suitable demos.

Suggested activities for new developers

This section is primarily aimed at new developers. Your Scrum Master may create tickets in JIRA; this will help you get used to managing your work via JIRA if this is new to you.

- Commit to an SKA project on GitLab. This may be as simple as fixing a typo in some documentation. We recommend that projects, especially projects where we expect external people to contribute, keep a list of easy issues to fix, as they're a good way in to a project. You'll need to look at how to branch your code. That page will tell you how to name your branch.
- Create a Merge Request (MR) on GitLab. You'll need to do that if you've committed a change!
  - Include the JIRA ticket number in the commit
  - Write a good commit message!
- Review someone else's code on GitLab.
- Read your team's documentation for the main project you're working on.

Suggested activities for other roles

Scrum Masters: lead a stand up, then a review and retrospective session, and a planning meeting!

Product Owners: create new tickets for your team. Remember that we want measureable outcomes, and the Definition of Done.

Members of the Solution or Program Management: attend feature development workshops as soon as you can. Also talk to the teams, and find out what they think they're doing.

1.1.4 First Three Months

These months are about filling out your knowledge of the project. Because some things happen on a 3-monthly cycle in the SKA, some of these events may be earlier or later in your onboarding. There will probably be training opportunities during the first 1-3 months, so some may technically happen in your first month if that's when the training is offered. We hope they're useful whenever they happen.

The training events and the suggested reading also provide an opportunity to revisit some of the topics you looked at in your first week or month, but now you'll have more context, and you can dive into a bit more detail.

Remember that we have training pages on Confluence!

- Attend an SKA Onboarding session.
- Attend SAFe for Teams training.
- Give a demo or lightning talk!
- Learn about ECPs (Engineering Change Proposals). These are often required for major architectural changes, so it's useful to understand the purpose and process of ECPs.
- Continue with your training plan, using the skills matrix!
- Make sure you know where to get help. This was covered in week one, but some revision may be helpful.
- Have a look at the various Monitoring Dashboards for the EngageSKA Cluster, so you can see what things look like when our prototype is running, and what data we are collecting about it.
- Learn about the SKA naming conventions for code, repositories, containers, etc. We need to make our code and the artefacts built from it easy to understand, so we have some standards to adhere to, and some recommendations.

This is the end of your formal onboarding! We hope that you've now got an idea of what the SKA is, what we're doing, and how you fit in. We hope that you've started making contributions to your team, and that you know some people in SKA who can help you out. We hope that you've learnt a lot, and that you've now got enough information to know where to go to learn more or get more training in the future.

We hope that you enjoy working with us!

1.2 Contributing to the SKA

Coming soon…

1.3 Configuring your development environment

1.3.1 Python and Tango development

A completely configured development environment can be set up very easily. This will include TANGO, PyTANGO, docker and properly configured IDEs.

PyCharm and VSCode are two IDEs that can be configured to support python and PyTANGO development activities. You will find detailed instructions and how-tos at:

Tango Development Environment set up

Definition

The Development Environment is the set of processes and software tools used to create software.

Tools include:

- python version 3.7
- TANGO-controls '9.3.3'
- Visual Studio Code, PyCharm Community Edition
- ZEROMQ '4.3.2'
- OMNIORB '4.2.3'

Processes include:

- writing code,
- testing code,
- packaging it.

The main process is a python/c++ developer working with one tango device server writing one or more devices:

1. (optional) Work with pogo and create the device(s) needed;
2. Work with a text editor (such as pycharm or VSCode);
3. The tango framework is running locally (with docker) together with other runtime application (generally other devices) needed for the specific development so that the developer can test the device(s) just created;
4. In order to test the work done, the developer creates unit tests (with pytest);
5. The developer creates (if needed) a document for non-trivial testing (for instance for integration testing, usability testing and so on) if the test automation is not possible;
6. The developer creates (if not done before) the docker file in order to ship its work and execute it in a different environment (GitLab CI infrastructure); note that this step can be deleted if the docker file is already available for some kind of development (i.e. for python tango devices the docker file can be the same for every project;
7. The developer creates the file '.gitlab-ci.yml' for the GitLab CI integration;
8. The developer tests the project in GitLab.

Prerequisites

- VirtualBox installed
- git

Creating a Development Environment

Download a image of ubuntu 18.04 like the following one:

- https://sourceforge.net/projects/osboxes/files/v/vb/55-U-u/18.04/18.04.2/18042.64.7z/download

Run the box and call the following commands:

```
sudo apt -y install git
git clone https://gitlab.com/ska-telescope/ansible-playbooks
cd ansible-playbooks
sudo apt-add-repository --yes --update ppa:ansible/ansible && \
    sudo apt -y install ansible
ansible-playbook -i hosts deploy_tangoenv.yml \
    --extra-vars "ansible_become_pass=osboxes.org" \
    -e ansible_python_interpreter=/usr/bin/python
sudo reboot
```

Start the tango system

In order to start the tango system, just call the following commands:

```
cd /usr/src/ska-docker/docker-compose
make up
make start tangotest
```

Other information

Please visit the following gitlab pages for more information:

1. https://gitlab.com/ska-telescope/ansible-playbooks.
2. https://gitlab.com/ska-telescope/ska-docker

PyCharm

PyCharm is a recommended IDE for developing SKA control system software.

Two versions of PyCharm are available: a free community edition, and a payware professional version. Both editions can be downloaded from the PyCharm download page.

PyCharm Professional Docker configuration

These instructions show how to configure PyCharm Professional for SKA control system development using the SKA Docker images. PyCharm can be configured to use the Python interpreter inside a Docker image, which allows:

- development and testing without requiring a local Tango installation;
- the development environment to be identical to the testing and deployment environment, eliminating problems that occur due to differences in execution environment.

Follow the steps below to configure PyCharm to develop new code and run tests for the tango-example project using the Docker images for the project.

Prerequisites

Make sure that the following prerequistes are met:

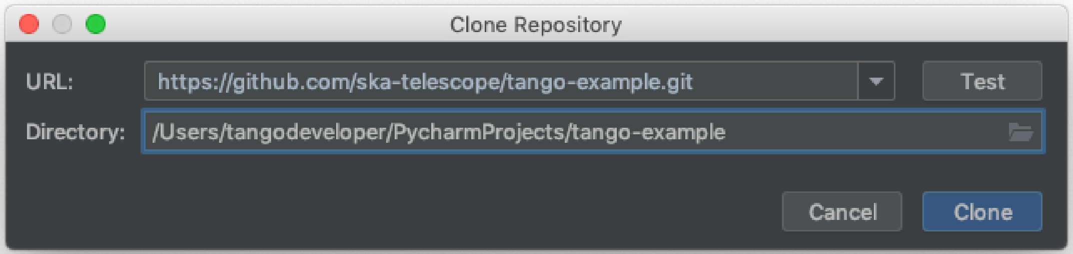
- Docker is installed, as described on the page Docker Docs.
- PyCharm Professional must be installed. *PyCharm Community is not sufficient!*

- You have basic familiarity with PyCharm. If this is the first time you have used PyCharm, follow the First Steps tutorials so that you know how to use PyCharm to develop, debug, and test a simple Python application using a local Python interpreter.

Clone the tango-example project

PyCharm allows you to check out (in Git terms clone) an existing repository and create a new project based on the data you've downloaded.

1. From the main menu, choose VCS | Checkout from Version Control | Git, or, if no project is currently opened, choose Checkout from Version Control | Git on the Welcome screen.
2. In the Clone Repository dialog, specify the URL of the tango-example repository (you can click Test to make sure that connection to the remote can be established).
3. In the Directory field, specify the path where the folder for your local Git repository will be created into which the remote repository will be cloned. The dialog should now look similar to this:


Clone Repository

URL: https://github.com/ska-telescope/tango-example.git

Directory: /Users/tangodeveloper/PycharmProjects/tango-example

4. Click Clone, then click Yes in the subsequent confirmation dialog to create a PyCharm project based on the sources you have cloned.

Build the application image

With the source code source code checked out, the next step is to build a Docker image for the application. This image will contain the Python environment which will we will later connect to PyCharm.

Begin a terminal session in the cloned repository directory and build the image:

```
mypc:tango-example tangodeveloper$ make build
docker build  -t nexus.engageska-portugal.pt/tango-example/powersupply:0.1.0-65c0927 .
↪ -f Dockerfile --build-arg DOCKER_REGISTRY_HOST=nexus.engageska-portugal.pt --build-
↪arg DOCKER_REGISTRY_USER=tango-example
Sending build context to Docker daemon  450.6kB
Step 1/4 : FROM nexus.engageska-portugal.pt/ska-docker/ska-python-buildenv:latest AS␣
↪buildenv
latest: Pulling from ska-docker/ska-python-buildenv
177e7ef0df69: Pull complete
d9178ba39f54: Pull complete
a1c86587108f: Pull complete
072891bac9fb: Pull complete
f7ec90efdf53: Pull complete
```

(continues on next page)

(continued from previous page)

```
877eee992e82: Pull complete
eb71e945bf43: Pull complete
6b50707e167c: Pull complete
6bb56dff13ba: Pull complete
8c3fe19826ab: Pull complete
4377cf316b50: Pull complete
209febb6128f: Pull complete
41eb9ed8ebf6: Pull complete
Digest: sha256:a909606b3d0d4b01b5102bd0e4f329d7fd175319f81c8706493e75504dd0439e
Status: Downloaded newer image for nexus.engageska-portugal.pt/ska-docker/ska-python-
↪buildenv:latest
# Executing 3 build triggers
 ---> Running in c98b60355c16
Installing dependencies from Pipfile.lock (48af56)...
Removing intermediate container c98b60355c16
 ---> 52007c1fb364
Step 2/4 : FROM nexus.engageska-portugal.pt/ska-docker/ska-python-runtime:latest AS␣
↪runtime
latest: Pulling from ska-docker/ska-python-runtime
177e7ef0df69: Already exists
d9178ba39f54: Already exists
a1c86587108f: Already exists
072891bac9fb: Already exists
f7ec90efdf53: Already exists
0f3a4ec2943c: Pull complete
Digest: sha256:9adf4810777d14b660b99fbe2d443f8871cc591313c8ac436dacee38de39160e
Status: Downloaded newer image for nexus.engageska-portugal.pt/ska-docker/ska-python-
↪runtime:latest
# Executing 6 build triggers
 ---> Running in edf8f96df923
Removing intermediate container edf8f96df923
 ---> Running in 246002732edf
Removing intermediate container 246002732edf
 ---> 1ac7b8a31b0f
Step 3/4 : RUN ipython profile create
 ---> Running in 6eccb0302ab8
[ProfileCreate] Generating default config file: '/home/tango/.ipython/profile_default/
↪ipython_config.py'
Removing intermediate container 6eccb0302ab8
 ---> d428fd337258
Step 4/4 : CMD ["/venv/bin/python", "/app/powersupply/powersupply.py"]
 ---> Running in e667e6c25b0b
Removing intermediate container e667e6c25b0b
 ---> 76e5e0e2e4b9
[Warning] One or more build-args [DOCKER_REGISTRY_HOST DOCKER_REGISTRY_USER] were not␣
↪consumed
Successfully built 76e5e0e2e4b9
Successfully tagged nexus.engageska-portugal.pt/tango-example/powersupply:0.1.0-
↪65c0927
docker tag nexus.engageska-portugal.pt/tango-example/powersupply:0.1.0-65c0927 nexus.
↪engageska-portugal.pt/tango-example/powersupply:latest
mypc:tango-example tangodeveloper$
```

The last lines of terminal output displays the name and tags of the resulting images, e.g.,

```
...
Successfully built 76e5e0e2e4b9
```

(continues on next page)

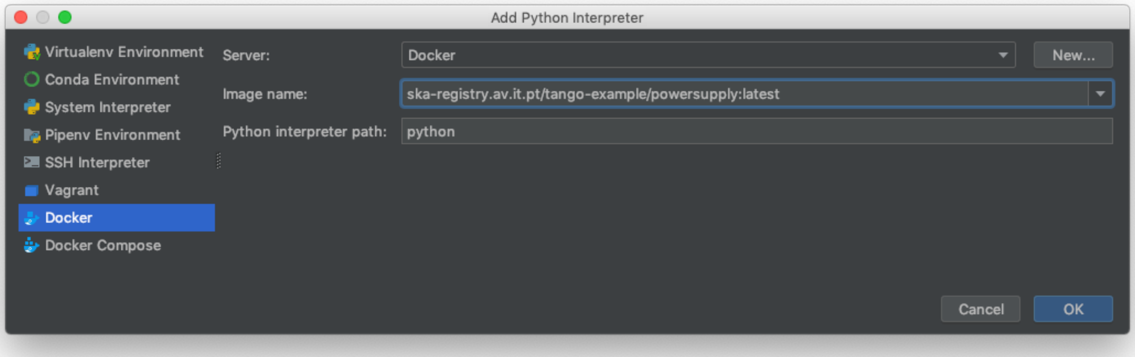
(continued from previous page)

```
Successfully tagged nexus.engageska-portugal.pt/tango-example/powersupply:0.1.0-
↪65c0927
docker tag nexus.engageska-portugal.pt/tango-example/powersupply:0.1.0-65c0927 nexus.
↪engageska-portugal.pt/tango-example/powersupply:latest
```

In the example above, the image name is tagged as *nexus.engageska-portugal.pt/tango-example/powersupply:0.1.0-65c0927* and *nexus.engageska-portugal.pt/tango-example/powersupply:latest*. Take a note of the image tagged as *latest* as you will need it when configuring the remote interpreter.

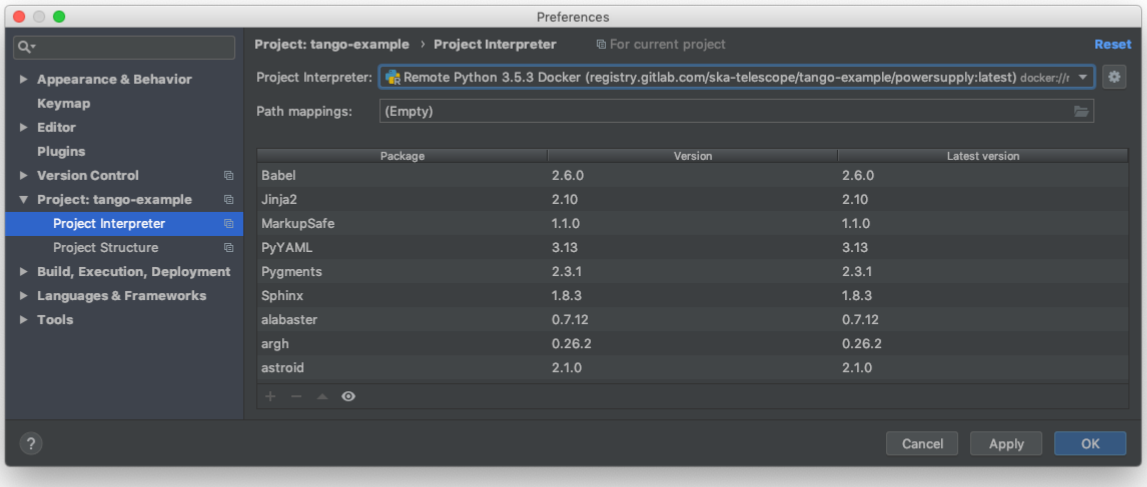
Configure the remote interpreter

Following the official PyCharm documentation, configure Docker as a remote interpreter using the image you just created. The 'Add Python Interpreter' dialog should look similar to this:



Server: Docker
Image name: ska-registry.av.it.pt/tango-example/powersupply:latest
Python interpreter path: python

As a result, the Python interpreter Preferences dialog should look something like this:



Project Interpreter: Remote Python 3.5.3 Docker (registry.gitlab.com/ska-telescope/tango-example/powersupply:latest) docker://r
Path mappings: (Empty)

Package	Version	Latest version
Babel	2.6.0	2.6.0
Jinja2	2.10	2.10
MarkupSafe	1.1.0	1.1.0
PyYAML	3.13	3.13
Pygments	2.3.1	2.3.1
Sphinx	1.8.3	1.8.3
alabaster	0.7.12	0.7.12
argh	0.26.2	0.26.2
astroid	2.1.0	2.1.0

Click 'OK' to apply your changes.

Note: It is recommended to use the remote interpreter in the image tagged as *:latest* rather than the image tagged with a git hash, e.g., *:0.1.0-65c0927*. The *:latest* version will always point to the most recent version of the image, whereas the hash-tagged image will be superceded every time you rebuild.

You can now navigate through the project. As an exercise, open the source code for the PowerSupply class, which is defined in powersupply/powersupply.py. Notice that the IDE notifications and intellisense / code completion are now based on information gathered from the remote Docker interpreter. Below an import statement, try typing `from tango import` and activate code completion (ctrl+space). Notice how the tango packages installed in the Docker image are suggested to complete the statement.

Whenever you change the Python environment, for example by adding or removing dependencies in Piplock, after rebuilding the Docker image you should regenerate the project skeletons to make PyCharm aware of the changes. To do this, select File | Invalidate Caches / Restart… from the main menu.

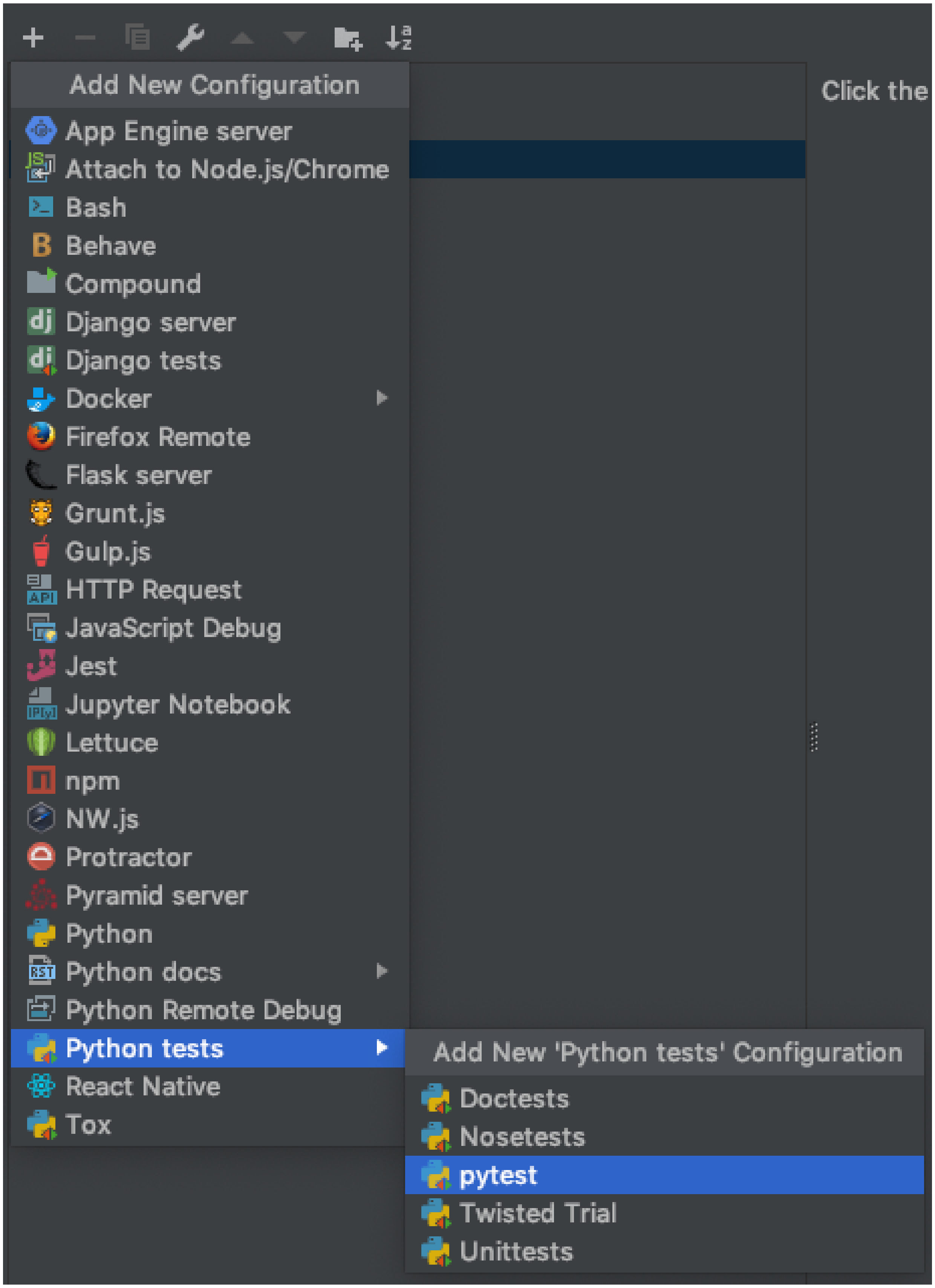
Running unit tests

The tango-example project illustrates two types of unit test:

1. Self-contained unit tests that execute within the confines of a single Docker container. These tests use the Tango class *DeviceTestContext*, which provides a mock connection to a Tango database. In the tango-example project, these tests are found in *tests/test_1_server_in_devicetestcontext.py*.
2. Unit tests that exercise a device in a real Tango environment, with connections to a Tango database and other devices. utilise require a connection. In the tango-example project, these tests are found in *tests/test_2_test_server_using_client.py*.

This tutorial illustrates how to run the self-contained unit tests described in 1. The second type of unit tests require a *docker-compose* PyCharm configuration, which is not described here.

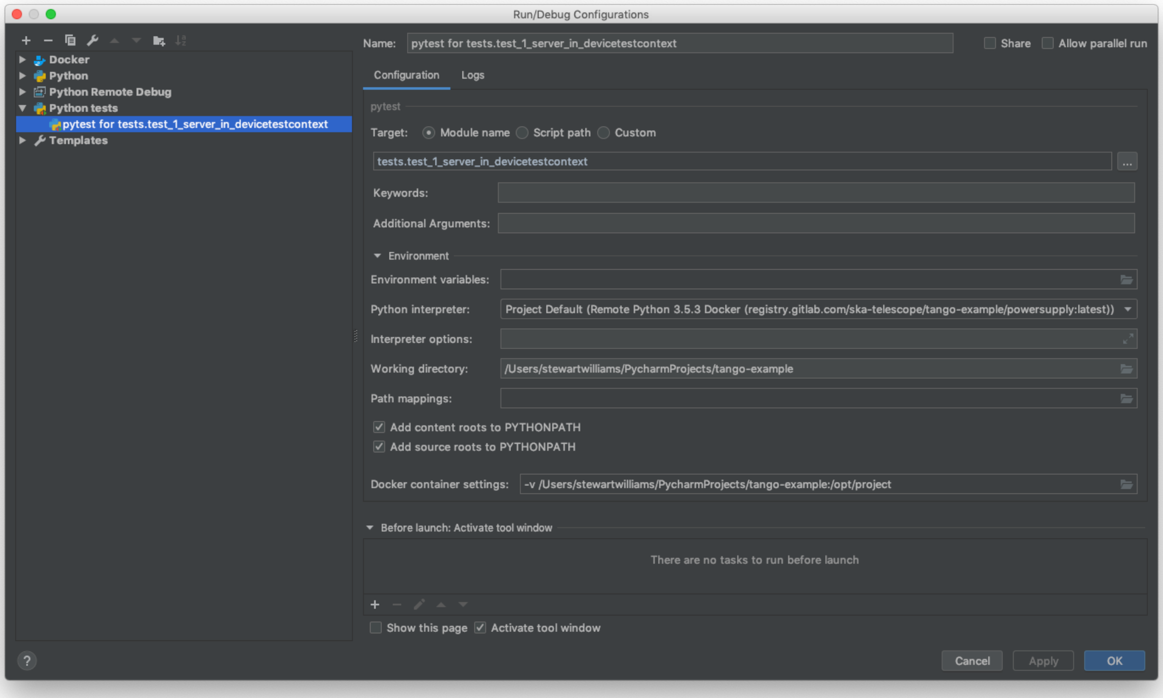
From the main menu, choose Run | Edit Configurations… and click on the '+' button to add a new configuration. From the menu that appears, select Python tests | pytest to add a new pytest test configuration. The menu selection looks like this:


Add New Configuration
App Engine server
Attach to Node.js/Chrome
Bash
Behave
Compound
Django server
Django tests
Docker
Firefox Remote
Flask server
Grunt.js
Gulp.js
HTTP Request
JavaScript Debug
Jest
Jupyter Notebook
Lettuce
npm
NW.js
Protractor
Pyramid server
Python
Python docs
Python Remote Debug
Python tests
React Native
Tox
Add New 'Python tests' Configuration
Doctests
Nosetests
pytest
Twisted Trial
Unittests
Click the

1. Change the Target radio button to 'Module Name'. Click '...' to select the target, choosing *test_1_server_in_devicetestcontext* as the module to be tested.
2. Select 'Project Default' as the Python interpreter for this configuration.

Note: If you change the project default interpreter to another configuration - a Docker Compose configuration, for instance - then you may want to revisit this run/debug configuration and explicitly select the Docker *:latest* interpreter rather than use the project default.

The configuration dialog should look like similar to this:



Name: pytest for tests.test_1_server_in_devicetestcontext

Target: [✓ Module name] Script path Custom

tests.test_1_server_in_devicetestcontext

Keywords:

Additional Arguments:

Environment variables:

Python interpreter: Project Default (Remote Python 3.5.3 Docker (registry.gitlab.com/ska-telescope/tango-example/powersupply:latest))

Interpreter options:

Working directory: /Users/stewartwilliams/PycharmProjects/tango-example

Path mappings:

☑ Add content roots to PYTHONPATH

☑ Add source roots to PYTHONPATH

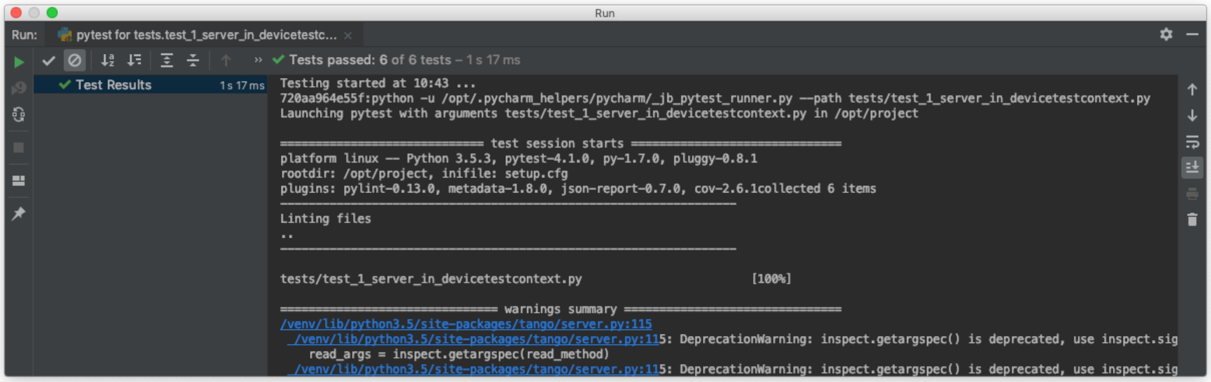
Docker container settings: -v /Users/stewartwilliams/PycharmProjects/tango-example:/opt/project

There are no tasks to run before launch

☐ Show this page ☑ Activate tool window

Click 'OK' to accept your changes.

From the main menu, choose Run | Run..., then from the Run dialog that opens, select the configuration you just created. The unit tests will execute, with the results displayed in PyCharm's Run panel. The results will look like this:



Tests passed: 6 of 6 tests – 1 s 17 ms

```
Testing started at 10:43 ...
720aa964e55f:python -u /opt/.pycharm_helpers/pycharm/_jb_pytest_runner.py --path tests/test_1_server_in_devicetestcontext.py
Launching pytest with arguments tests/test_1_server_in_devicetestcontext.py in /opt/project

============================= test session starts ==============================
platform linux -- Python 3.5.3, pytest-4.1.0, py-1.7.0, pluggy-0.8.1
rootdir: /opt/project, inifile: setup.cfg
plugins: pylint-0.13.0, metadata-1.8.0, json-report-0.7.0, cov-2.6.1collected 6 items
-----------------------------------------------------------------
Linting files
..
-----------------------------------------------------------------

tests/test_1_server_in_devicetestcontext.py                        [100%]

=============================== warnings summary ===============================
/venv/lib/python3.5/site-packages/tango/server.py:115
  /venv/lib/python3.5/site-packages/tango/server.py:115: DeprecationWarning: inspect.getargspec() is deprecated, use inspect.sig
    read_args = inspect.getargspec(read_method)
  /venv/lib/python3.5/site-packages/tango/server.py:115: DeprecationWarning: inspect.getargspec() is deprecated, use inspect.sig
```

Debugging Configuration

Note: The *coverage* module is not compatible with the PyCharm or Visual Studio Code debugger and must be disabled before any debugging session. Do so by editing *setup.cfg*, commenting out the `addopts=...` line of the tool:pytest section so that it looks like this:

```
[tool:pytest]
testpaths = tests
#addopts = --cov=powersupply --json-report --json-report-file=htmlcov/report.
↪json --cov-report term --cov-report html --cov-report xml --pylint --
↪pylint-error-types=EF
```

PyCharm has a *debug* mode that allows breakpoints to be added to code and the runtime state of the device examined. Refer to the official PyCharm documentation for comprehensive documentation on how to add breakpoints and run in debug mode.

The steps in the official documentation can also be used to debug and interact with ah Tango device, using the configuration set up in the previous section as the basis for the debug configuration. However, full breakpoint functionality requires some workarounds. Breakpoints set outside device initialisation code (i.e., outside `__init__()` and *init_device()*) only function if the Tango device uses asyncio green mode. In non-asyncio modes, Tango creates new Python threads to service requests. Unfortunately these threads do not inherit the debugging configuration attached by PyCharm.

For working breakpoints, there are two solutions:

1. the device must be converted to use asyncio green mode;
2. add `pydevd` to your Piplock as a project dependency, rebuild the Docker image and refresh the project skeletons, then add `pydevd.settrace()` statements where the breakpoint is required. For example, to add a breakpoint in the `PowerSupply.get_current()` method, the code should look like this:

```
def get_current(self):
    """Get the current"""
    import pydevd
    pydevd.settrace()  # this is equivalent to setting a breakpoint in IDE
    return self.__current
```

Troubleshooting

- **SegmentationFaults when using DeviceTestContext**

  Unit tests that create a new DeviceTestContext per test must run each DeviceTestContext in a new process to avoid SegmentationFault errors. For more info, see:

  – https://gitlab.com/tango-controls/pytango/pull/77

  – http://www.tango-controls.org/community/forum/c/development/python/testing-tango-devices-using-pytest/?page=1#post-3761

- **Errors when mixing test types**

  Running DeviceTestContext tests after test that use a Tango client results in errors where the DeviceTestContext gets stuck in initialisation. One workaround is to set the filenames so that the DeviceTestContext tests run first.

PyCharm Professional docker-compose configuration

These instructions show how to configure PyCharm Professional to use a docker-compose configuration, which allows development and testing of devices that requires interactions with other live devices. Follow the steps below to configure PyCharm to run tests from the tango-example project that require a live Tango system.

Prerequisites

We recommend you be comfortable with PyCharm and the standard PyCharm Docker configuration before using docker-compose.

Please follow the steps in the *PyCharm Professional Docker configuration* topic for some basic familiarity with PyCharm and Docker.

Goal

In the Docker configuration, PyCharm used the Python interpreter inside the *powersupply:latest* image for development and testing.

The docker-compose.yml file for the tango-example project declares three containers:

1. tangodb: the relational database used for the Tango installation.
2. databaseds: a container running the device service for the Database.
3. powersupply: a container running the PowerSupply device itself.

We want PyCharm to take the place of the powersupply container, so that tests execute against the code we are developing. As a result, unit tests should execute in an additional container created for the duration of the test.

Registering the device server

Tango devices must be registered so that other devices and clients can locate them. The docker-compose.yml entry for the powersupply service automatically registers the powersupply device. However, this automatic registration is not performed when PyCharm launches the service Manual device registration is easily performed from an interactive session. The example below registers a test instance of the powersupply device:

```
mypc:tango-example tangodeveloper$ make interactive
docker build  -t nexus.engageska-portugal.pt/tango-example/powersupply:0.1.0-067e5b3-
↪dirty . -f Dockerfile --build-arg DOCKER_REGISTRY_HOST=nexus.engageska-portugal.pt -
↪-build-arg DOCKER_REGISTRY_USER=tango-example
Sending build context to Docker daemon  914.4kB
Step 1/4 : FROM nexus.engageska-portugal.pt/ska-docker/ska-python-buildenv:latest AS␣
↪buildenv
# Executing 3 build triggers
 ---> Using cache
 ---> Using cache
 ---> Using cache
 ---> 10811adeaf7d
Step 2/4 : FROM nexus.engageska-portugal.pt/ska-docker/ska-python-runtime:latest AS␣
↪runtime
# Executing 6 build triggers
 ---> Using cache
 ---> Using cache
```

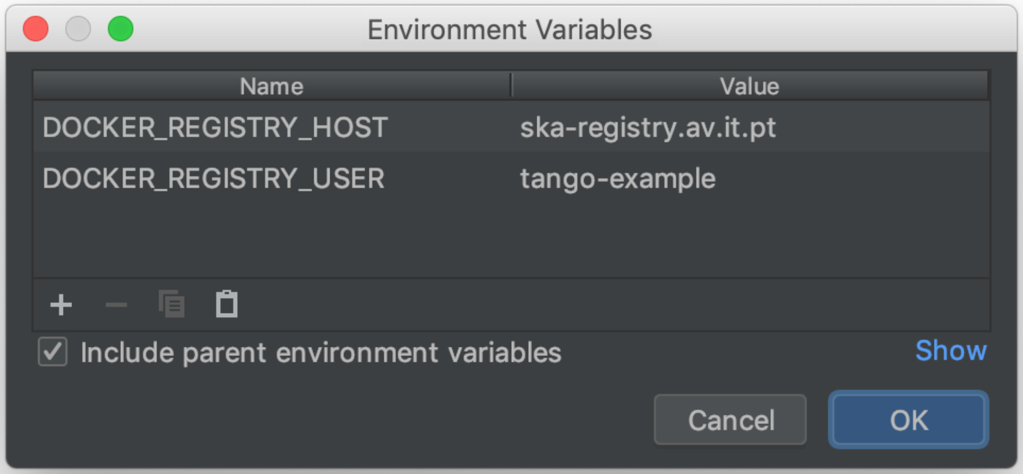
(continues on next page)

(continued from previous page)

```
 ---> Using cache
 ---> Using cache
 ---> Using cache
 ---> Using cache
 ---> 1d24e7c0f8aa
Step 3/4 : RUN ipython profile create
 ---> Using cache
 ---> 93c8f22c5f87
Step 4/4 : CMD ["/venv/bin/python", "/app/powersupply/powersupply.py"]
 ---> Using cache
 ---> b54df79f52d6
[Warning] One or more build-args [DOCKER_REGISTRY_HOST DOCKER_REGISTRY_USER] were not␣
↪consumed
Successfully built b54df79f52d6
Successfully tagged nexus.engageska-portugal.pt/tango-example/powersupply:0.1.0-
↪067e5b3-dirty
docker tag nexus.engageska-portugal.pt/tango-example/powersupply:0.1.0-067e5b3-dirty␣
↪nexus.engageska-portugal.pt/tango-example/powersupply:latest
DOCKER_REGISTRY_HOST=nexus.engageska-portugal.pt DOCKER_REGISTRY_USER=tango-example␣
↪docker-compose up -d
tango-example_tangodb_1 is up-to-date
tango-example_databaseds_1 is up-to-date
tango-example_powersupply_1 is up-to-date
docker run --rm -it --name=powersupply-dev -e TANGO_HOST=databaseds:10000 --
↪network=tango-example_default \
      -v /Users/stewartwilliams/PycharmProjects/tango-example:/app nexus.engageska-
↪portugal.pt/tango-example/powersupply:latest /bin/bash
tango@0f360f86d436:/app$ tango_admin --add-server PowerSupply/test PowerSupply test/
↪power_supply/1
```

Configure the remote interpreter

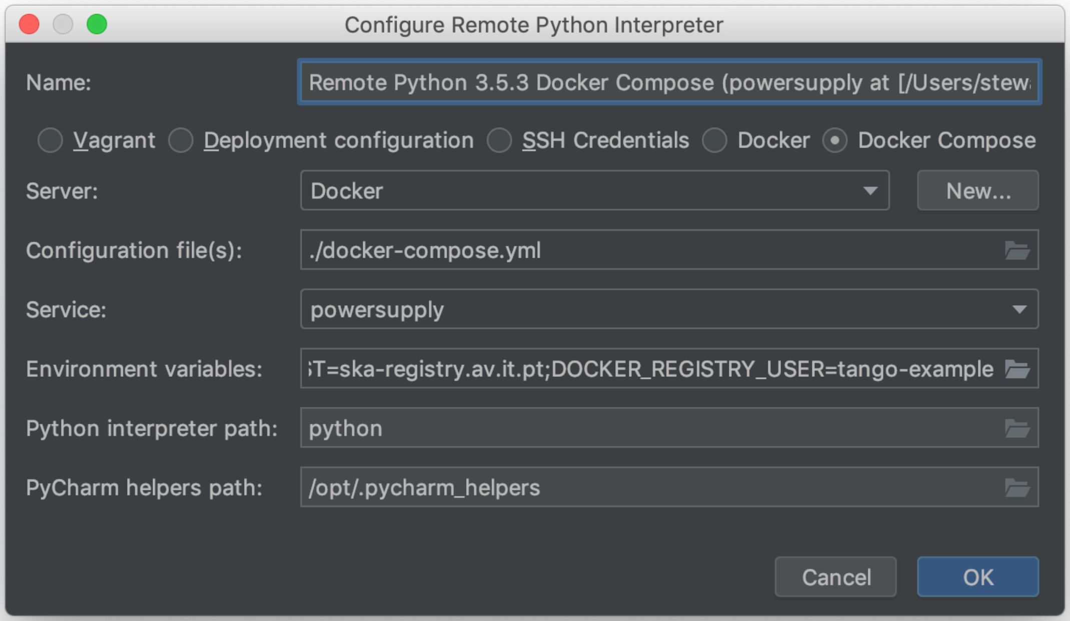
Following the official PyCharm documentation, configure Docker Compose as a remote interpreter. Use the *docker-compose.yml* file found in the root of the tango-example project, and set the service to *powersupply*. The docker-compose.yml file expects the `DOCKER_REGISTRY_HOST` and `DOCKER_REGISTRY_USER` arguments to be provided. Normally these would be provided by the Makefile, but as we are running outside Make then these variables need to be defined. Set the environment variables to look like this:


Environment Variables

Name	Value
DOCKER_REGISTRY_HOST	ska-registry.av.it.pt
DOCKER_REGISTRY_USER	tango-example

✓ Include parent environment variables

The final Configure Remote Python Interpreter dialog should look like this:


Configure Remote Python Interpreter

Name: Remote Python 3.5.3 Docker Compose (powersupply at [/Users/stew

Vagrant Deployment configuration SSH Credentials Docker [✓ Docker Compose]

Server: Docker

Configuration file(s): ./docker-compose.yml

Service: powersupply

Environment variables: T=ska-registry.av.it.pt;DOCKER_REGISTRY_USER=tango-example

Python interpreter path: python

PyCharm helpers path: /opt/.pycharm_helpers

Click 'OK' to apply your changes.

You can now navigate through the project. As an exercise, open the source code for the PowerSupply class, which is defined in powersupply/powersupply.py. Like the Docker configuration, notice that the IDE notifications and intellisense / code completion are now based on information gathered from the remote Docker image.

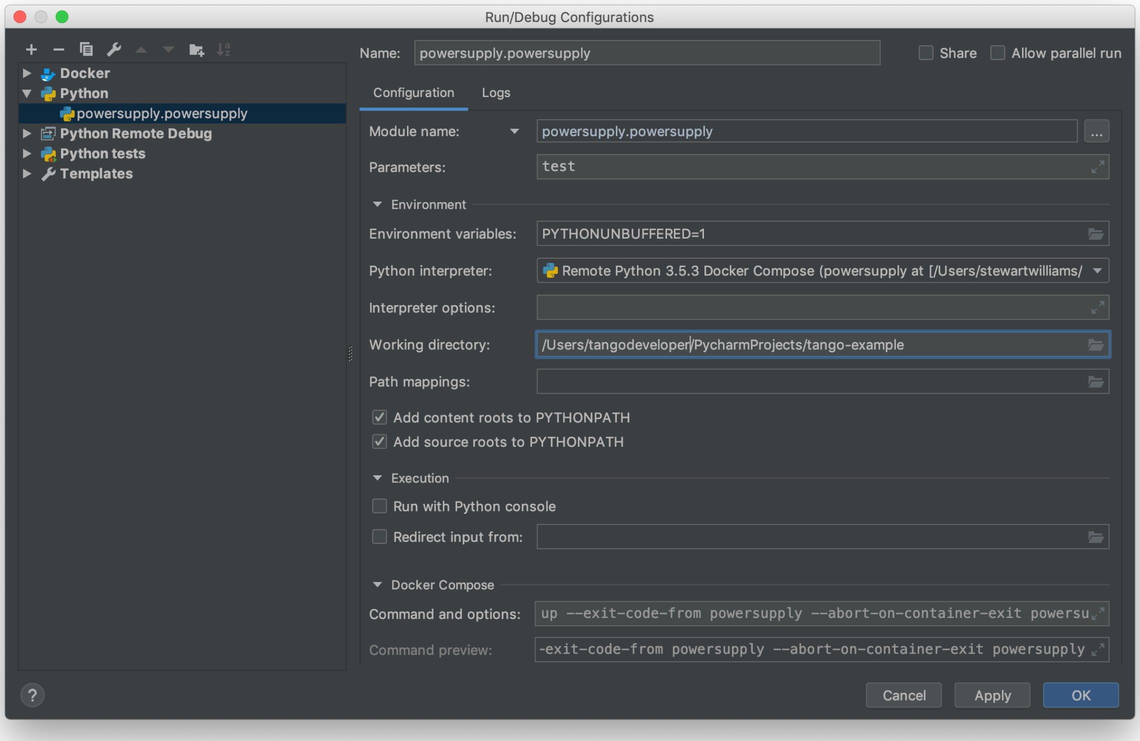
Just as for the Docker configuration, whenever you change the Python environment you should regenerate the project skeletons to make PyCharm aware of the changes. To do this, select File | Invalidate Caches / Restart… from the main menu.

Running the device

From the main menu, choose Run | Edit Configurations… and click on the '+' button to add a new configuration. From the menu that appears, select Python to add a new Python execution configuration. In the dialog, perform these steps:

1. Edit the script/module to be executed to point to the `powersupply.powersupply` module.
2. Add `test` as an execution parameter; this tells the PowerSupply device to execute as the PowerSupply/test instance we registered earlier.
3. Change the working directory to the root of the project.

The final Run/Debug dialog should look like this:



Name: powersupply.powersupply

Module name: powersupply.powersupply

Parameters: test

Environment variables: PYTHONUNBUFFERED=1

Python interpreter: Remote Python 3.5.3 Docker Compose (powersupply at [/Users/stewartwilliams/

Interpreter options:

Working directory: /Users/tangodeveloper/PycharmProjects/tango-example

Path mappings:

☑ Add content roots to PYTHONPATH

☑ Add source roots to PYTHONPATH

☐ Run with Python console

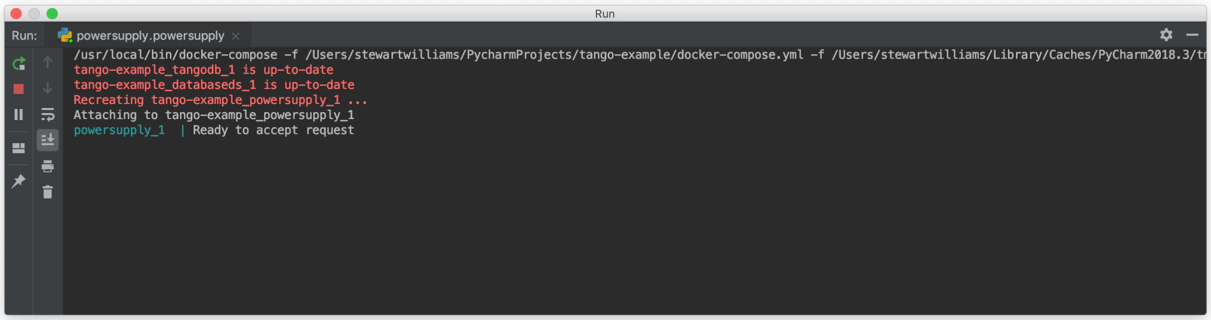
☐ Redirect input from:

Command and options: `up --exit-code-from powersupply --abort-on-container-exit powersu`

Command preview: `-exit-code-from powersupply --abort-on-container-exit powersupply`

Press OK to apply your changes.

From the main menu, choose Run | Run… and select the configuration that you just created in the Run dialog that opens. The PowerSupply device will launch alongside the partner containers defined in docker-compose.yml. PyCharm's Run panel will display output like this, showing the device is executing and ready to accept requests.



```
/usr/local/bin/docker-compose -f /Users/stewartwilliams/PycharmProjects/tango-example/docker-compose.yml -f /Users/stewartwilliams/Library/Caches/PyCharm2018.3/tr
tango-example_tangodb_1 is up-to-date
tango-example_databaseds_1 is up-to-date
Recreating tango-example_powersupply_1 ...
Attaching to tango-example_powersupply_1
powersupply_1  | Ready to accept request
```

Debugging configuration and limitations

The Run configuration also functions as a Debug configuration. Debugging using the docker-compose configuration behaves identically and is subject to the same limitations as debugging using the Docker configuration. If you are familiar with these limitations then free to skip ahead to the next section.

Note: The *coverage* module is not compatible with the PyCharm or Visual Studio Code debugger and must be disabled before any debugging session. Do so by editing *setup.cfg*, commenting out the `addopts=...` line of the tool:pytest section so that it looks like this:

```
[tool:pytest]
testpaths = tests
#addopts = --cov=powersupply --json-report --json-report-file=htmlcov/report.
↪json --cov-report term --cov-report html --cov-report xml --pylint --
↪pylint-error-types=EF
```

PyCharm has a *debug* mode that allows breakpoints to be added to code and the runtime state of the device examined. Refer to the official PyCharm documentation for comprehensive documentation on how to add breakpoints and run in debug mode.

The steps in the official documentation can also be used to debug and interact with ah Tango device, using the configuration set up in the previous section as the basis for the debug configuration. However, full breakpoint functionality requires some workarounds. Breakpoints set outside device initialisation code (i.e., outside `__init__()` and *init_device()*) only function if the Tango device uses asyncio green mode. In non-asyncio modes, Tango creates new Python threads to service requests. Unfortunately these threads do not inherit the debugging configuration attached by PyCharm.

For working breakpoints, there are two solutions:

1. the device must be converted to use asyncio green mode;
2. add `pydevd` to your Piplock as a project dependency, rebuild the Docker image and refresh the project skeletons, then add `pydevd.settrace()` statements where the breakpoint is required. For example, to add a breakpoint in the `PowerSupply.get_current()` method, the code should look like this:

```
def get_current(self):
    """Get the current"""
    import pydevd
    pydevd.settrace()  # this is equivalent to setting a breakpoint in IDE
    return self.__current
```

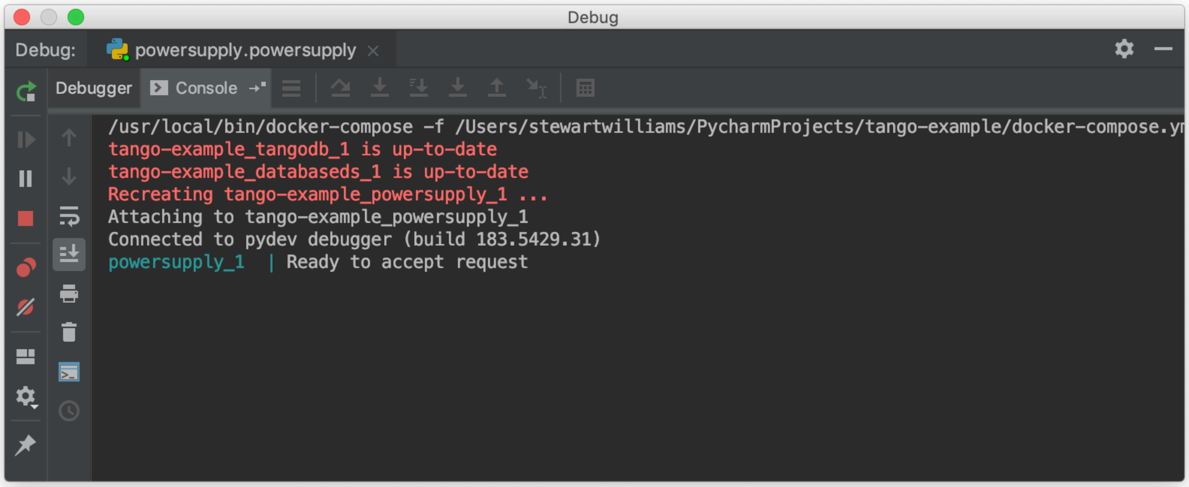
Debugging unit tests

To debug a unit test, we want the unit tests to run in one container while the PyCharm debugger runs and is attached to the PowerSupply device in another container. The easiest way to accomplish this is to launch the device using the debug configuration while the tests we are examining are executed from an interactive session.

First, launch an interactive session with *make interactive*. Keep this session open as we will return to it later.

Note: launching *make interactive* refreshes and recreates the containers defined in docker-compose.yml. Any devices launched by PyCharm will be stopped, requiring the device to be started again in PyCharm once the interactive session is up and running. In short, if you use 'make interactive' while devices are running, expect to have to restart your devices in PyCharm.

From the main menu, choose Run | Debug. . . and select the PowerSupply run configuration you created earlier. The device will be launched and the PyCharm debugger attached to the session. The Debug panel of PyCharm should look similar to this:



```
/usr/local/bin/docker-compose -f /Users/stewartwilliams/PycharmProjects/tango-example/docker-compose.y
tango-example_tangodb_1 is up-to-date
tango-example_databaseds_1 is up-to-date
Recreating tango-example_powersupply_1 ...
Attaching to tango-example_powersupply_1
Connected to pydev debugger (build 183.5429.31)
powersupply_1  | Ready to accept request
```

Returning to the interactive session, run the unit tests that exercise the live Tango device. For the tango-example project, these tests are found in the file *test_2_test_server_using_client.py*.

```
tango@069dde501ca7:/app$ pytest tests/test_2_test_server_using_client.py
============================== test session starts ===============================
platform linux -- Python 3.5.3, pytest-4.2.0, py-1.7.0, pluggy-0.8.1
rootdir: /app, inifile: setup.cfg
plugins: pylint-0.14.0, metadata-1.8.0, json-report-1.0.2, cov-2.6.1
collected 5 items

tests/test_2_test_server_using_client.py .....                               [100%]

============================ 5 passed in 0.18 seconds ============================
```

Set a breakpoint in the PowerSupply.turn_on() method and a single unit test that exercises this function.

```
tango@069dde501ca7:/app$ pytest tests/test_2_test_server_using_client.py -k test_turn_
↪on
============================== test session starts ===============================
platform linux -- Python 3.5.3, pytest-4.2.0, py-1.7.0, pluggy-0.8.1
```

(continues on next page)

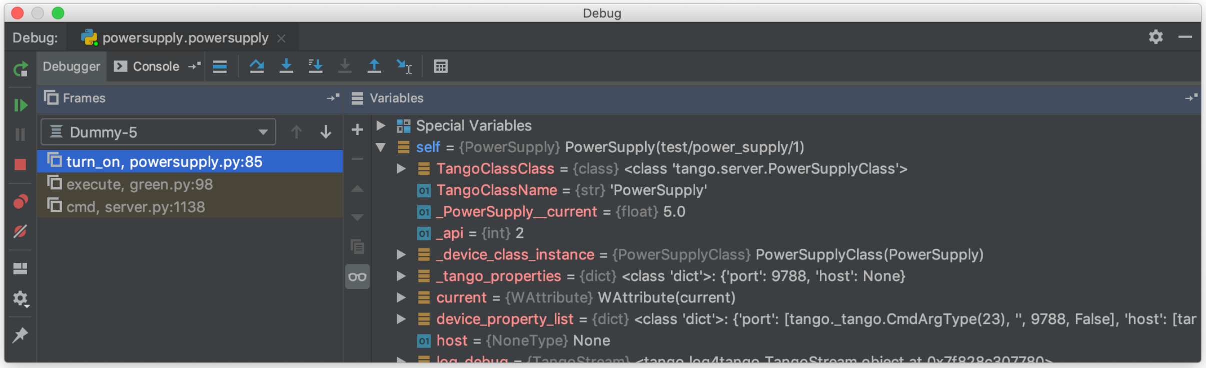
(continued from previous page)

```
rootdir: /app, inifile: setup.cfg
plugins: pylint-0.14.0, metadata-1.8.0, json-report-1.0.2, cov-2.6.1
collected 5 items / 4 deselected / 1 selected

tests/test_2_test_server_using_client.py .                              [100%]

==================== 1 passed, 4 deselected in 0.15 seconds ====================
```

The tests execute but unfortunately the breakpoint is not hit. This is because breakpoints in the main body of the device are not activated (see *Debugging configuration and limitations* for the reasons for this). To work around this, a breakpoint must be introduced into the code itself. Edit the *turn_on* method in *powersupply.py* to look like this:

```
@command
def turn_on(self):
    """Turn the device on"""
    # turn on the actual power supply here
    import pydevd
    pydevd.settrace()
    self.set_state(DevState.ON)
```

Restart the debugging configuration for the code change to take effect and re-execute the test in the interactive session. This time, the breakpoint is respected and execution is frozen, allowing program state to be examined in PyCharm. The debug panel in PyCharm will look something like this, showing that execution is frozen:



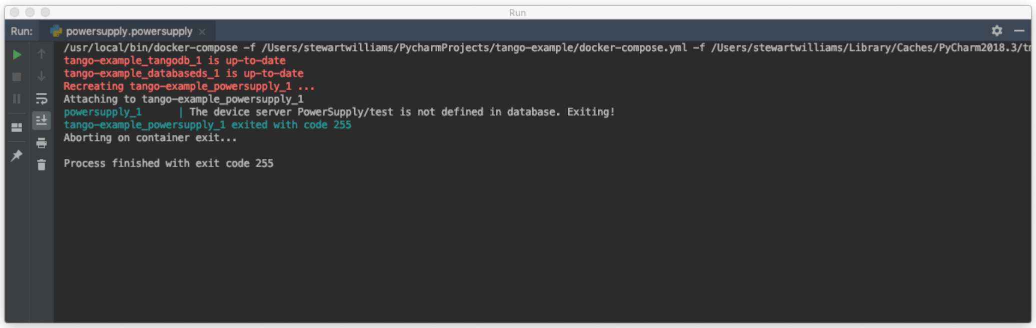
```
turn_on, powersupply.py:85
execute, green.py:98
cmd, server.py:1138
```

```
▶ Special Variables
▼ self = {PowerSupply} PowerSupply(test/power_supply/1)
  ▶ TangoClassClass = {class} <class 'tango.server.PowerSupplyClass'>
    TangoClassName = {str} 'PowerSupply'
    _PowerSupply__current = {float} 5.0
    _api = {int} 2
  ▶ _device_class_instance = {PowerSupplyClass} PowerSupplyClass(PowerSupply)
  ▶ _tango_properties = {dict} <class 'dict'>: {'port': 9788, 'host': None}
  ▶ current = {WAttribute} WAttribute(current)
  ▶ device_property_list = {dict} <class 'dict'>: {'port': [tango._tango.CmdArgType(23), '', 9788, False], 'host': [tar
    host = {NoneType} None
```

Troubleshooting

- **The device server is not defined in the database**

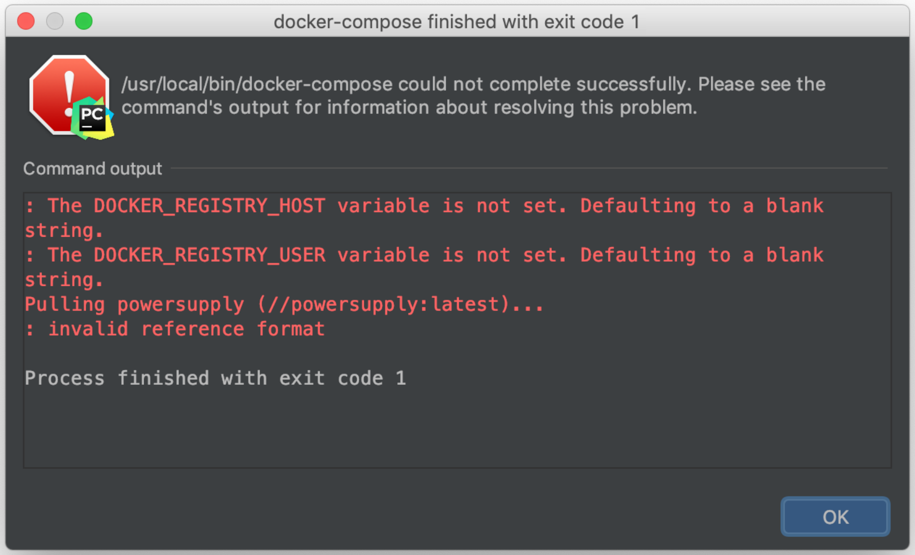
  If you see an error message like the one below, then the device is unregistered and needs to be registered manually. Follow the steps in *Registering the device server*.



```
/usr/local/bin/docker-compose -f /Users/stewartwilliams/PycharmProjects/tango-example/docker-compose.yml -f /Users/stewartwilliams/Library/Caches/PyCharm2018.3/tr
tango-example_tangodb_1 is up-to-date
tango-example_databaseds_1 is up-to-date
Recreating tango-example_powersupply_1 ...
Attaching to tango-example_powersupply_1
powersupply_1      | The device server PowerSupply/test is not defined in database. Exiting!
tango-example_powersupply_1 exited with code 255
Aborting on container exit...

Process finished with exit code 255
```

- **The DOCKER_REGISTRY_HOST variable is not set**

  If you see an error message like the one below, then you forgot to define the environment variables for the remote interpreter. Edit the variables section in your PyCharm docker-compose configuration and try again.



/usr/local/bin/docker-compose could not complete successfully. Please see the command's output for information about resolving this problem.

```
: The DOCKER_REGISTRY_HOST variable is not set. Defaulting to a blank
string.
: The DOCKER_REGISTRY_USER variable is not set. Defaulting to a blank
string.
Pulling powersupply (//powersupply:latest)...
: invalid reference format

Process finished with exit code 1
```

- *PyCharm Professional Docker configuration*
- *PyCharm Professional docker-compose configuration*

Visual Studio Code

Visual Studio Code is a recommended IDE for developing SKA control system software.

Visual Studio Code is an open source project and available for free from the VSCode download page.

Visual Studio Code docker configuration

These instructions show how to configure Visual Studio Code for SKA control system development using the SKA Docker images. VSCode can be configured to debug using the Python interpreter inside a Docker image, which allows:

- development and testing without requiring a local Tango installation;
- the development environment to be identical to the testing and deployment environment, eliminating problems that occur due to differences in execution environment.

Limitations of VSCode docker container debugging compared to PyCharm:

- Unlike PyCharm Pro Edition, VSCode docker integration doesn't allow for code completion and linting using a docker container though. Therefore in order to have intellisense (code completion inside VSCode) and linting you will need to have a local installation of the project as well (i.e. a *pipenv* environment).
- VSCode remote debugging library *ptvsd* presently conflicts with *pytest*, meaning that debugging breakpoints cannot be set while running the automated unit testing. Still, you can set any particular unit test file as the entry point for debugging and set breakpoints normally in it. The developing approach should then be to run the unit tests from the terminal, and then in case of errors, to analyze the specific test routine from within the debugger in VSCode.

Improvements to debugging capabilities in VSCode compared to PyCharm:

- Unlike PyCharm, VSCode does allow for setting up breakpoints on non-asyncio modes.

Follow the steps below to configure VSCode to develop new code and run tests for the tango-example project using the Docker images for the project.

Prerequisites

Make sure that the following prerequisites are met:

- Docker is installed, as described on the page Docker Docs.
- Visual Studio Code must be installed.
- You have basic familiarity with VSCode - If this is the first time you have used VSCode, follow the First Steps tutorials so that you know how to use VSCode to develop, debug, and test a simple Python application using a local Python interpreter.

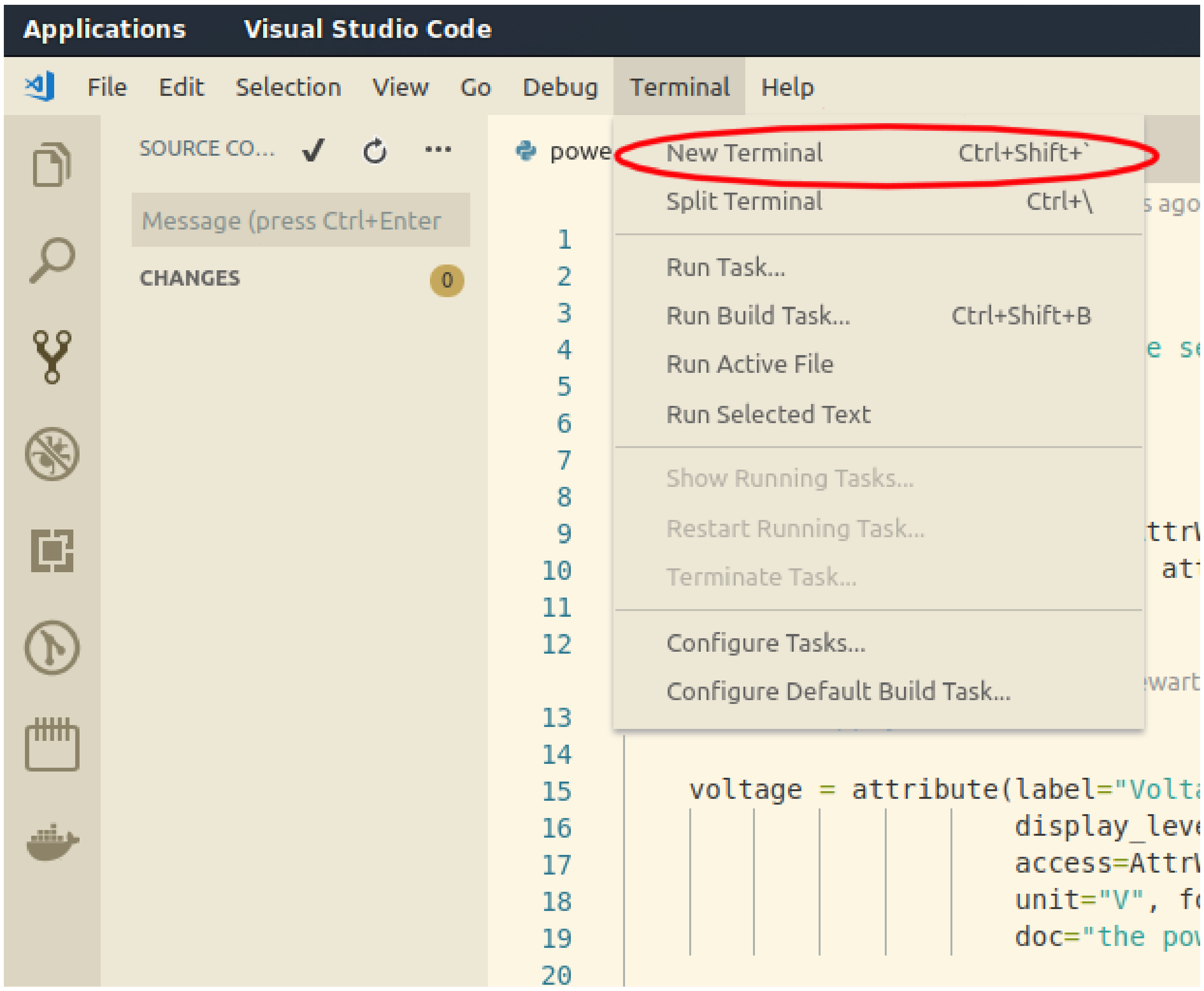
Clone the tango-example project and get VSCode to recognize it

1. Clone the tango-example repository in your local machine.
2. Open VSCode from inside the *tango-example* folder.

Build the application image (this step is optional)

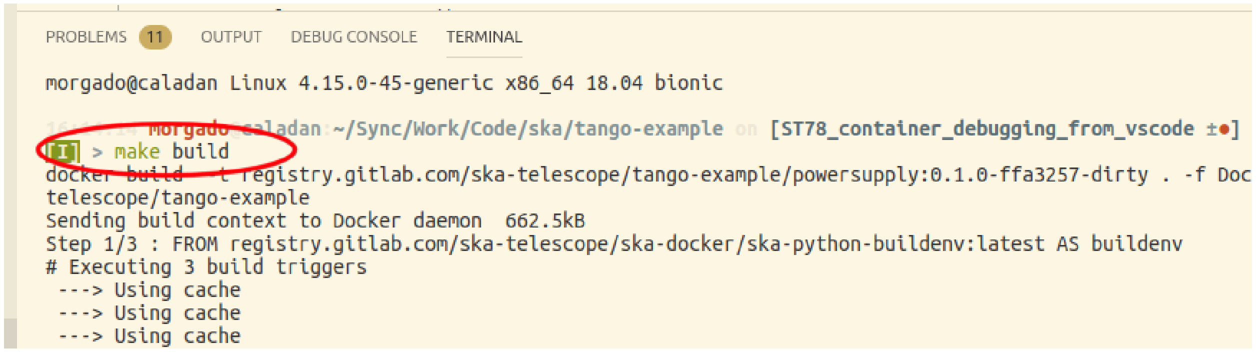
With the source code source code checked out, the next step is to build a Docker image for the application. This image will contain the Python environment which will we will later connect to VSCode.

Start a terminal session inside VSCode:



```
    voltage = attribute(label="Volta
                        display_leve
                        access=AttrW
                        unit="V", fo
                        doc="the pow
```

On the terminal tab build the image by typing `make build`. *This step is optional since the ‘‘make interactive‘‘ command described bellow, takes care of this task if needed*:

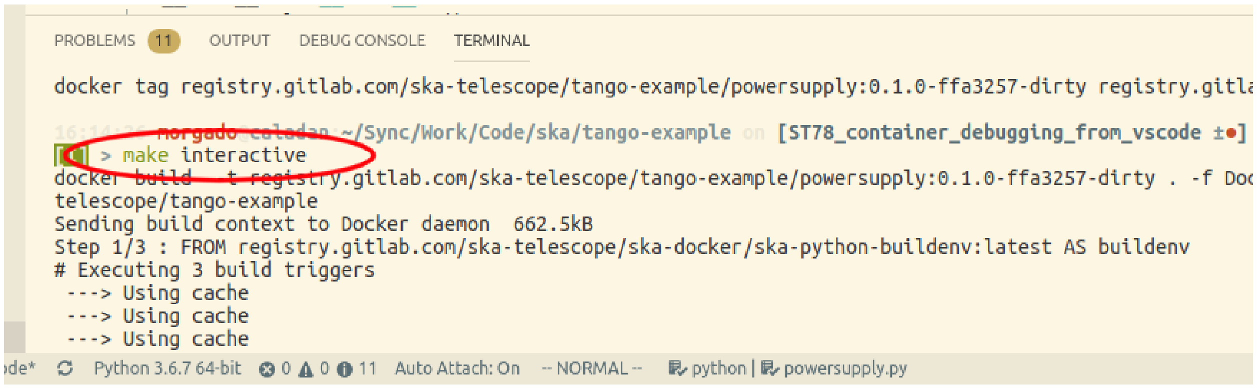


```
morgado@caladan Linux 4.15.0-45-generic x86_64 18.04 bionic

16...  morgado@caladan ~/Sync/Work/Code/ska/tango-example on [ST78_container_debugging_from_vscode ±●]
[I] > make build
docker build -t registry.gitlab.com/ska-telescope/tango-example/powersupply:0.1.0-ffa3257-dirty . -f Doc
telescope/tango-example
Sending build context to Docker daemon  662.5kB
Step 1/3 : FROM registry.gitlab.com/ska-telescope/ska-docker/ska-python-buildenv:latest AS buildenv
# Executing 3 build triggers
 ---> Using cache
 ---> Using cache
 ---> Using cache
```

Start the docker container in interactive mode and debug

Having the built docker image in the system we now start the docker container in interactive mode and are ready to debug.

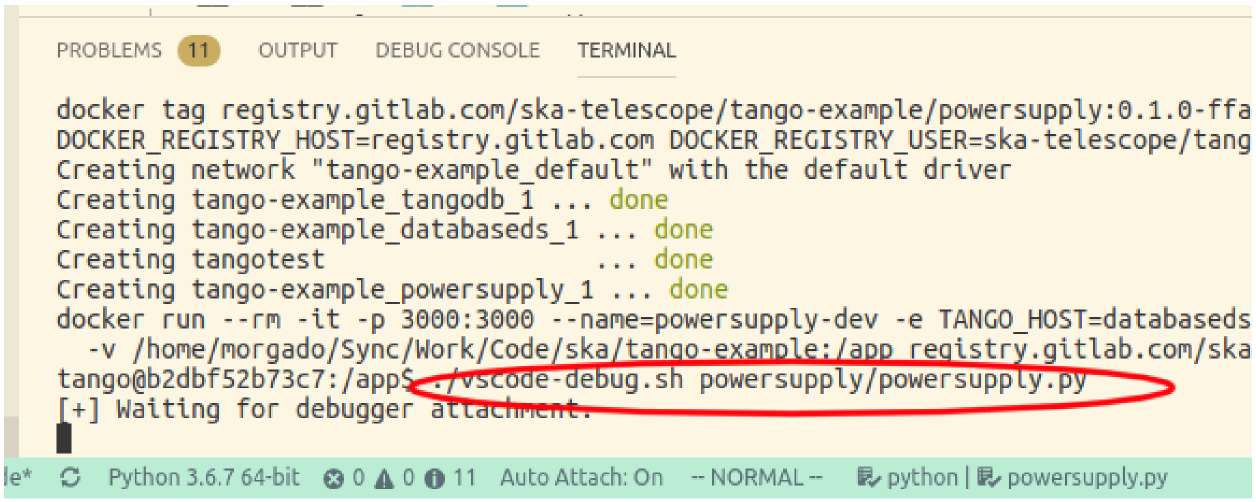
- On the terminal tab start the container interactively with `make interactive`:



```
docker tag registry.gitlab.com/ska-telescope/tango-example/powersupply:0.1.0-ffa3257-dirty registry.gitla

16:14 morgado-caladan ~/Sync/Work/Code/ska/tango-example on [ST78_container_debugging_from_vscode ±●]
> make interactive
docker build -t registry.gitlab.com/ska-telescope/tango-example/powersupply:0.1.0-ffa3257-dirty . -f Doc
telescope/tango-example
Sending build context to Docker daemon  662.5kB
Step 1/3 : FROM registry.gitlab.com/ska-telescope/ska-docker/ska-python-buildenv:latest AS buildenv
# Executing 3 build triggers
 ---> Using cache
 ---> Using cache
 ---> Using cache
```

ode* Python 3.6.7 64-bit 0 0 11 Auto Attach: On -- NORMAL -- python | powersupply.py

- Debug a particular file using the `vscode-debug.sh` utility inside the docker image. For instance `./vscode-debug.sh powersupply/powersupply.py`:



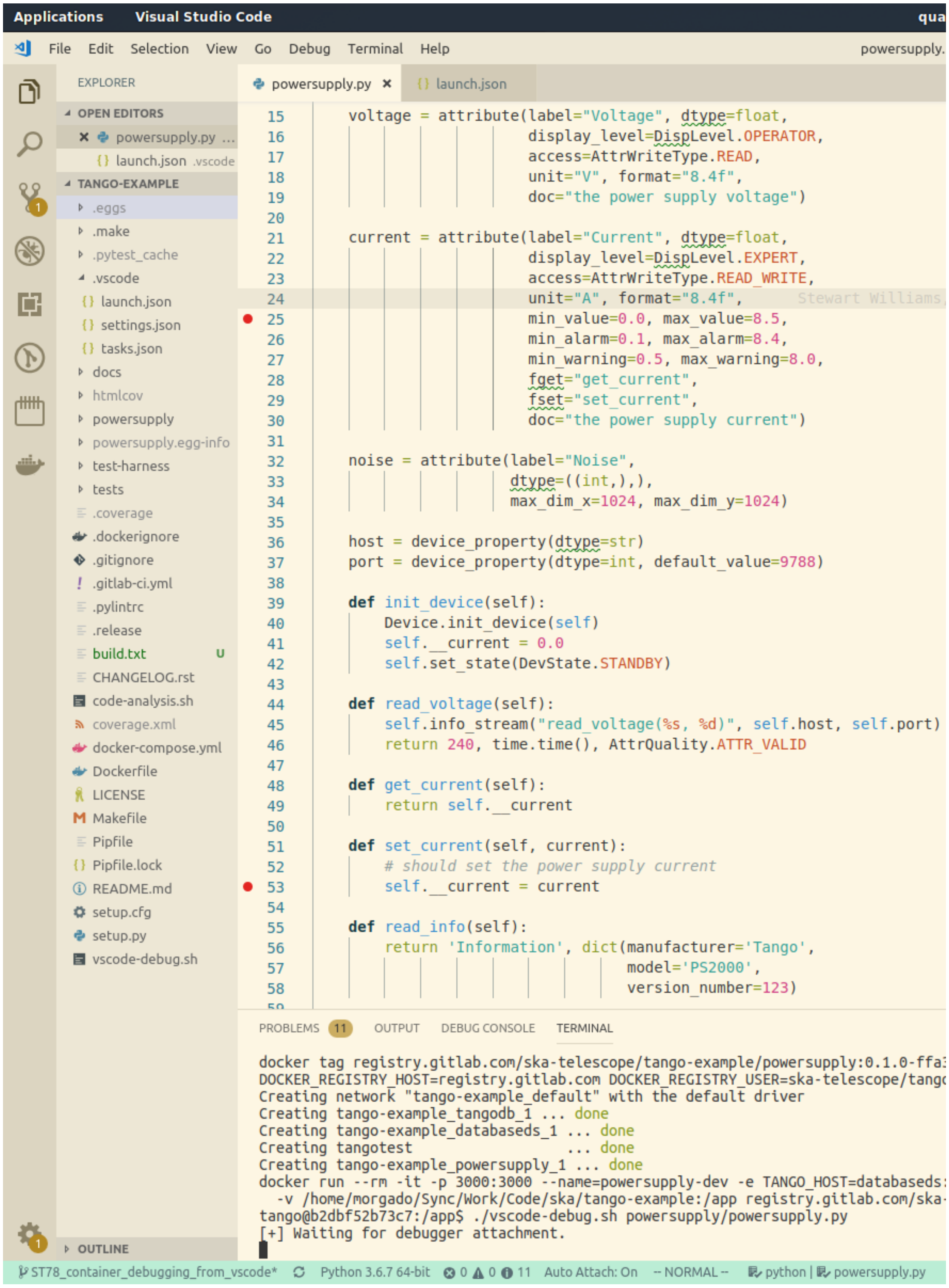
```
docker tag registry.gitlab.com/ska-telescope/tango-example/powersupply:0.1.0-ffa
DOCKER_REGISTRY_HOST=registry.gitlab.com DOCKER_REGISTRY_USER=ska-telescope/tang
Creating network "tango-example_default" with the default driver
Creating tango-example_tangodb_1 ... done
Creating tango-example_databaseds_1 ... done
Creating tangotest                   ... done
Creating tango-example_powersupply_1 ... done
docker run --rm -it -p 3000:3000 --name=powersupply-dev -e TANGO_HOST=databaseds
  -v /home/morgado/Sync/Work/Code/ska/tango-example:/app registry.gitlab.com/ska
tango@b2dbf52b73c7:/app$ ./vscode-debug.sh powersupply/powersupply.py
[+] Waiting for debugger attachment.
```

le* Python 3.6.7 64-bit 0 0 11 Auto Attach: On -- NORMAL -- python | powersupply.py

Notice that the terminal shell now shows a message stating that it is waiting for the debugger attachment:

```
tango@b2dbf52b73c7:/app$ ./vscode-debug.sh powersupply/powersupply.py
[+] Waiting for debugger attachment.
```

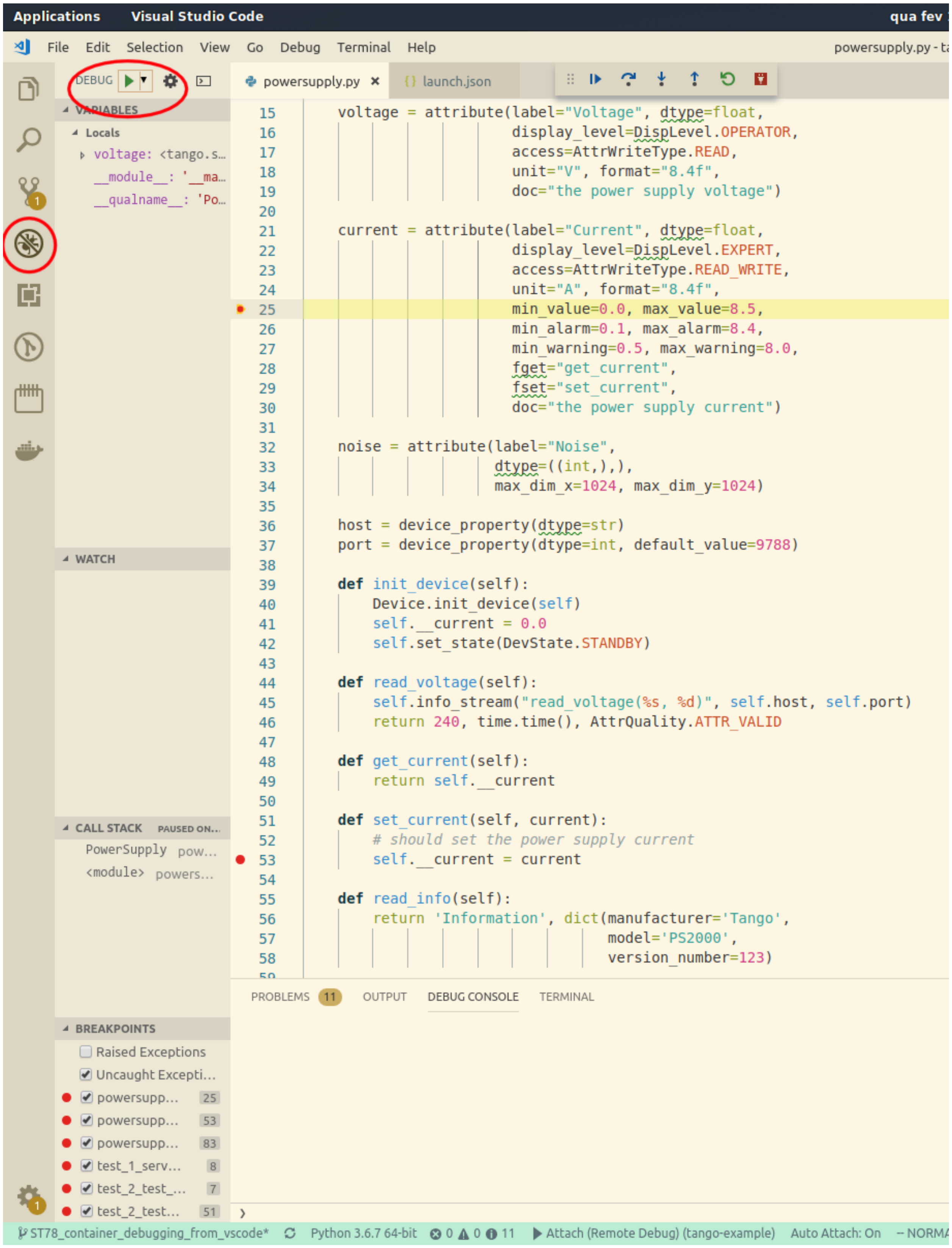
- You can now set breakpoints inside the VSCode editor (or use previously set ones):



```
    voltage = attribute(label="Voltage", dtype=float,
                        display_level=DispLevel.OPERATOR,
                        access=AttrWriteType.READ,
                        unit="V", format="8.4f",
                        doc="the power supply voltage")

    current = attribute(label="Current", dtype=float,
                        display_level=DispLevel.EXPERT,
                        access=AttrWriteType.READ_WRITE,
                        unit="A", format="8.4f",
                        min_value=0.0, max_value=8.5,
                        min_alarm=0.1, max_alarm=8.4,
                        min_warning=0.5, max_warning=8.0,
                        fget="get_current",
                        fset="set_current",
                        doc="the power supply current")

    noise = attribute(label="Noise",
                      dtype=((int,),),
                      max_dim_x=1024, max_dim_y=1024)

    host = device_property(dtype=str)
    port = device_property(dtype=int, default_value=9788)

    def init_device(self):
        Device.init_device(self)
        self.__current = 0.0
        self.set_state(DevState.STANDBY)

    def read_voltage(self):
        self.info_stream("read_voltage(%s, %d)", self.host, self.port)
        return 240, time.time(), AttrQuality.ATTR_VALID

    def get_current(self):
        return self.__current

    def set_current(self, current):
        # should set the power supply current
        self.__current = current

    def read_info(self):
        return 'Information', dict(manufacturer='Tango',
                                   model='PS2000',
                                   version_number=123)
```

```
docker tag registry.gitlab.com/ska-telescope/tango-example/powersupply:0.1.0-ffa
DOCKER_REGISTRY_HOST=registry.gitlab.com DOCKER_REGISTRY_USER=ska-telescope/tang
Creating network "tango-example_default" with the default driver
Creating tango-example_tangodb_1 ... done
Creating tango-example_databaseds_1 ... done
Creating tangotest                  ... done
Creating tango-example_powersupply_1 ... done
docker run --rm -it -p 3000:3000 --name=powersupply-dev -e TANGO_HOST=databaseds
  -v /home/morgado/Sync/Work/Code/ska/tango-example:/app registry.gitlab.com/ska-
tango@b2dbf52b73c7:/app$ ./vscode-debug.sh powersupply/powersupply.py
[+] Waiting for debugger attachment.
```

- Start the debugger from whitin VSCode by pressing `F5` or the *debug* button under the debug tab:



```
    voltage = attribute(label="Voltage", dtype=float,
                        display_level=DispLevel.OPERATOR,
                        access=AttrWriteType.READ,
                        unit="V", format="8.4f",
                        doc="the power supply voltage")

    current = attribute(label="Current", dtype=float,
                        display_level=DispLevel.EXPERT,
                        access=AttrWriteType.READ_WRITE,
                        unit="A", format="8.4f",
                        min_value=0.0, max_value=8.5,
                        min_alarm=0.1, max_alarm=8.4,
                        min_warning=0.5, max_warning=8.0,
                        fget="get_current",
                        fset="set_current",
                        doc="the power supply current")

    noise = attribute(label="Noise",
                      dtype=((int,),),
                      max_dim_x=1024, max_dim_y=1024)

    host = device_property(dtype=str)
    port = device_property(dtype=int, default_value=9788)

    def init_device(self):
        Device.init_device(self)
        self.__current = 0.0
        self.set_state(DevState.STANDBY)

    def read_voltage(self):
        self.info_stream("read_voltage(%s, %d)", self.host, self.port)
        return 240, time.time(), AttrQuality.ATTR_VALID

    def get_current(self):
        return self.__current

    def set_current(self, current):
        # should set the power supply current
        self.__current = current

    def read_info(self):
        return 'Information', dict(manufacturer='Tango',
                                   model='PS2000',
                                   version_number=123)
```

Note: For general information on how to use the native VSCode debugger, consult the Debugging documentation from VSCode.

Troubleshooting

- **make interactive fails**

  If the debugger is disconnected improperly, there is a possibility that the docker containers are left running in the background and it isn't possible to start a new interactive sessions from the VSCode terminal:

  ```
  docker run --rm -it -p 3000:3000 --name=powersupply-dev -e TANGO_
  ↪HOST=databaseds:10000 --network=tango-example_default \
    -v /home/morgado/Sync/Work/Code/ska/tango-example:/app registry.gitlab.com/ska-
  ↪telescope/tango-example/powersupply:latest /bin/bash
  docker: Error response from daemon: Conflict. The container name "/powersupply-dev
  ↪" is already in use by container
  ↪"215a9150910605a0670058a0023cbd2d180f1cea11d196b2a413910fb428e290". You have to␣
  ↪remove (or rename) that container to be able to reuse that name.
  See 'docker run --help'.
  Makefile:59: recipe for target 'interactive' failed
  make: *** [interactive] Error 125
  ```

  In this case you need to check what are the docker containers running using `docker ps`, and then kill the containers that are running in the background with `docker kill CONTAINER_NAME`.

- *Visual Studio Code docker configuration*
- *Onboarding: Welcome to the SKA developer community*
- *Contributing to the SKA*
- *Configuring your development environment*

A list of the tools we are using to collaborate, together with guidance on how to use them can be found at this confluence page: SKA Guidelines to Remote Working (requires an SKA Confluence account).

CHAPTER 2

SKA Repositories

2.1 Projects by Area

Topics: skampi, infrastructure

Documentation	Gitlab repository
testdoc	testgit
testdoc	testgit

2.1.1 SDP

Science Data Processor

The Science Data Processor (SDP) is part of the evolutionary prototype of the SKA. The SDP is built from software modules which produce a number of different types of artefacts. The components of the system are built as Docker images which are deployed on a Kubernetes cluster using a Helm chart. The Docker images depend on libraries containing common code.

The components of the system are integrated in the SDP integration repository, which contains the Helm chart to deploy the SDP. More details on the design of the SDP and how to run it locally or in the integration environment can be found in the documentation.

Fig. 1: SDP module view

The software modules are organised according to the SDP module view, shown above. Those developed so far are part of:

Execution Control

Local Monitoring and Control

The local monitoring and control is the interface between the telescope control system and the SDP. It consists of Tango device servers which control different aspects of the system. The SDP Master device provides the top-level control of the system. The SDP Subarray device controls the processing associated with a telescope subarray.

- Repository
- Documentation

Processing Controller

The processing controller controls the execution of processing blocks. The processing blocks define the workflows to be run and the parameters to be passed to the workflows. It detects processing blocks by monitoring the configuration database.

- Repository
- Documentation

Configuration Library

The configuration library is the interface to the configuration database. The database is the central store of configuration data in the SDP and it is used to coordinate actions between the components of the system. The configuration library defines the schema for the entries in the database and it provides ways for controller and processing components to discover and manipulate the intended state of the system.

- Repository
- Documentation

Console

The console provides an environment for interacting with the configuration database.

- Repository

Science Pipeline Workflows

The science pipeline workflows define the processing to be done by SDP to generate the data products. They express the high-level organisation of the processing, which is carried out by execution engines and processing functions. Details of the realtime and and batch workflows, and instructions on how to develop a workflow can be found here.

- Repository
- Documentation

Workflow Library

The workflow library is a high-level interface for writing workflows. Its goal is to provide abstractions to enable the developer to define a workflow without needing to interact directly with the low-level interfaces such as the configuration library.

- Repository

- Documentation

Platform Services

Helm Deployer

The Helm deployer is a prototype of a platform controller for the SDP. It dynamically allocates processing and buffer resources on a Kubernetes cluster.

- Repository
- Documentation

Helm Deployer Charts

The charts used by the Helm deployer are maintained in this repository.

- Repository

2.1.2 Simulations

Simulations

RASCIL

The Radio Astronomy Simulation, Calibration and Imaging Library expresses radio interferometry calibration and imaging algorithms in Python and Numpy.

RASCIL

- Repository
- Documentation

RASCIL Docker images

- Repository

RASCIL examples

- Repository

OSKAR

OSKAR is designed to simulate visibility data from radio telescopes containing aperture arrays, such as SKA-Low.

OSKAR

- Repository
- Documentation

OSKAR Python interface

- Documentation

SKA-Low Simulations

This repository contains scripts for generating SKA-Low simulations.

- Repository
- Documentation

SKA-Mid Simulations

This repository contains scripts for generating SKA-Mid simulations.

- Repository
- Documentation

2.2 Alpahbetical list of projects and subgroups

The majority of the SKA projects are currently housed at the root of our gitlab organisation repository at [https://gitlab.com/ska-telescope] but the SKA encourages the use of Gitlab subgroups to house closely related repositories.

2.2.1 List of subgroups

The following subgroup list is automatically extracted from our gitlab organisation repository at [https://gitlab.com/ska-telescope]

Subgroups: Science Data Challanges, Software Defined Infrastructure, templates-

2.2.2 List of non-grouped projects

The following table is automatically extracted from our gitlab organisation repository at [https://gitlab.com/ska-telescope] and it contains the full list of projects at the root of ska-telescope.

Documentation	Gitlab repository
testdoc	testgit
testdoc	testgit

2.3 Create a new project

The SKA code repositories are all stored on the SKA Gitlab account, on gitlab.com/ska-telescope. The SKA's repositories on Gitlab have to be created by a member of the Systems team. If you need a repository simply go to the Slack channel #team-system-support and ask for a new repository to be created. Choose the name well (see below). Respositories will be created with public access by default. Other permissions schemes, such as private and IP protected repositories, are also possible upon request.

You will be given Maintainer privileges on this project. This will make it possible for you to (among other things) add users to the project and edit their permissions. For more information about permissions on Gitlab, go to https://docs.gitlab.com/ee/user/permissions.html.

In early 2020 the creation of repositories by developers or team members on the SKA Gitlab instance may be supported.

On groups in Gitlab

The SKA Telescope group on Gitlab has two sub-groups, *SKA Developers* and *SKA Reporters*.

Groups on Gitlab are like directory structures which inherit permissions, and users can be added to these groups with certain permissions. All users in the SKA are added to the main group as Guest users.

There are some repositories which are IP protected, that may not belong to one or either of the two subgroups. Users that need access to these repositories must be added individually - please ask the System team for assistance.

When creating a new repo there is a number of aspects to be considered.

2.3.1 Mono VS Multi repositories

One of the first choices when creating a new project is how to split the code into repositories. In a project such as SKA there is no strict rule that can be applied, and a degree of judgement is necessary from the developers. Too many repositories create an integration problem and are subject to difficult dependency management, whereas too few repositories make it more difficult to concurrently develop and release independent features. Both scenarios can be approached with the aid of right tooling and processes, but in general a repository should be created for every software component following this definition:

> *All software and firmware source code handed over to the SKA organisation shall be organised into source code repositories. A source code repository is a set of files and metadata, organized in a directory structure. It is expected that source code repositories map to individual applications or modules according to the following definition: A module is reusable, replaceable with something else that implements the same API, independently deployable, and encapsulates some coherent set of behaviors and responsibilities of the system.*
>
> —*adapted from 'Continuous Delivery'*

2.3.2 Naming a repository

Repository names shall clearly map to a particular element of the SKA software architecture, as described in the SKA software design documentation. That is to say, someone familiar with the SKA software architecture shuold be able to identify the content of a repository just by its name.

Names shall be all lowercase, multiple words shall be separated by hyphens.

2.3.3 Repository contents

All software repositories shall host whatever is necessary to download and run the code they contain. This does not only include code, but also documentation, dependencies and configuration data. Can someone external to the project point at your repository and have all the means to run your code? A software repository shall contain:

Repository Checklist

- A *LICENSE* file (see *Licensing a project*)
- A README.md file containing basic instructions on what the repository contains. And how to install, test and run the software
- A *docs* folder containing RST formatted documentation (see *Documenting a project*)
- All application code and dependencies (e.g. libraries, static content etc…)

- Any script used to create database schemas, application reference data etc. . .
- All the environment creation tools and artifacts described in the previous step (e.g. Puppet or Ansible playbooks)
- Any file used to create containers (Dockerfile, docker-compose.yml . . .)
- All supported automated tests and any manual test scripts
- Any script that supports code packaging, deployment, database migration and environment provisioning
- All project artefacts (deployment procedures, release notes etc..)
- All cloud configuration files
- Any other script or configuration information required to create infrastructure that supports multiple services (e.g. enterprise service buses, database management systems, DNS zoe files, configuration rules for firewalls, and other networking devices)

– *adapted from 'The DevOps Handbook'*

2.3.4 SKA Skeleton Projects

The SKA Organisation repository contain a number of skeleton projects which are intended to be forked when starting a new project. The skeleton projects contain all necessary hooks to documentation, test harness, continuous integration, already configured according to SKA standards.

2.4 Licensing a project

SKA organisation promotes a model of open and transparent collaboration. In this model collaboration is made possible using permissive licenses, and not by pursuing the ownership of code by the organisation. Copyright will thus belong to the institutions contributing to source code development for the lifetime of the project, and software developed for the SKA telescope will be available to the wider community as source code. Redistribution of the SKA software will always maintain the original copyright information, acknowledging the original software contributor.

2.4.1 License File

Every software repository shall be licensed according to the SPDX standard. Every repository shall contain a LICENSE file in its top directory, indicating the copyright holder and the license used for the software in its full textual representation.

2.4.2 Acceptable License

SKA office will automatically accept the BSD 3-clause new LICENSE and any exception to this shall be justified and agreed with SKA office Software Quality Assurance engineer. A template of the license is presented at the end of this page. Existing repositories already published with permissive licenses such as Apache 2.0 or MIT licenses will also be accepted as part of the handover procedure, while new repositories are encouraged to adopt the recommended BSD license.

2.4.3 Copyright Information

Copyright information shall be included in the license file, clearly stating the year and the institution the copyright applies to, in the form:

```
Copyright <years> <institution>

An example of this for the SKA organisation would be:

Copyright 2018 SKA Organisation
```

A non exhaustive list of possible copyright notices, based on pre construction SKA collaborators, may include one or more of the following:

```
Copyright 2018 AIT Aveiro
Copyright 2018 ASTRON
Copyright 2018 ATC
Copyright 2018 CSIRO
Copyright 2018 ICRAR
Copyright 2018 INAF
Copyright 2018 NCRA
Copyright 2018 SARAO
Copyright 2018 University of Malta
Copyright 2018 University of Manchester
Copyright 2018 University of Oxford
...
```

A single license file can contain multiple copyright notices, indicating the major contributors to the software repositories. It is not in the scope of the copyright notice to maintain an updated list of single contributors which can always be extracted from the DVCS server system in a more reliable and maintainable way. Whenever a license assumes that copyright is explicitly stated as part of the header of every source code file, this can be summarized into a single centralized COPYRIGHT file in the top directory of the repository, containing all copyright attributions and referred to by the single header comments in the source code:

```
Copyright 2018 The Foo Project Developers. See the COPYRIGHT file at the top-level␣
↪directory of this distribution.
```

It will be the duty of single repository administrators to make sure that copyright notices are maintained and updated according to the institution contributing to the project.

2.4.4 BSD 3-Clause "New" or "Revised" License text template

```
Copyright <YEAR> <COPYRIGHT HOLDER>

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its
contributors may be used to endorse or promote products derived from this
software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
```

(continues on next page)

(continued from previous page)

2.5 Working with GitLab

2.5.1 About git

Git is the version control system of choice used by SKA. Describing the basics of how to use Git is out of the scope of this developer portal, but it is fundamental that all developers contributing to SKA get familiar with Git and how to use it. These online resources are a good starting point:

- Learn git interactively: https://learngitbranching.js.org/
- Official git reference at: https://git-scm.com/docs
- Interactive Git cheatsheet: http://www.ndpsoftware.com/git-cheatsheet.html

2.5.2 GitLab as the Git repository manager

The SKA Software team have adopted the GitLab social coding platform as the main Git repository manager for its CI/CD tools.

The following describes how to access the service, and how to setup the basic working environment to integrate with GitLab for the SKA.

Use institutional email

Create a GitLab account using your **institutional email** address at https://gitlab.com/users/sign_in. If you already have an account on GitLab, you should have your institutional email added to your profile: click on your user icon on the top right corner and select *Settings->Emails->Add email address* . It is recommended that 2FA (two-factor authentication) is enabled. There are a variety of OTP (One Time Pin) applications available for mobile phones with instructions available: https://docs.gitlab.com/ee/user/profile/account/two_factor_authentication.html.

Setup SSH key

To enable *git+ssh* based authentication for clients, associate your ssh-key to your user at *Settings->SSH keys* (https://gitlab.com/profile/keys).

SKA Organization

SKA Organization can be found on GitLab at https://gitlab.com/ska-telescope. Send a request to the System Team on Slack (*team-system-support* channel) to link your account to the SKA Gitlab group, and assist with creation of repositories and integrating CI.

Code Snippets

You can share code snippets (code blocks) within the SKA Organization using the *ska-snippets* repository, and also you can always share code snippets with the project members using project level snippets *(If they are enabled)*

2.5.3 Committing code

When working on a development project, it is important to stick to these simple commit rules:

- Work in feature branches where possible (see *Branching policy*)
- Commit early, commit often.
- Have the **Jira story ID** at the beginning of your commit messages. (You can also use Gitlab and JIRA integration defined in *Working with Jira*)
- Git logs shall be human readable in sequence, describing the development activity.
- Use imperative forms in the commit message:

```
ST-320 make fluentd,kibana,elasticsearch optional

* add a pipeline example for disabling ELK
* add enabled checks
* filebeat,journalbeat support added to clusters

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Tue May 12 11:24:17 2020 +1200
#
# On branch st-320-swap-out-fluentd
# Your branch is up-to-date with 'origin/st-320-swap-out-fluentd'.
```

You can find additional information on how to write a good commit message here.

2.5.4 Configure Git

Set Git institutional email address

Setup Git so that it uses your institutional email account to sign commits, this can be done in your global Git configuration:

```
$ git config --global user.email "your@institutional.email"
```

Or you can configure the mail address on a project basis.

```
$ cd your/git/project
$ git config user.email "your@institutional.email"
```

Signing commits with GPG

Developers are strongly encouraged to use a GPG key to sign Git commits. The procedure for obtaining a GPG key and uploading it to the GitLab account is described at https://docs.gitlab.com/ee/user/project/repository/gpg_signed_commits/, or got straight to uploading at https://gitlab.com/profile/gpg_keys.

The only difference from a non signed commit is the addition of the -S flag:

```
$ git commit -S -m "My commit msg"
```

The passphrase of your GPG key will then be asked. To avoid having to type the -S flag every time a commit is made, Git can be configured to sign commits automatically:

```
$ git config --global commit.gpgsign true
```

When working in a remote repository by ssh connection, you need to create new GPG key and add it as explained above or you can forward your existing gpg key to the remote machine following the instructions below. You can find more information here.

- Find your local socket: `gpgconf --list-dir agent-extra-socket`
- Find your remote socket: `gpgconf --list-dir agent-socket`
- Configure your SSH configuration file by adding the following line after your host settings: `RemoteForward <socket_on_remote_box> <extra_socket_on_local_box>`. Note that you need to reconnect to the remote machine to apply the changes.
- Add `StreamLocalBindUnlink yes` into `/etc/ssh/sshd_config` in the remote machine and restart the sshd service to close the gpg forwarding socket when closing the ssh connection.

Squashing commits

If you want to replace a series of small commits with a single commit or if you want to make their order more logical you can use an interactive rebase (git rebase -i) to squash multiple commits into one or reorder them. When squashing commits it is important to consider the following:

- You should never rebase commits you have pushed to a remote server.
- You should also never rebase commits authored by other people.

In general the squashing of commits is discouraged for SKA repositories.

2.5.5 Branching policy

The preference within the SKA is that a feature branch workflow be adopted, however it is recognised that there are multiple workflows in use, and team requirements and composition vary. Regardless of the adopted workflow employed, two concepts are important to the SKA way of using Git:

1. The master branch of a repository shall always be stable, and tested.
2. 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 the precedence over any other activity on the repository.

The following sections discuss the two of the most common workflows:

- Master or trunk based development
- Feature based branching

Master based development

Teams may adopt a particular Git workflow designated as Master based development approach, where each developer commits code into the master branch on a daily basis. While this practice may seem counter intuitive, there is good evidence in literature that it leads to a better performing system. Branches are reduced to a minimum in this model, and the discipline of daily commits into master greatly enhances the communication within the team and the modularity of the software system under construction. The workflow follows these steps:

- As a developer starts working on a story, all their commits related to the story shall contain the story Jira ID in the message. i.e. *AT-51 method stubs*
- The developer continues working on their local master branch with multiple commits on the same story.
- Each day the local master pulls the remote and incorporates changes from others.
- The local master is tested successfully.
- The local commits are pushed onto the remote master.
- The CI pipeline is correctly executed on the remote master by the CI server.

Implemented correctly, this practice leads to having an integrated, tested, working system at the end of each development interval, that can be shipped directly from the master branch.

However, this workflow relies on great discipline, and tends to suit small teams with a highly controlled work funnel that ensures work can be completed and tested on a daily iteration, with well defined and highly independent work packages. There is no buffer against integration failures, so the discipline must extend to dropping all other tasks until the master branch is stable again should there be issues, which will have associated productivity costs against the entire team.

Feature based branching

The SKA organisation advocates adopting a story-based branching model, often referred to as **feature branching**. This workflow effectively leverages GitLab **Merge Requests** enabling code reviews and continuous branch testing, but it is important to stress the importance of having short lived branches. It is easy to abuse this policy and have long living branches resulting in painful merge activities and dead or stale development lines. Bearing in mind that a *story* by definition is a piece of work a developer should conclude in the time of a sprint, the workflow should follow these steps:

- As a developer starts working from master on a new story, they create a new branch.
- The new branch shall be named after the story, i.e. *at1-26-the-new-widget*. Note: branch names are by convention all lower case.

```
$ git clone git@gitlab.com:ska-telescope/skampi.git
$ cd skampi
$ git branch
* master
$ git checkout -b at1-26-the-new-widget
$ git branch
master
* at1-26-the-new-widget
```

- All the commit messages contributing to the development of the story begin with the story ID, i.e. *AT1-26 - basic testing*.
- The developer makes sure that all tests execute correctly on their local story branch.
- When the story is ready for acceptance the developer pushes the story branch upstream.

```
$ git push -u origin at1-26-the-new-widget
Enumerating objects: 48, done.
Counting objects: 100% (48/48), done.
Delta compression using up to 12 threads
Compressing objects: 100% (23/23), done.
Writing objects: 100% (25/25), 4.80 KiB | 614.00 KiB/s, done.
Total 25 (delta 14), reused 0 (delta 0)
remote:
remote: To create a merge request for at1-26-the-new-widget, visit:
remote:   https://gitlab.com/ska-telescope/skampi/-/merge_requests/new?merge_request
↪%5Bsource_branch%5D=at1-26-the-new-widget
remote:
To gitlab.com:ska-telescope/skampi.git
* [new branch]      at1-26-the-new-widget -> at1-26-the-new-widget
Branch 'at1-26-the-new-widget' set up to track remote branch 'at1-26-the-new-widget'␣
↪from 'origin'.
```

- The branch CI pipeline is automatically triggered.
- A Merge Request is created on GitLab to merge the story branch into the master branch. The above commit reponse shows a conveniently supplied URL to start this process.
- Reviewers interact with comments on the Merge Request until all conflicts are resolved and reviewers accept the Merge Request.
- The Merge Request is merged into Master.
- The CI pipeline is executed successfully on the master branch by the CI server.

There are some considerations with Feature Branching:

- continually branching and merging is an overhead for small teams and very short work packages where there is a high prevalence of one-commit to one-merge-request
- branching requires discipline in that they should be short lived and developers need to remember to delete them after use
- stale and orphaned branches can pollute the repository
- developers must resolve merge conflicts with master before pushing changes, so there can be a race to merge to avoid these issues

Alternate Strategy

Whenever a team deviates from one of the recommended policies, it is important that the team captures its decision and publicly describe its policy, discussing it with the rest of the community.

See a more detailed description of this workflow at https://docs.gitlab.com/ee/topics/gitlab_flow.html

Merge requests

When the story is ready for acceptance a Merge Request should be created on GitLab to merge the story branch into the master branch. The Merge Request UI on GitLab includes a platform for the discussion threads, and indeed an important purpose of the Merge Request is to provide an online place for the team to discuss the changes and review the code before doing the actual merge.

It is recommended that A new Merge Request will include, among others, the following options:

- The Merge Request Title should always include the related JIRA issue id - this will be automatic following the above branching naming convention.
- Merge Request Description should include a concise, brief description about the issue.
- Add approval rules.
- Select one or more people for review (use the Reviewer field in the MR to differentiate between assignees and reviewers) and include anyone who has worked in the Merge Request.
- Delete source branch when Merge Request is accepted.
- Do not Squash commits when Merge Request is accepted.

At the moment the SKA organisation does not enforce approval rules, but it is recommended as good practice to involve other team members as assignees/reviewers for the Merge Request, and ensure that there is code review.

As part of best practices it is important to delete feature branches on merge or after merging them to keep your repository clean, showing only work in progress. It is not recommended to squash commits submitted to the remote server, in particular if using GitLab and JIRA integration, so the enabling squash commits option should be left unchecked. However you can arrange your commits before pushing them to remote.

Gitlab MR Settings for Project Maintainers

There are more additional settings in GitLab that only project maintainers could tune. The following settings are configured for the developer portal itself and they are the recommended settings for the projects in the SKA organisation. Normally, these settings would not be needed to change.

Note that the System team may from time to time batch update all of the SKA projects' settings as to confirm with the policies and recommendations.

2.5.6 Merge Request Quality Checks

To ensure the guidelines and policies described in this Developer Portal are followed for a consistent and robust development/security/review and Software Quality Assurance processes for SKA repositories, there are a series of automated checks in place. The result of the checks are reported back to the developers in the main Merge Request page on GitLab. It is advised to look for this comment and respond to any issue arisen.

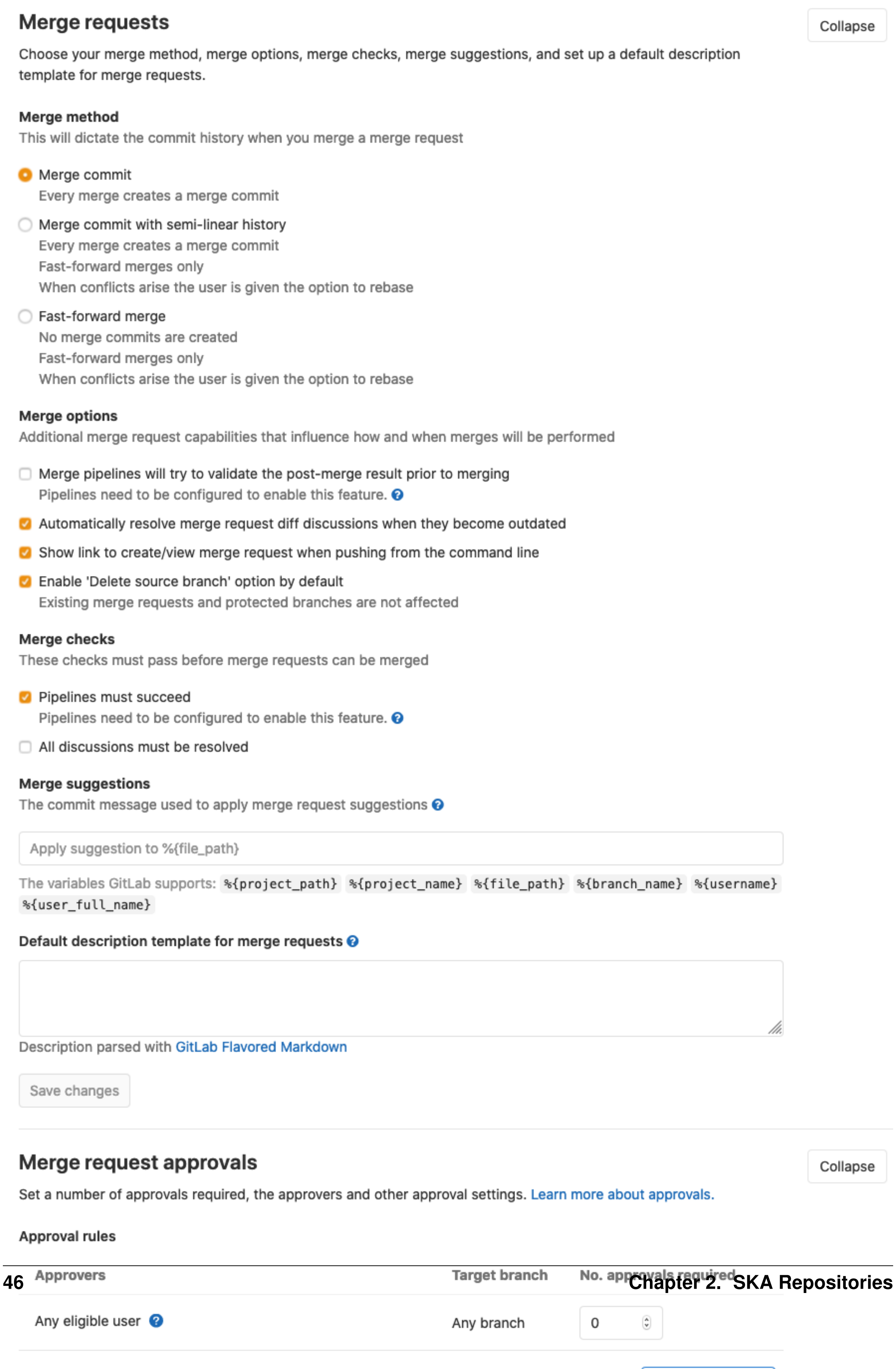
A check is either a:

- Failure (): The Merge Request is violating the SKA guidelines and it should be fixed by following the mitigation defined in the check
- Warning (): The Merge Request is following anti patterns/non-advised guidelines/policies and it would be better if it is fixed by the mitigation defined in the check
- Information (): You should be aware of the information conveyed in this Merge Request quality check message

Each check has a brief description that explains what it does and a mitigation/explanation (depending on check type) which gives detailed information about the check and how to fix it or explains its findings more. You can find a list of each check below.

Workflow

When a new Merge Request is created, a webhook triggers the SKA MR Service to carry out the checks described below and **Marvin the Paranoid Android** (*username: marvin-42*) happily reports back to the Merge Request by adding a comment (probably the first comment). The comment includes a table (like the example below) with each check and associated information.



Merge requests

Choose your merge method, merge options, merge checks, merge suggestions, and set up a default description template for merge requests.

Merge method

This will dictate the commit history when you merge a merge request

[✓] Merge commit
Every merge creates a merge commit

[ ] Merge commit with semi-linear history
Every merge creates a merge commit
Fast-forward merges only
When conflicts arise the user is given the option to rebase

[ ] Fast-forward merge
No merge commits are created
Fast-forward merges only
When conflicts arise the user is given the option to rebase

Merge options

Additional merge request capabilities that influence how and when merges will be performed

[ ] Merge pipelines will try to validate the post-merge result prior to merging
Pipelines need to be configured to enable this feature.

[✓] Automatically resolve merge request diff discussions when they become outdated

[✓] Show link to create/view merge request when pushing from the command line

[✓] Enable 'Delete source branch' option by default
Existing merge requests and protected branches are not affected

Merge checks

These checks must pass before merge requests can be merged

[✓] Pipelines must succeed
Pipelines need to be configured to enable this feature.

[ ] All discussions must be resolved

Merge suggestions

The commit message used to apply merge request suggestions

Apply suggestion to %{file_path}

The variables GitLab supports: `%{project_path}` `%{project_name}` `%{file_path}` `%{branch_name}` `%{username}` `%{user_full_name}`

Default description template for merge requests

Description parsed with GitLab Flavored Markdown

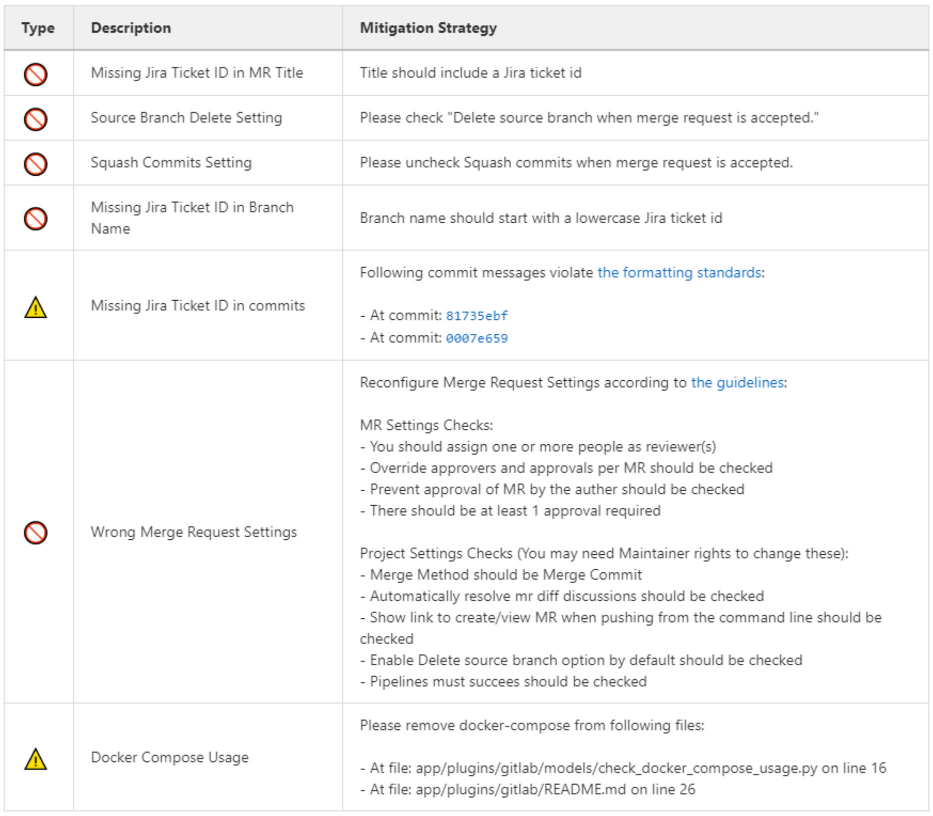
Merge request approvals

Set a number of approvals required, the approvers and other approval settings. Learn more about approvals.

Approval rules

Approvers	Target branch	No. approvals required
Any eligible user	Any branch	0

For the subsequent changes pushed to the Merge Request, the comment is updated to reflect the latest status of the Merge Request.



Type	Description	Mitigation Strategy
🚫	Missing Jira Ticket ID in MR Title	Title should include a Jira ticket id
🚫	Source Branch Delete Setting	Please check "Delete source branch when merge request is accepted."
🚫	Squash Commits Setting	Please uncheck Squash commits when merge request is accepted.
🚫	Missing Jira Ticket ID in Branch Name	Branch name should start with a lowercase Jira ticket id
⚠	Missing Jira Ticket ID in commits	Following commit messages violate the formatting standards: - At commit: `81735ebf` - At commit: `0007e659`
🚫	Wrong Merge Request Settings	Reconfigure Merge Request Settings according to the guidelines: MR Settings Checks: - You should assign one or more people as reviewer(s) - Override approvers and approvals per MR should be checked - Prevent approval of MR by the auther should be checked - There should be at least 1 approval required Project Settings Checks (You may need Maintainer rights to change these): - Merge Method should be Merge Commit - Automatically resolve mr diff discussions should be checked - Show link to create/view MR when pushing from the command line should be checked - Enable Delete source branch option by default should be checked - Pipelines must succees should be checked
⚠	Docker Compose Usage	Please remove docker-compose from following files: - At file: app/plugins/gitlab/models/check_docker_compose_usage.py on line 16 - At file: app/plugins/gitlab/README.md on line 26

Fig. 3: Marvin's Check Table.

Checks

Type	Description	Mitigation Strategy
Failure	Missing Jira Ticket ID in MR Title	Title should include a valid Jira ticket id
Warning	Docker-Compose Found	**Please remove docker-compose from following files:** • At file: <file_location> on line <line_number> • At file: <file_location> on line <line_number>
Failure	Missing Jira Ticket In Branch Name	Branch name should start with a lowercase Jira ticket id
Failure	Wrong Merge Request Setting	Reconfigure Merge Request Settings according to *Merge requests* **MR Settings Checks:** • You should assign one or more people as reviewer(s) • Automatically resolve mr diff discussions should be checked • Override approvers and approvals per MR should be checked • Remove all approvals when new commits are pushed should be checked • Prevent approval of MR by the author should be checked • There should be at least 1 approval required • Please uncheck Squash commits when Merge Request is accepted. • Please check Delete source branch when merge request is accepted. **Project Settings Checks(You may need Maintainer r** • Pipelines must succeed should be checked • Enable Delete source branch option by default should be checked • Show link to create/view MR when pushing from the command line should be checked
Warning	Missing Jira Ticket in commits	**Following commit messages violate *Committing code*** • <commit-hash>

Missing Jira Ticket ID in MR Title

This check warns users from raising a Merge Request without A Jira ticket ID in Merge Request title. This will make every Merge Request identifiable with its Jira ticket (through the GitLab/Jira integration). The level of this check is a failure, and to avoid it users should include a valid Jira ticket id in title of the Merge Request.

Docker-Compose Found

This check is to prevent users from using Docker-Compose in their project. This will make it easier to remove Docker-Compose from the projects as it shouldn't be used anymore (creates issues with the underlying networks). To avoid this warning, the user needs to remove Docker-Compose from the project. The details of the files involved can be seen in the warning message under the Mitigation Strategy column along with the line numbers where Docker-Compose is found.

Missing Jira Ticket In Branch Name

This check warns users from raising a Merge Request without A Jira ticket ID in the branch name. This will make every branch identifiable with its Jira ticket. The level of this check is a failure, and to avoid it users should follow the steps listed in *Master based development*.

Wrong Merge Request Setting

This check warns users from merging their branch without the Merge Request being configured with the right settings. The level of this check is a failure, and to avoid it the Merge Request should be configured as listed in *Merge requests*. Some of the settings can only be changed by the maintainers. These settings are listed in *Gitlab MR Settings for Project Maintainers*.

Missing Jira Ticket in commits

This check warns users of any commit that was made without using a Jira ticket ID in it's message. Having the Jira ticket ID at the beginning of your commit messages is one of the basic rules listed at *Committing code*. The Jira Ticket ID in the commit messages are used by the developers to keep track of the changes made on the ticket through JIRA, and is a key part of the Software Quality Assurance programme.

Pipeline Checks

This check warns users from merging their Merge Request without having a pipeline with the needed jobs. The level of this check is a failure, and to avoid it 2 steps may be needed. The first one is to create a pipeline (i.e. add .gitlab-ci.yml) if there is not one created yet. The second one can only be done after the first one, and it consists on including the jobs that are listed on the mitigation strategy column (i.e. helm-publish) in the created pipeline. How to add the jobs to the pipeline is explained on the developer portal (job name as hyperlink).

Non-compliant License Information

This check warns users if license in their project is not compatible with SKA approved license so that the quality of the software is improved and compliance is ensured with the SKA standards. This does not apply to projects in the 'External' project.

2.6 Working with Jira

Todo:

- Add info about Jira, ticket types, workflows etc.

2.6.1 GitLab to Jira Integration

If you commit message or make a merge request (MR) in GitLab has **Jira Issue ID mentioned**, then:

- GitLab will hyperlink the issue for easy navigation
- Jira issue will have an issue link to the commit/MR
- Jira issue will have a comment reflecting the comment made in GitLab, the comment author, and a link to the commit/MR in GitLab (If it is enabled)

  Example Commit Message: `SKA-34 added gitlab-jira integration`

If you mention that **a commit or MR 'closes', 'resolves', or 'fixes' a Jira issue ID**, then:

- GitLab's merge request page displays a note that it "Closed" the Jira issue, with a link to the issue. (Note: Before the merge, an MR will display that it "Closes" the Jira issue.)
- Jira issue will transition to `READY FOR ACCEPTANCE` status if applicable

  Example Commit Message: `closes SKA-34`

Also, You can do other things like adding comments to Jira issues, time-tracking and transitioning Jira issue states directly from GitLab commits. You can find more about it in Jira Smart Commits.

More info could be found at GitLab to Jira Integration

2.6.2 Jira to GitLab Integration

A Development Panel is added automatically to any Jira issues referred by its ID in:

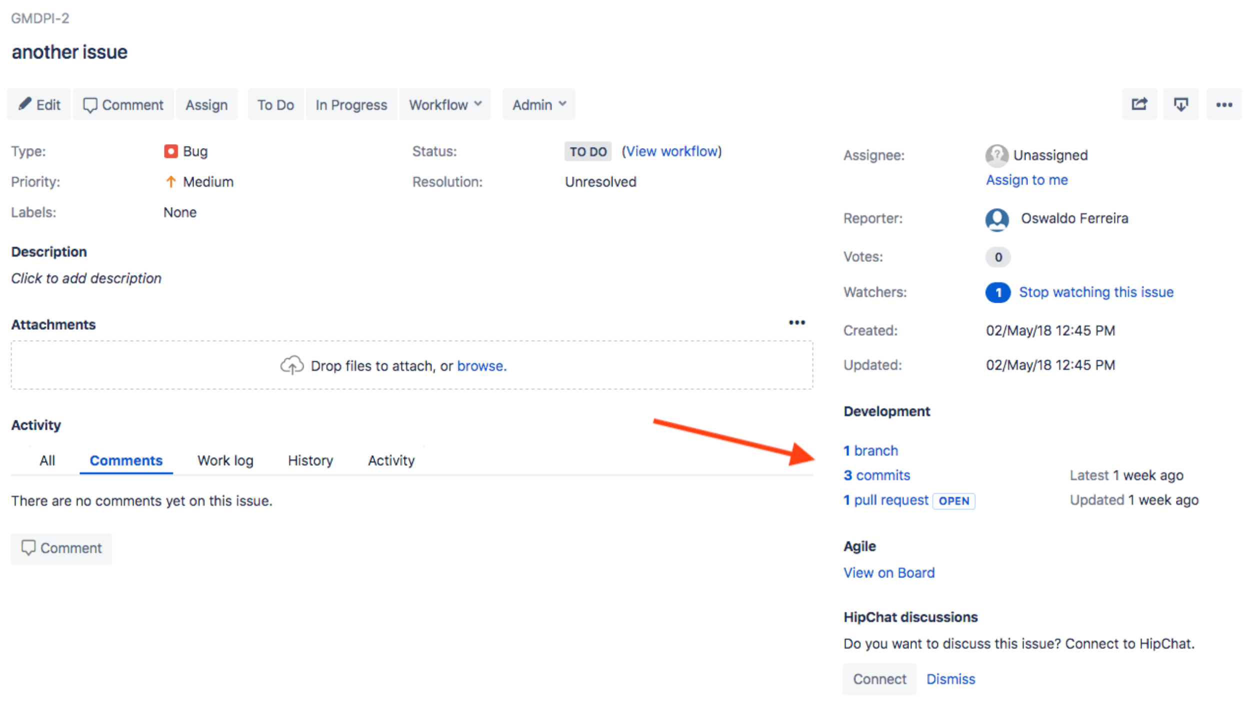
- branch names
- commit messages
- merge request titles

in GitLab and you will be able to see the linked `branches`, `commits`, and `merge requests` when entering a Jira issue (inside the Jira issue, merge requests will be called "pull requests").

More info could be found at Jira to GitLab Integration

2.7 CI/CD

How the SKA does Continuous Integration and Deployment, using GitLab's tools. These pages describe the full *CI process*, the *GitLab Global Variables* used for projects, and a mirror of the *CI-Dashboard*.


GMDPI-2

another issue

Type: Bug
Priority: Medium
Labels: None
Status: TO DO (View workflow)
Resolution: Unresolved

Description
Click to add description

Attachments
Drop files to attach, or browse.

Activity
All Comments Work log History Activity

There are no comments yet on this issue.

Assignee: Unassigned
Assign to me
Reporter: Oswaldo Ferreira
Votes: 0
Watchers: 1 Stop watching this issue
Created: 02/May/18 12:45 PM
Updated: 02/May/18 12:45 PM

Development
1 branch
3 commits Latest 1 week ago
1 pull request OPEN Updated 1 week ago

Agile
View on Board

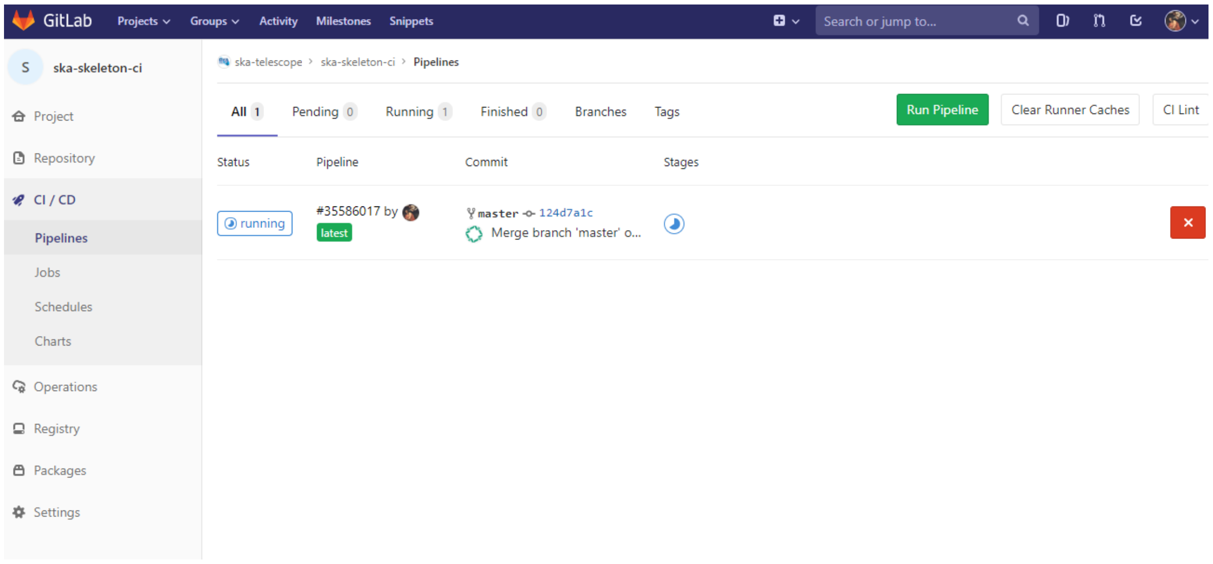
HipChat discussions
Do you want to discuss this issue? Connect to HipChat.

Fig. 4: Jira Development Panel.

2.7.1 Continuous Integration

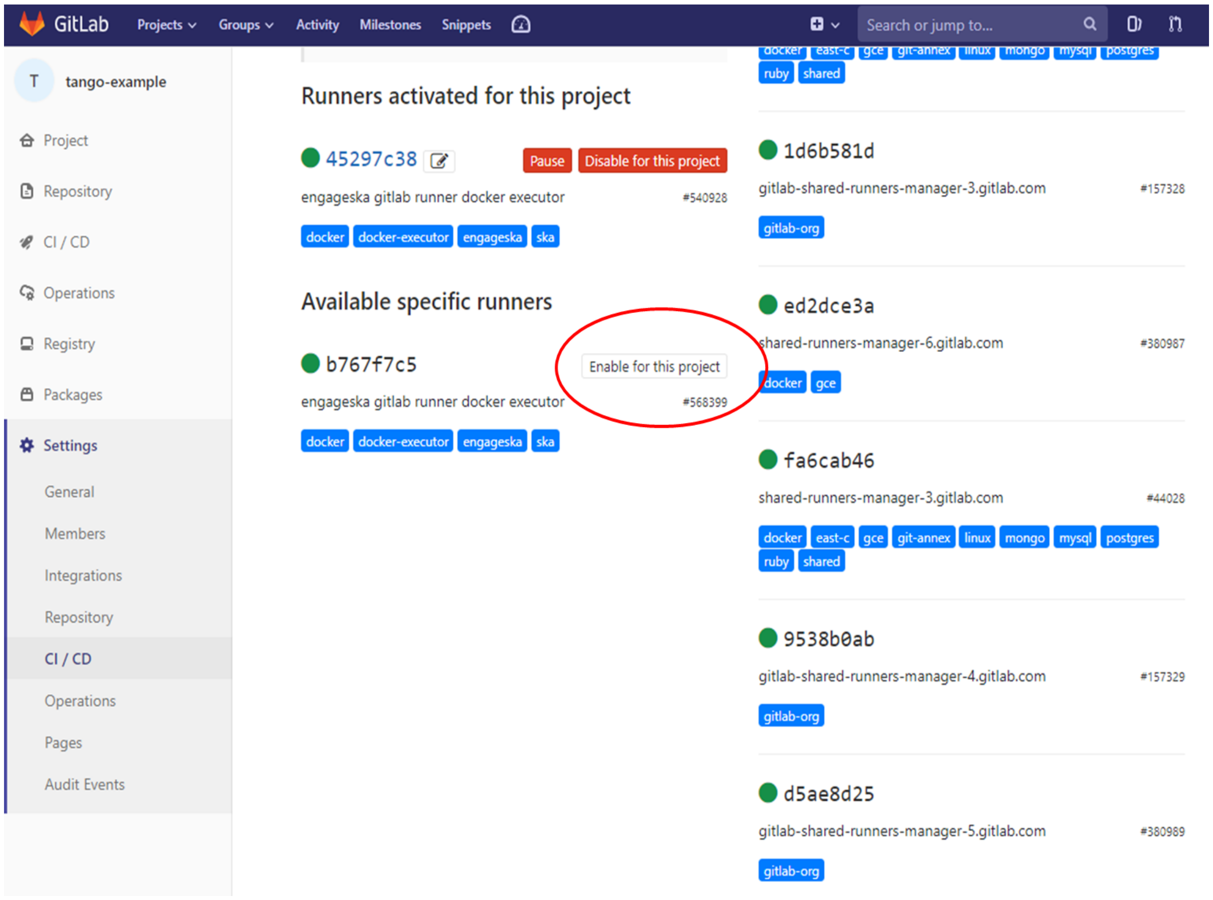
Configuring a CI pipeline

To enable the Gitlab automation, it is needed to insert a configuration file that must be placed in the root of the repository and called ".gitlab-ci.yml". It mainly contains definitions of how your project should be built. An example of it can be found within the project "ska-python-skeleton" available here. Once the file is in the root directory, it is possible to run the CI pipeline manually (creating a pipeline) or with a commit in gitlab as soon as the mirroring finishes. The following pipeline was created manually pressing the button "Run pipeline" on a specific branch (i.e. master).



Using a specific executor

The pipeline by default will run with a shared runner made available from GitLab. It is also possible to assign specific SKA runners to the project (by adding the tags). To do that the option must be enabled:



The EngageSKA cluster located at the Datacenter of Institute of Telecommunication (IT) in Aveiro provides some virtual machines available adding the tag "engageska" or "docker-executor" as shown here.

Note: In order to have the SKA runners available, a project *must* be under the SKA Telescope group. Currently projects can only be added to the SKA group by the System Team - refer to our guide to *Create a new project*.

The typically used SKA CI Pipeline stages are being documented as they are developed and improved. The currently available documentation on this is discussed under *CI pipeline stage descriptions*. The following section deals with the required CI metrics that need to be included in the pipeline.

Automated Collection of CI health metrics as part of the CI pipeline

As part of the CI/CD process all teams are expected to collect and consolidate the required code health metrics. Namely **unit tests**, **linting (static code analysis)** and **coverage**.

Part of the CI/CD functionality is to add a quick glance of those metrics to each repository in the form of badges. These badges will always show the status of the **default** branch in each repository.

Teams have the option to use the automatic parsing of their CI and code health metrics and have the badges created automatically as long as the output from their code health reports follows the requirements described bellow. As an alternative the teams can instead create the `ci-metrics.json` file themselves according to what is described in *Manual Metrics*.

These metrics reports must pass the following requirements:

1. These files must **not** be part of the repository, but be created under their respective steps (`test`, `linting`) in the CI pipeline.
2. Unit Tests report must be a JUnit XML file residing under `./build/reports/unit-tests.xml`
3. Linting report must be a JUnit XML file residing under `./build/reports/linting.xml`
4. Coverage report must be a XML file in the standard used by *Coverage.py* residing under `./build/reports/code-coverage.xml`
5. The XML format expected for the coverage is the standard XML output from Coverage.py for Python or from a similar tool like Cobertura for Javascript with the `line-rate` attribute specifying the coverage. See the example code bellow.

```
<?xml version="1.0" encoding="UTF-8"?>
<coverage branch-rate="0" branches-covered="0" branches-valid="0" complexity="0" line-
↪rate="0.6861" lines-covered="765" lines-valid="1115" timestamp="1574079100055"␣
↪version="4.5.4">
```

Note: To always ensure these requirements are fulfilled, you should copy/move the files in the *after_script* part of your job definition instead of *script* part after running tests etc. since if the tests/linting fails then the files won't be copied. For example:

```
# Do not use this:
job:
  ...
  script:
    ...
    - python3 -m pytest ...
    - cp unit-tests.xml report.json cucumber.json ../build/reports/
    ...

# Use this instead:
```

(continues on next page)

(continued from previous page)

```
job:
  ...
  script:
    ...
    - python3 -m pytest ...
  after_script:
    - cp unit-tests.xml report.json cucumber.json ../build/reports/
    ...
```

In order to automate the process as much as possible for the teams, the templates repository repository was created and it will automate the all metrics collection, and badge creation as long as the 5 points above are observed.

In order to use this automation, the *post_step* from the *templates-repository* must be included, i.e.: `.gitlab-ci.yml`.

```
# Create Gitlab CI badges from CI metrics
# https://developer.skatelescope.org/en/latest/tools/continuousintegration.html
↪#automated-collection-of-ci-health-metrics-as-part-of-the-ci-pipeline
include:
  - project: 'ska-telescope/templates-repository'
    file: 'gitlab-ci/includes/post_step.yml'
```

Note: You can't redefine the *.post* step in your CI code, or it will break the functionality. In case you need to use the *.post* step for the CI pipeline then you must use the manual method for generating the badges.

Manual Collection of CI health metrics as part of the CI pipeline

The teams that prefer to create their own `ci-metrics.json` file instead of using the provided automation, can do so. They are still expected to observe all the points described in *Automated Metrics*.

The `ci-metrics.json` file is expect to be created automatically as part of the CI pipeline by the teams by collecting the relevant information from the *unit tests*, *coverage*, *linting* and *build status*. **An important point to notice, is that** `ci-metrics.json` **shouldn't exist as part of the repository, but, be created specifically as part of the CI pipeline.** The file must be created and properly populated before the start of the marked `stage:  .post` step in the CI pipeline (`.gitlab-ci.yml` file).

The metrics should be collected under the following structure:

- **commit-sha** (string): *sha tag for the git commit*
- **build-status**: *top level placeholder for the build process status*
  - **last**: *placeholder about the last build process*
    - **timestamp** (float): *the Unix timestamp with the date and time of the last build status*
- **coverage**: *placeholder about the unit test coverage*
  - **percentage** (float): *the coverage percentage of the unit tests*
- **tests**: *placeholder about the unit tests*
  - **errors** (int): *number of test errors*
  - **failures** (int): *number of test failures - this denotes a serious error in the code that broke the testing process*
  - **total** (int): *total number of tests*
- **lint**: *placeholder about the linting (static code analysis)*
  - **errors** (int): *number of linting errors*

- **failures** (int): *number of linting failures - this denotes a serious error in the code that broke the linting process*
- **total** (int): *total number of linting tests*

`ci-metrics.json` example:

```
{
  "commit-sha": "cd07bea4bc8226b186dd02831424264ab0e4f822",
  "build-status": {
      "last": {
          "timestamp": 1568202193.0
      }
  },
  "coverage": {
      "percentage": 60.00
      },
  "tests": {
      "errors": 0,
      "failures": 3,
      "total": 170
  },
  "lint": {
      "errors": 4,
      "failures": 0,
      "total": 7
  }
}
```

CI pipeline stage descriptions

Caution: This section is a work in progress

The CI/CD pipeline will ensure that software projects are packaged, tested and released in a consistent and predictable manner. SKA Pipelines are viewable and executable at https://gitlab.com/ska-telescope

General Notes

- Every commit could potentially trigger a pipeline build. There may be different rules applied to determine which stages are executed in the pipeline based on factors like the branch name.
  - E.g Every commit in a feature branch may trigger the "Lint" stage, but not a slow test suite.
- When doing a release with a git tag, the full pipeline will be run.
- Every pipeline job is associated with its git commit (including tag commits).
- Try and have the stages complete as fast as possible.
  - In some cases it may be possible to parallelize jobs. For example, unit tests and static analysis could be run in parallel.
- All projects must include all the stages listed below.
- Project dependencies must be stored in, and made available from the SKA software repository.
- All tests must pass on the "master" branch and should be kept stable.

Stages

Build

The build stage packages/compiles the software project into distributable units of software. The project will be checked out at the git commit hash. This specific version of the code must then be built. Failing the build stage will stop the further steps from being executed. Where possible Semantic Versioning should be used. To create a release a git tag should be used. See *Software Package Release Procedure* for details.

Input Git commit hash

Output A distributable unit of software. E.g .deb .whl .jar or docker image. These must be stored as part of the artifacts and will then be available to subsequent jobs. One could also store metadata together with the artefact, such as a hash of the binary artefact. This should be provided by our artefact registry.

Linting

The static analysis stage does static code analysis on the source code such as Linting.

Input None

Output Quality analysis results in JUnit format.

Test

The test stage must install/make use of the packages created during the build stage and execute tests on the installed software. Tests should be grouped into Fast / Medium / Slow / Very Slow categories. For more details, read the *Software Testing Policy and Strategy*.

Input The output from the Build stage. E.g .deb or .whl or docker image. Input could also consist of test data or environment.

Output

- The results of the tests in JUnit format. These need to be added to the artifacts. See Gitlab Test Reports.
- Coverage metrics in JUnit format.

Test types

Todo:

- Further define components to be mocked or not
- Further define smoke/deployments tests

Unit tests The smallest possible units/components are tested in very fast tests. Each test should complete in milliseconds.

Component tests Individual components are tested.

Integration/Interface tests Components are no longer being mocked, but the interactions between them are tested. If a component is a docker image, the image itself should be verified along with its expected functionality.

Deployment tests Tests that software can be deployed as expected and once deployed, that it behaves as expected.

Configuration tests Multiple combinations of software and hardware are tested.

System tests The complete solution, integrated hardware and software is tested. There tests ensure that the system requirements are met.

Publish

Once the build and test stages have completed successfully the output from the build stage is uploaded to the SKA software repository. This stage may only be applicable on git tag commits for full releases in certain projects.

Input The output from the Build stage. .deb or .whl for example. This could also include docker images.

Output The packages are uploaded to the SKA software repository.

Pages

This is a gitlab stage publishes the results from the stages to Gitlab

Input The JUnit files generated in each pipeline stage.

Output The generated HTML containing the pipeline test results.

Documentation

Currently the documentation is generated by the "readthedocs" online service. The list of SKA projects available *Alpahbetical list of projects and subgroups*. The project documentation will be updated and accessible at the following URL https://developer.skatelescope.org/projects/<PROJECT> E.g lmc-base-classes

Input A *docs* folder containing the project documentation.

Output The generated HTML containing the latest documentation.

Using environment variables in the CI pipeline to upload to the Central Artefact Repository

There are several environment variables available in the CI pipeline that should be used when uploading Python packages and Docker images to the Central Artefact Repository. This will make these packages available to the rest of the SKA project. This section describes some of these variables. A *full list* is also available.

Python Modules

The Central Artefact Repository PYPI destination as well as a username and password is available. For a reference implementation see the lmc-base-classes .gitlab-ci.yaml

Note the following:

- The Central Artefact Repository CAR_PYPI_REPOSITORY_URL is where the packages will be uploaded to.
- *twine* uses the local environment variables (*TWINE_USERNAME*, *TWINE_PASSWORD*) to authenticate the upload, therefore they are defined in the *variables* section.

```
publish to nexus:
  stage: publish
  tags:
    - docker-executor
  variables:
    TWINE_USERNAME: $CAR_PYPI_USERNAME
    TWINE_PASSWORD: $CAR_PYPI_PASSWORD
  script:
    # check metadata requirements
    - scripts/validate-metadata.sh
    - pip install twine
    - twine upload --repository-url $CAR_PYPI_REPOSITORY_URL dist/*
  only:
    variables:
      - $CI_COMMIT_MESSAGE =~ /^.+$/ # Confirm tag message exists
      - $CI_COMMIT_TAG =~ /^((([0-9]+)\.([0-9]+)\.([0-9]+)(?:-([0-9a-zA-Z-]+(?:\.[0-
↪9a-zA-Z-]+)*))?)(?:\+([0-9a-zA-Z-]+(?:\.[0-9a-zA-Z-]+)*))?)$/ # Confirm semantic␣
↪versioning of tag
```

Docker images

The Central Artefact Repository Docker registry host and user is available. For a reference implementation see the SKA docker gitlab-ci.yml

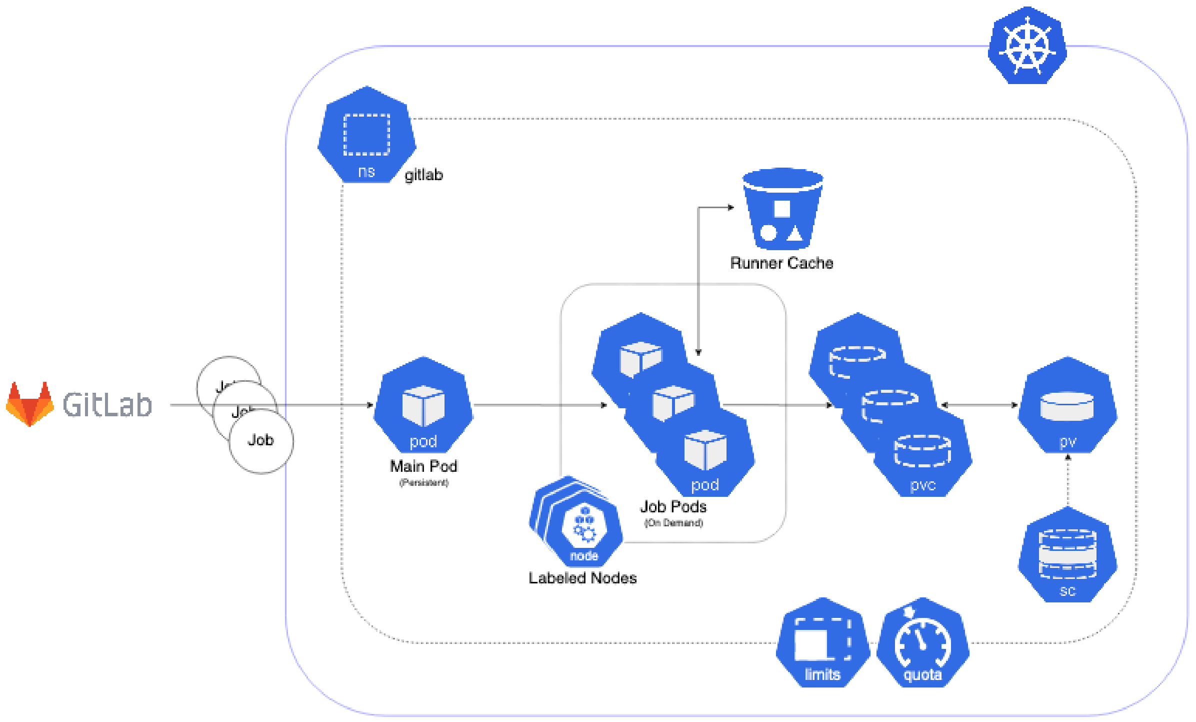
Note the following:

- The *DOCKER_REGISTRY_USER* corresponds to the folder where the images are uploaded, hence the *$CAR_OCI_REGISTRY_USERNAME* is used.

```
script:
- cd docker/tango/tango-cpp
- echo ${CAR_OCI_REGISTRY_PASSWORD} | docker login --username ${CAR_OCI_REGISTRY_
↪USERNAME} --password-stdin ${CAR_OCI_REGISTRY_HOST}
- make DOCKER_BUILD_ARGS="--no-cache" DOCKER_REGISTRY_USER=$CAR_OCI_REGISTRY_USERNAME␣
↪DOCKER_REGISTRY_HOST=$CAR_OCI_REGISTRY_HOST build
- make DOCKER_REGISTRY_USER=$CAR_OCI_REGISTRY_USERNAME DOCKER_REGISTRY_HOST=$CAR_OCI_
↪REGISTRY_HOST push
```

Kubernetes based Runners Architecture

GitLab runners are orchestrated by Kubernetes cluster. They could be deployed to any Kubernetes clusters with following the instructions on deploy-gitlab-runners repository. The main architecture is illustrated below.



Features

- The main runner pod is deployed with Helm Chart under **gitlab** namespace with the repository.
- Main runner pod is registered to **ska-telescope group** shared runners with configurable tags.
- The main pod picks up **GitLab Jobs** and creates **on-demand pods**. This is configured using helm chart values file/or config.toml file of GitLab runners below.
- Runners are scaled according to configuration.
- Runners have resource **limits** *i.e. cpuRequests, memoryRequests, cpuLimit, memoryLimit*. This is not applied at the moment.
- Runners are running in nodes that are **specifically labelled** for ci/cd jobs.
- Runners share a **cache** between them that is used to speed up the job times.
- **Docker support**
- **Kubernetes support**

With this approach, GitLab Runners are proven to be a viable option to be used in a cluster with auto-scaling and easy management. Docker Support

Docker can be used in the CI/CD jobs as with the normal runners. Note that: docker-compose cannot be used in conjunction with Kubernetes! You should follow the instruction on the developer portal to set up your repo.

To elevate some of the security concerns listed below with using Docker in Docker, another docker daemon is deployed in the nodes. This daemon then used as default docker-daemon in the runner pods. Kubernetes Support

Kubernetes clusters could be created in ci/cd jobs. These clusters are created on the ci-worker nodes and destroyed at the end of the job.

Note: in order to run deploy clusters, the account permissions need to be set up correctly for the runner services.

Migrating to new Runner Infrastructure

Compose is a commonly used tool for defining and running multi-container Docker applications. It works in all environments including CI workflows and requires a three-step process:

1. Prepare your Dockerfile.
2. Define the services that make up your app in `docker-compose.yml`.
3. Run `docker-compose up`.

The SKA is currently promoting migration to Kubernetes as the container orchestrator, and this requires for applications developed with docker-compose to be converted into the new runner infrastructure. In principle there is no need to make any changes if one is not using docker-compose.

The conversion tool allowing the migration is Kompose. A detailed guide for changing from docker-compose to kubernetes can be found at https://kubernetes.io/docs/tasks/configure-pod-container/translate-compose-kubernetes/

The conversion process is relatively straightforward, requiring two steps:

1. Run `kompose convert` in the same directory of `docker-compose.yml` file.
2. Prepare a make target with `kubectl apply -f <output_file>`.

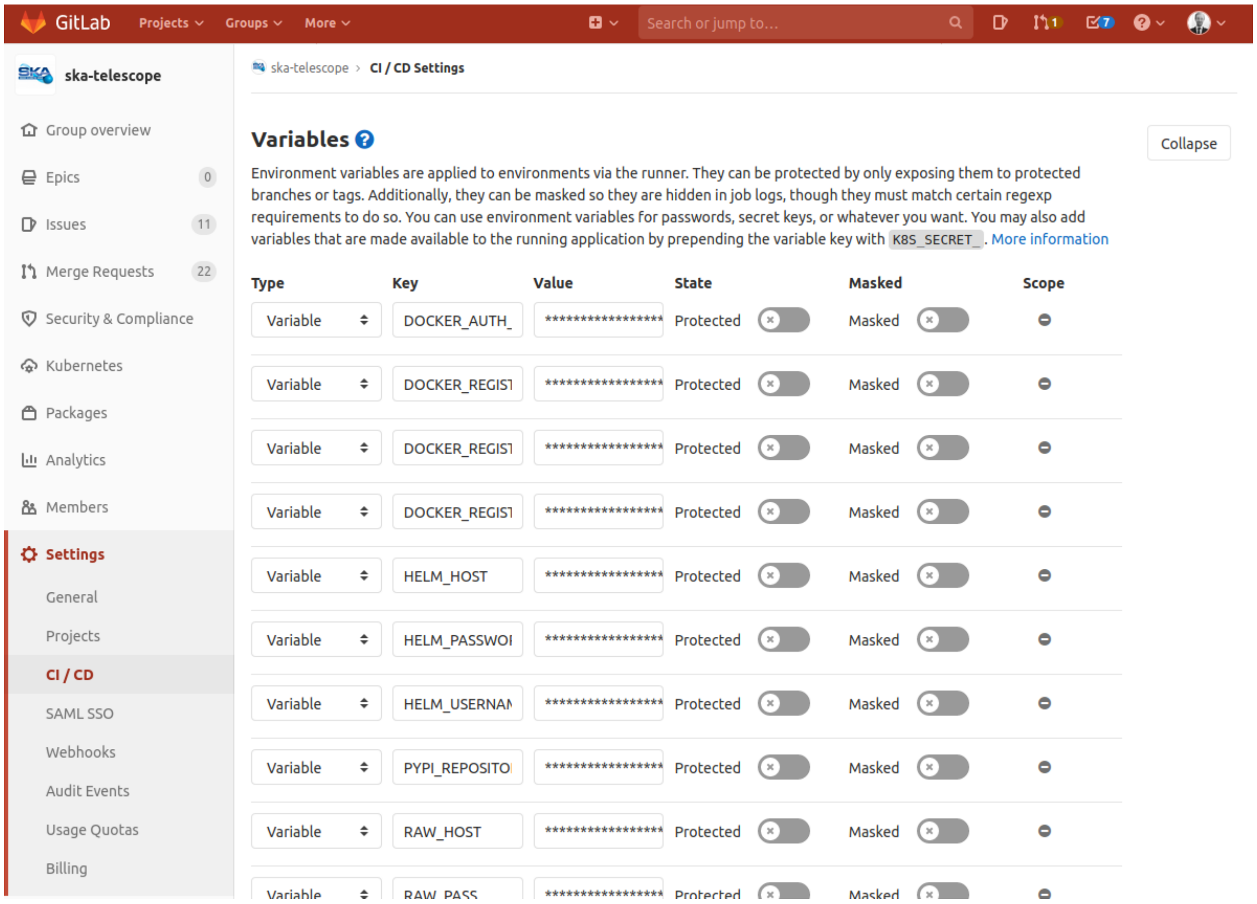
This converts the `docker-compose.yml` file to files that you can use with `kubectl`. To make sure that the transition will work in the runner cluster you can test locally with Minikube. If it works on your local Minikube then it will work in the kubernetes runner cluster.

2.7.2 GitLab Global Variables

This section describes the global variables that are presently being used as part of the GitLab CI/CD infrastructure.

Variables Interface

The variables are set under the CI/CD Settings on GitLab.



Variables

Environment variables are applied to environments via the runner. They can be protected by only exposing them to protected branches or tags. Additionally, they can be masked so they are hidden in job logs, though they must match certain regexp requirements to do so. You can use environment variables for passwords, secret keys, or whatever you want. You may also add variables that are made available to the running application by prepending the variable key with `K8S_SECRET_`. More information

Type	Key	Value	State	Masked	Scope
Variable	DOCKER_AUTH_	*****************	Protected	Masked	⊖
Variable	DOCKER_REGIST	*****************	Protected	Masked	⊖
Variable	DOCKER_REGIST	*****************	Protected	Masked	⊖
Variable	DOCKER_REGIST	*****************	Protected	Masked	⊖
Variable	HELM_HOST	*****************	Protected	Masked	⊖
Variable	HELM_PASSWOI	*****************	Protected	Masked	⊖
Variable	HELM_USERNAN	*****************	Protected	Masked	⊖
Variable	PYPI_REPOSITO	*****************	Protected	Masked	⊖
Variable	RAW_HOST	*****************	Protected	Masked	⊖
Variable	RAW_PASS	*****************	Protected	Masked	⊖

Fig. 5: The variables interface on GitLab

Variables Description

Key	Description	Notes
`CAR_OCI_REGISTRY_HOST`	Is the FQDN of the Central Artefact Repository - artefact.skatelescope.org	
`CAR_OCI_REGISTRY_USERNAME`	OCI Image Publishing user name	
`CAR_OCI_REGISTRY_PASSWORD`	OCI Publishing user password	
`CAR_PYPI_REPOSITORY_URL`	https://artefact.skatelescope.org/repository/pypi-internal/	
`CAR_PYPI_USERNAME`	PyPi Publishing user name	
`CAR_PYPI_PASSWORD`	PyPi user password	
`CAR_HELM_REPOSITORY_URL`	Is the FQDN of the Central Artefact Repository - artefact.skatelescope.org	
`CAR_HELM_USERNAME`	Helm Chart Publishing user name	
`CAR_HELM_PASSWORD`	Helm Chart Publishing user password	
`CAR_ANSIBLE_REPOSITORY_URL`	https://artefact.skatelescope.org/repository/ansible-internal	
`CAR_ANSIBLE_USERNAME`	Ansible role/collection Publishing user name	
`CAR_ANSIBLE_PASSWORD`	Ansible role/collection Publishing user password	
`CAR_RAW_REPOSITORY_URL`	https://artefact.skatelescope.org/repository/raw-internal	
`CAR_RAW_USERNAME`	Raw Repository Publishing user name	
`CAR_RAW_PASSWORD`	Raw Repository Publishing user password	

Historical Variable Use (Deprecated)

The following variables have been used historically to drive behaviour in the pipelines, but must be set at the individual repository level:

Key	Description	Notes
`DOCKER_REGISTRY_FOLDER`	Specify the base path pre-fixed for an image eg: `ska-docker`	Used to produce the image path: with `DOCKER_REGISTRY_HOST/DOCKER_REGISTRY_FOLDER/<image-name>`
`DOCKER_REGISTRY_HOST`	FQDN of the Nexus Docker registry	Replaced by `CAR_OCI_REGISTRY_HOST`
`DOCKER_REGISTRY_PASSWORD`	Password for uploading to the Nexus registry	Replaced by `CAR_OCI_REGISTRY_USERNAME`
`DOCKER_REGISTRY_USERNAME`	Username for uploading to the Nexus registry	Replaced by `CAR_OCI_REGISTRY_PASSWORD`
`HELM_HOST`	FQDN of the Helm Chart repository	Replaced by `CAR_HELM_REPOSITORY_URL`
`HELM_USERNAME`	Username for uploading to the Helm Chart repo	Replaced by `CAR_HELM_USERNAME`
`HELM_PASSWORD`	Password for uploading to the Helm Chart repo	Replaced by `CAR_HELM_PASSWORD`
`PYPI_REPOSITORY_URL`	FQDN of the PyPi Nexus repository	Replaced by `CAR_PYPI_REPOSITORY_URL`
`TWINE_USERNAME`	Username for uploading to the PyPi Nexus repo	Replaced by `CAR_PYPI_USERNAME`
`TWINE_PASSWORD`	Password for uploading to the PyPi Nexus repo	Replaced by `CAR_PYPI_PASSWORD`
`RAW_HOST`	FQDN for the Raw file hosting	Replaced by `CAR_RAW_REPOSITORY_URL`
`RAW_USER`	Username for uploading to the Raw repo on Nexus	Replaced by `CAR_RAW_USERNAME`
`RAW_PASS`	Password for uploading to the Raw repo on Nexus	Replaced by `CAR_RAW_PASSWORD`

2.7.3 CI-Dashboard

The following table is automatically extracted from our gitlab project dashboard page at [https://ska-telescope.gitlab.io/ska_ci_dashboard/]

Inputs		Output
A	B	A or B
False	False	False
True	False	True
False	True	True
True	True	True

2.8 BDD testing guide

This is based off work originally done by Giorgio Brajnik and Ross Lipscomb. It focuses on the architecture of our BDD system, and the context of BDD testing. For a checklist to use when working with BDD tests, see *BDD Walkthrough*.

2.8.1 What is BDD testing?

BDD stands for Behaviour Driven Development. Thus BDD is an approach based on a testing style that concentrates on describing and evaluating how the System Under Test (SUT) behaves in response to specified inputs. BDD testing typically focuses on component interactions or other system-level tests. (For more on the different levels of testing, see *Software Testing Policy and Strategy*).

BDD tests are defined using simple natural language formulations, so BDD tests can be understood by people who aren't developers (though these people will typically have a good understanding of the domain that's being tested). Tests are defined using "given", "when", and "then" steps; this uses the Gherkin specification language. We'll look at a short example:

```
Given the SKA Community Confluence website
And I am not logged in
When I click on the login button
Then I see a login page
```

These three steps (augmented by an "And" step - which can be used in all three steps) define a simple test for a website. I can carry this test out manually, or I can work with programmers to automate it. It's easy to understand this test, though some tests may require more domain knowledge (for example if you're setting a test around telescope pointing, or one on the control system).

BDD tests thus provide a way in to the verification tests that need to be carried out. You can read more about BDD testing at:

- https://github.com/pytest-dev/pytest-bdd
- https://automationpanda.com/2018/10/22/python-testing-101-pytest-bdd/

2.8.2 Why do BDD tests for the SKA?

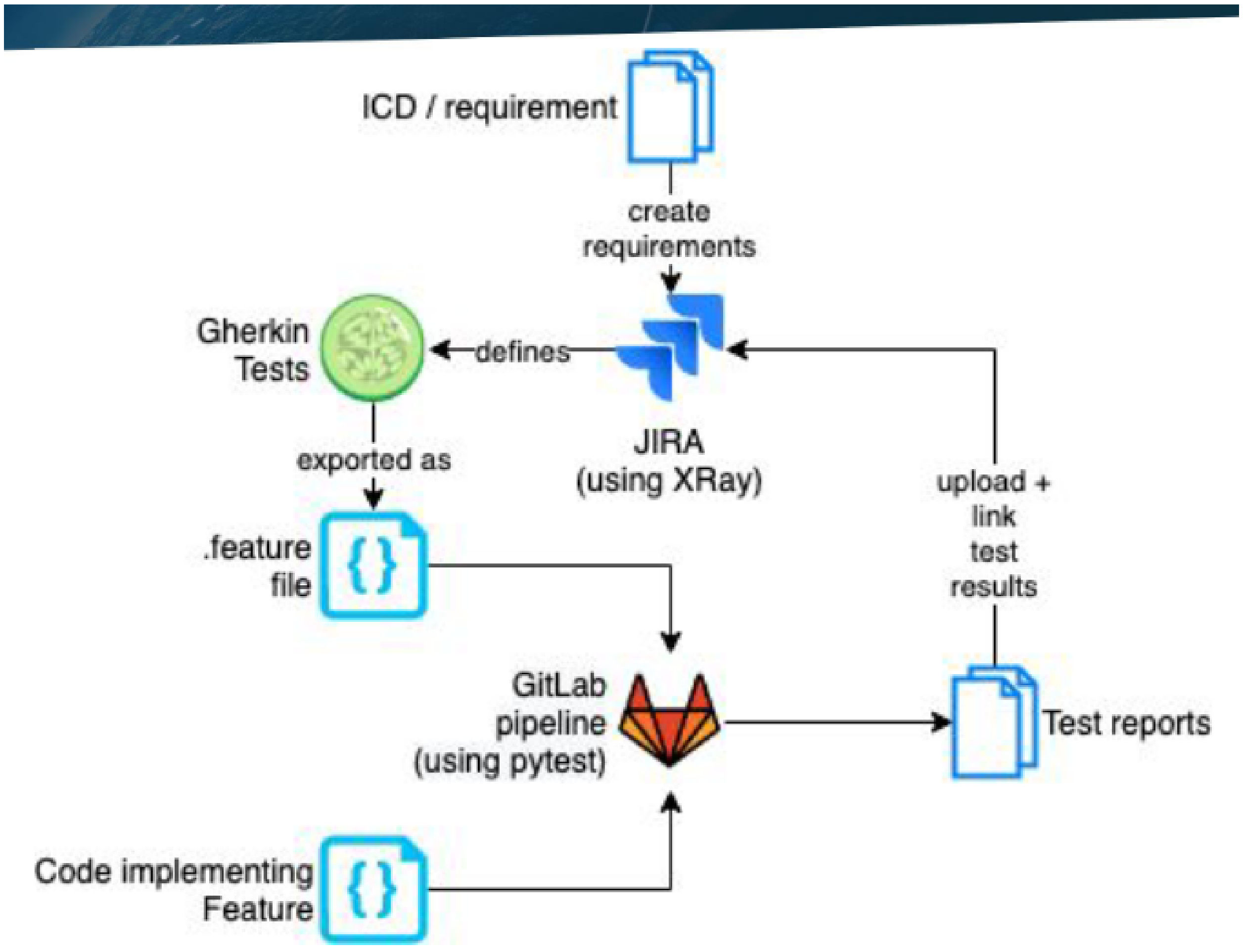
The SKA does not just need software to work. There is a vast amount of hardware to run, and the software and hardware must be integrated, as well as the various software components. The SKA has a large number of formal requirements, which specify what the telescope is and what it should do. We have L1 requirements on the whole telescope or observatory, L2 requirements on the various sub-systems, interface requirements on the interfaces between the sub-systems, and verification requirements, which help demonstrate the correct operation of the SKA telescopes.

BDD tests, with their plain language description, are based on concrete examples of the system's behaviour. Thus they can also provide living documentation of the current capabilities of the system. They allow for a good mapping between software and requirements, by allowing a clear presentation of the capabilities of the system to stakeholders.

BDD tests are automatable, and can integrate with other management systems, such as JIRA. BDD tests can help provide a traceability report, which fits in well with an engineering culture that needs to meet formal requirements. Because we can integrate BDD tests into our standard software CI/CD pipelines, we can run these tests automatically, and thus verify new features and also provide assurance that there's been no regression. We can use JIRA to collate the output of the CI/CD pipeline, and to link tests to requirements.

BDD JIRA integration

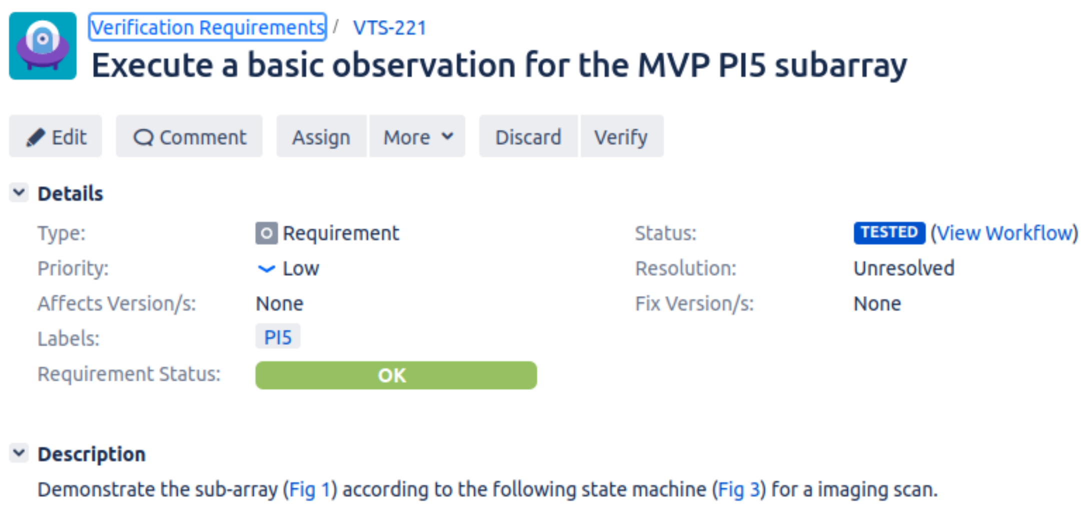
The architecture of our BDD implementation:



We work from the ICDs (Interface Control Documents) and requirements that were mostly defined in the pre-construction period. These initial requirements have been imported from the SKA JAMA system, and added to the following projects: L1, L2, IF-ID (interface requirements), and VTS (verification requirements). There's also a project for the L3 requirements, but it's not currently populated. JAMA remains the source of truth for the requirements, and we are looking at how we integrate JAMA and JIRA and how we manage new verification requirements. It is also possible to strt by creating a Feature in JIRA, and associate Tests with that Feature. PI planning is a good opportunity to consider wheter there are any requirements or features that would benefit from BDD tests.

In JIRA, we define a BDD Gherkin test (using the Given, When, Then steps) in the XTP project. We then link it to the requirement, usually via a Test Set (explained in more detail later). We export the Gherkin tests to a `.feature` file, which we store in the relevant code repository on GitLab. We write the code to implement the tests, and run the tests using the usual CI/CD pipeline, using the `pytest-bdd` extension. The CI/CD pipeline outputs the test result to JIRA, where they can be viewed on a custom Dashboard.

There are a few issues with this process at the moment. We haven't got a clear process for integrating new verification requirements with JAMA yet, nor a way of identifying which verification requirements need to go through this formal process. Exporting the `.feature` file from JIRA is currently a manual step, and we'd prefer to automate this. It's also possible to generate thousands of JIRA tickets when running tests; we plan to only keep a certain number of test runs in JIRA, but we also want to be tactical about how many tickets we generate in the first place.


Verification Requirements / VTS-221

Execute a basic observation for the MVP PI5 subarray

Details

Type:	Requirement	Status:	TESTED (View Workflow)
Priority:	Low	Resolution:	Unresolved
Affects Version/s:	None	Fix Version/s:	None
Labels:	PI5		
Requirement Status:	OK		

Description

Demonstrate the sub-array (Fig 1) according to the following state machine (Fig 3) for a imaging scan.

Fig. 6: A VTS ticket, showing the test status of the verification requirement.

Warning: We recommend only running the JIRA integration steps of the BDD testing pipeline on the main/master branch of the repository, and only on projects such as skampi that are doing a lot of integration.

This will keep the JIRA tickets under control, so we don't spam JIRA.

If you like the BDD style of testing, but don't need the formal integration with JIRA for verification, we suggest you use `pytest-bdd`, and view the test outcomes in the usual CI/CD outputs on GitLab.

Writing good BDD tests

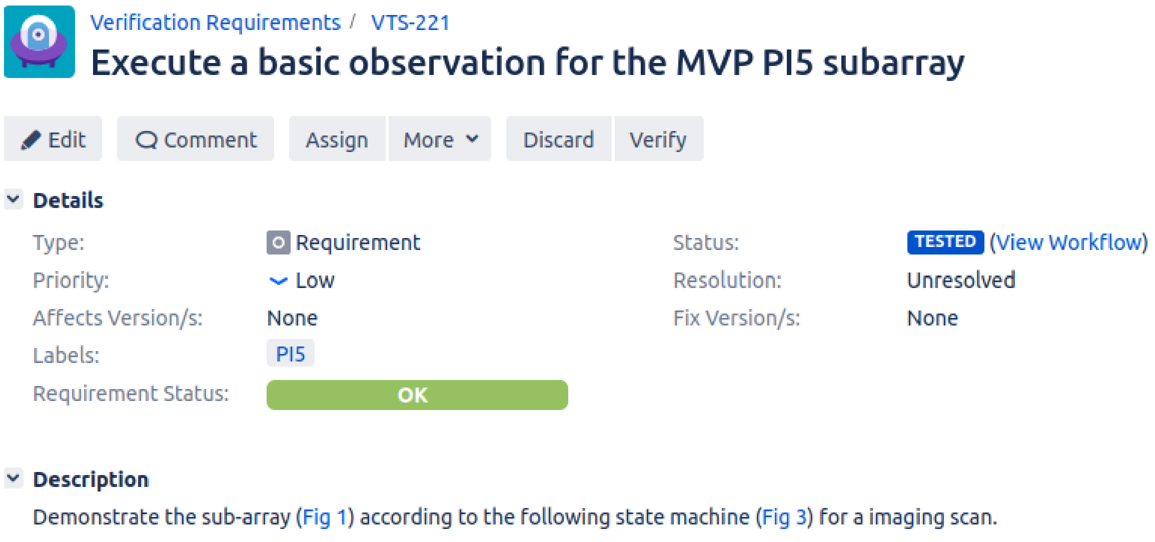
Writing good BDD tests takes practice. This section is a summary of a presentation on BDD test quality by Giorgio Brajnik. The formulation of the Gherkin steps needs domain expertise and a good understanding of the System Under Test and its potential failure modes. When implementing the test steps, you can use your existing unit and component test infrastructure. You may well be able to write most of your tests using the pytest framework (if you're working in Python, or you can exercise your code using Python bindings).

Your formulation of the Gherkin steps should be as consistent as possible. This facilitates reuse of steps. So check out the existing tests in JIRA to see what they say. This make it easier to reuse more of the underlying test infrastructure; if you have reused a test step, the underlying implementation code can be reused. The Gherkin test defintion should make it clear to stakeholders what is meant to happen to the system.

We expect that Feature and Product Owners will have the domain knowledge to write good BDD tests, with help from their verification colleagues and developers. These tests help close the loop between the formal requirements and our implementation, to verify correct implementation.

JIRA organisation for BDD tests

While we can use BDD tests with many projects, we'll look at the VTS project for verification requirements as an example. We'll use VTS-221 as an example, though we could also use a Feature ticket as a similar example. This requirement is checking whether we can configure a sub-array to perform an imaging scan.



Verification Requirements / VTS-221

Execute a basic observation for the MVP PI5 subarray

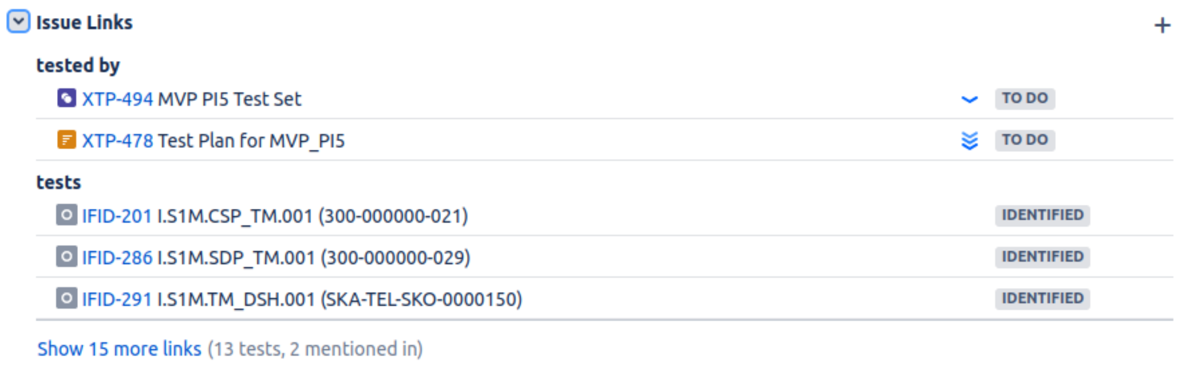
Details

Type:	Requirement	Status:	TESTED (View Workflow)
Priority:	Low	Resolution:	Unresolved
Affects Version/s:	None	Fix Version/s:	None
Labels:	PI5		
Requirement Status:	OK		

Description

Demonstrate the sub-array (Fig 1) according to the following state machine (Fig 3) for a imaging scan.

This ticket is linked to a Test Set, XTP-494, which is itself linked to four tests (XTP-417, XTP-427, XTP-436, XTP-428).



Issue Links

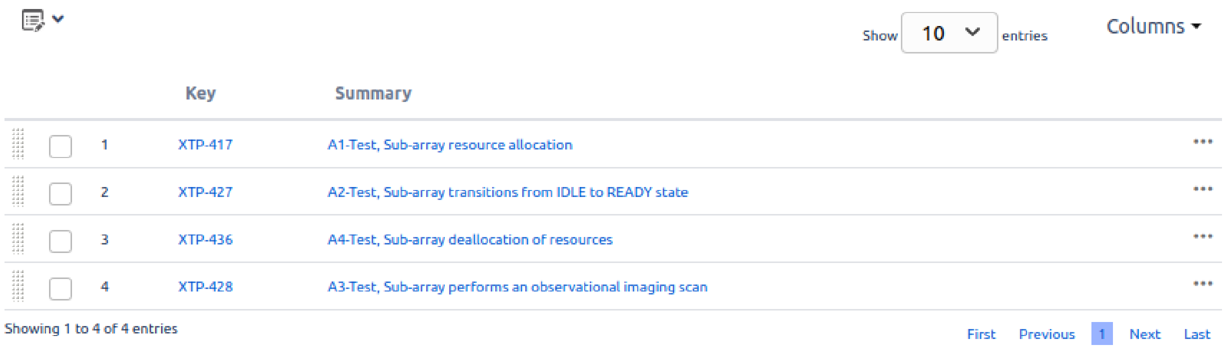
tested by

XTP-494 MVP PI5 Test Set	TO DO
XTP-478 Test Plan for MVP_PI5	TO DO

tests

IFID-201 I.S1M.CSP_TM.001 (300-000000-021)	IDENTIFIED
IFID-286 I.S1M.SDP_TM.001 (300-000000-029)	IDENTIFIED
IFID-291 I.S1M.TM_DSH.001 (SKA-TEL-SKO-0000150)	IDENTIFIED

Show 15 more links (13 tests, 2 mentioned in)



Show 10 entries Columns

	Key	Summary
1	XTP-417	A1-Test, Sub-array resource allocation
2	XTP-427	A2-Test, Sub-array transitions from IDLE to READY state
3	XTP-436	A4-Test, Sub-array deallocation of resources
4	XTP-428	A3-Test, Sub-array performs an observational imaging scan

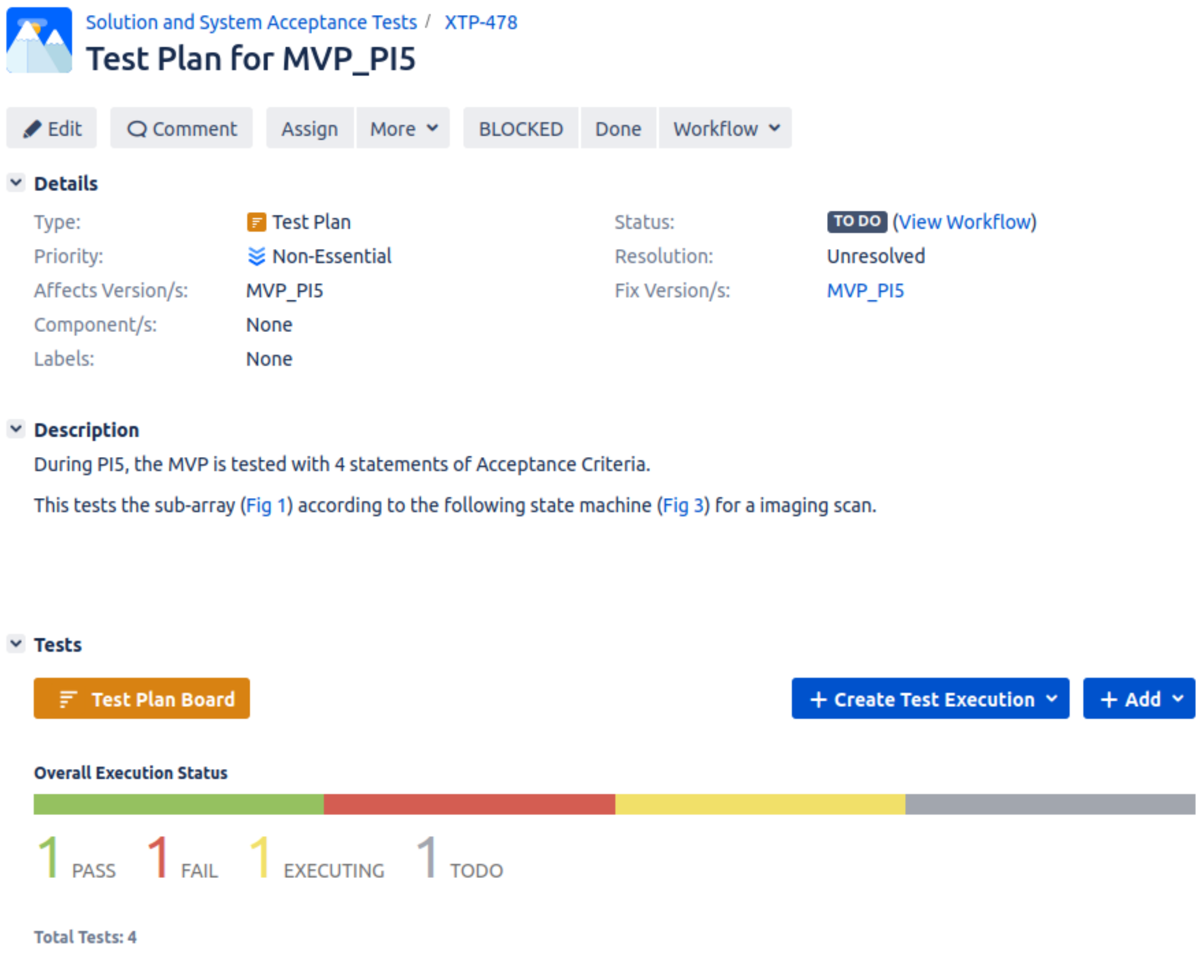
Showing 1 to 4 of 4 entries

First Previous 1 Next Last

It is from the Test Set XTP-494 that we export the `.feature` file, for adding to our repository. It is the Test Set that defines the collection - the set of tests - needed to verify VTS-221. The individual tests linked to the Test Set can be reused, and linked to other Test Sets. Similarly, we can associate multiple Test Sets to a requirement, and the same Test Set can be associated with multiple requirements.

Test Sets allow us to gather tests exercising different parts of the requirement - for example, happy and sad paths through the feature. Or a requirement may say the telescope should either point to a new target, or declare an error state. The error state Test may be reused across many Test Sets. The Test Sets allow us to group tests together logically, and to reuse lower level tests efficiently.

We can create Test Plans, such as XTP-478, which can be associated with Test Sets. Test Plans are most usefully used for manual test executions. We can describe test conditions in Test Plans, which is most useful for people configuring the test environment by hand; they are less useful for fully automated tests that happen as part of the CI/CD pipeline. A Test Execution JIRA ticket can then be created to record the results.



Test Plan for MVP_PI5

Details

Type:	Test Plan	Status:	TO DO (View Workflow)
Priority:	Non-Essential	Resolution:	Unresolved
Affects Version/s:	MVP_PI5	Fix Version/s:	MVP_PI5
Component/s:	None		
Labels:	None		

Description

During PI5, the MVP is tested with 4 statements of Acceptance Criteria.

This tests the sub-array (Fig 1) according to the following state machine (Fig 3) for a imaging scan.

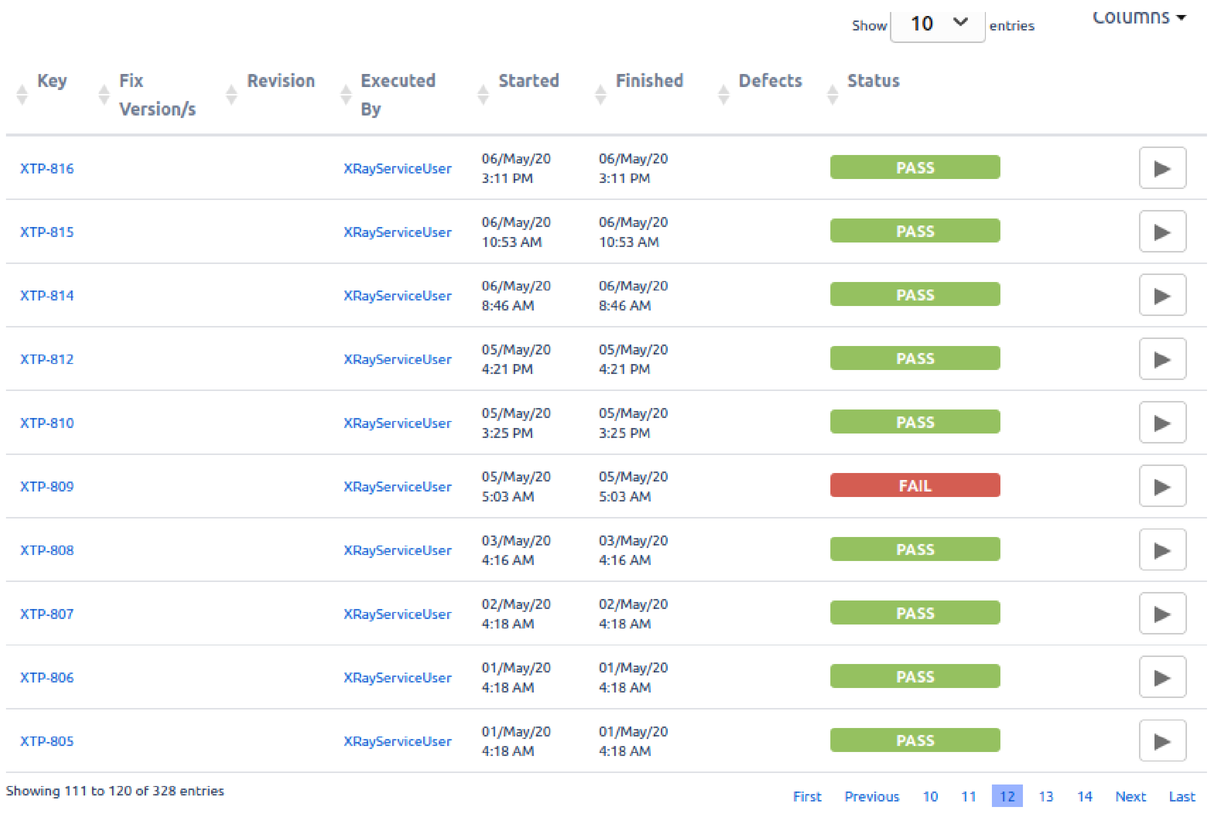
Tests

Overall Execution Status

1 PASS 1 FAIL 1 EXECUTING 1 TODO

Total Tests: 4

On the individual test tickets, such as XTP-436, we can see when the test was last run, and whether it passed or failed. You'll see that each time the test is run, a Test Execution ticket is generated, reported by the XrayServiceUser, which shows the test was carried out as part of the CI/CD pipeline. This can generate thousands of tickets very quickly; hence our recommendation to only run this on the main branch of the repository.



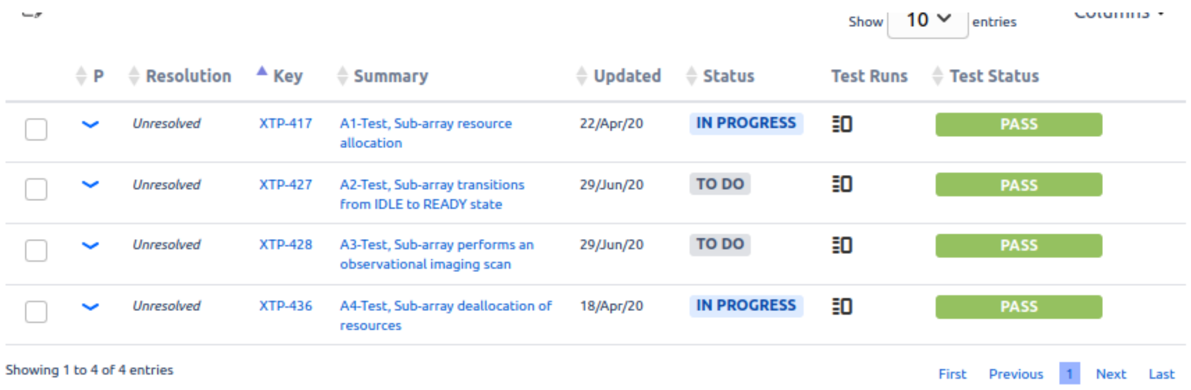
Show 10 entries Columns

Key	Fix Version/s	Revision	Executed By	Started	Finished	Defects	Status
XTP-816			XRayServiceUser	06/May/20 3:11 PM	06/May/20 3:11 PM		PASS
XTP-815			XRayServiceUser	06/May/20 10:53 AM	06/May/20 10:53 AM		PASS
XTP-814			XRayServiceUser	06/May/20 8:46 AM	06/May/20 8:46 AM		PASS
XTP-812			XRayServiceUser	05/May/20 4:21 PM	05/May/20 4:21 PM		PASS
XTP-810			XRayServiceUser	05/May/20 3:25 PM	05/May/20 3:25 PM		PASS
XTP-809			XRayServiceUser	05/May/20 5:03 AM	05/May/20 5:03 AM		FAIL
XTP-808			XRayServiceUser	03/May/20 4:16 AM	03/May/20 4:16 AM		PASS
XTP-807			XRayServiceUser	02/May/20 4:18 AM	02/May/20 4:18 AM		PASS
XTP-806			XRayServiceUser	01/May/20 4:18 AM	01/May/20 4:18 AM		PASS
XTP-805			XRayServiceUser	01/May/20 4:18 AM	01/May/20 4:18 AM		PASS

Showing 111 to 120 of 328 entries

First Previous 10 11 12 13 14 Next Last

These results are what are passed to the requirements ticket, such as VTS-221, and to any Dashboards that are configured.



Show 10 entries Columns

	P	Resolution	Key	Summary	Updated	Status	Test Runs	Test Status
☐		Unresolved	XTP-417	A1-Test, Sub-array resource allocation	22/Apr/20	IN PROGRESS		PASS
☐		Unresolved	XTP-427	A2-Test, Sub-array transitions from IDLE to READY state	29/Jun/20	TO DO		PASS
☐		Unresolved	XTP-428	A3-Test, Sub-array performs an observational imaging scan	29/Jun/20	TO DO		PASS
☐		Unresolved	XTP-436	A4-Test, Sub-array deallocation of resources	18/Apr/20	IN PROGRESS		PASS

Showing 1 to 4 of 4 entries

First Previous 1 Next Last

CI/CD integration

The actual tests are defined using an extension to `pytest` - `pytest-bdd`. This plugin runs the tests, and uses the `.feature` file plus annotations to output a `.json` file as part of the CI/CD pipeline. A post-test step in the `.gitlab-ci.yml` file for the repository pushes the `.json` file which contains the ticket metadata to JIRA, where it's parsed by the XRay JIRA plugin. XRay then creates Test Execution tickets, and updates the statuses of the Test and Requirements tickets to show the test result. JIRA statuses can be used in the usual way to build dashboards and other reports on the status of requirements. Failures are reported as well as successes, provided the CI/CD pipeline completes.

A feature file can be very simple:

```
@XTP-1156
Scenario: Observation Execution Tool
  Given The Observation Execution Tool create command
  When OET create is given a <file> that does not exist
  Then the OET returns an <error>

  Examples:
| file                        | error                                               ␣
↪           |
| file:///FileNotFound.py     | FileNotFoundError: No such file or directory: /
↪FileNotFound.py   |
| sdljfsdjkfhsd               | ValueError: Script URI type not handled: sdljfsdjkfhsd␣
↪           |
```

This file refers to a single test, XTP-1156. Additional annotations can provide the Test Set JIRA ticket number. The Examples table allows for handling multiple inputs for checking, without having to write lots of When and Then steps.

The associated test code then can be relatively simple:

```
import pytest
import pytest-bdd

@pytest.fixture
def result():
    return {}

@scenario("XTP-1156.feature", "Observation Execution Tool")
def test_create(:
    pass

@given('the Observation Execution Tools create command')
def command():
    // code to issue the command goes here

@when('OET create is given a <file> that does not exist')
def output_from_junk_file(file):
   // code to feed the tool a non-existent file

@then('the OET returns an <error>')
def return_error(file, error):
   // code to return the correct error goes here
```

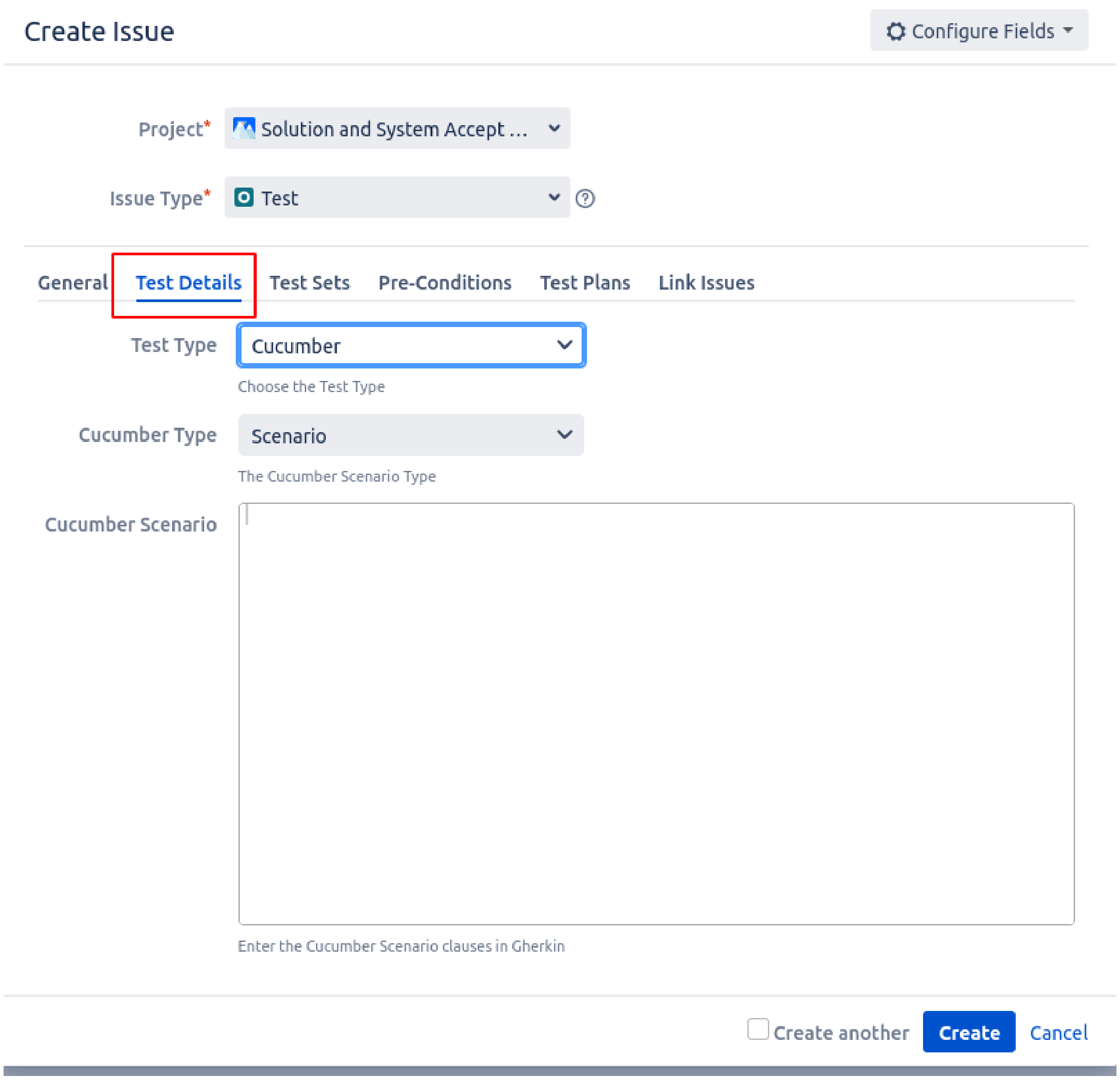
This imports the relevant pytest infrastructure, creates a pytest fixture to allow information to be passed between steps, and then annotates the pytest infrastructure to allow the JIRA metadata to be associated with the outputs. It also includes the test description, so you can see what the code is meant to be doing. The given, when, then steps and associated methods can be reused for other tests if this is useful.

This code is lightly adapted from code in https://gitlab.com/ska-telescope/skampi/-/tree/master/post-deployment.

2.9 BDD Walkthrough

This is a small adaptation of material by Giorgio Brajnik. For background on how BDD tests work, please read *BDD testing guide*.

- Identify an SKA requirement or Feature. This may be an existing requirement, such as an interface requirement, or, if you are working on system verification, you may create a new requirement, such as VTS-221.
  - If writing a new requirement, please label it with the PI (Program Increment) in which you plan to implement it.
- Create a JIRA issue of type "Test Set" in the XTP project.
  - (optional) Add a fix version corresponding to the relevant PI.
  - Link this issue to the requirement or Feature defined above, using the "tests" relationship. (This can also be done from the requirement/Feature, but then the relationship used should be "tested by".) This can be done from the Test Set Create screen using the "link" field.
- Create the tests for the Test Set.
  - Create issue of type "Test" in the XTP project.
  - (optional) Add fix version.
  - Click on the "Test Details" tab in the newly-created issue


Create Issue

Project* Solution and System Accept ...

Issue Type* Test

Test Type: Cucumber

Choose the Test Type

Cucumber Type: Scenario

The Cucumber Scenario Type

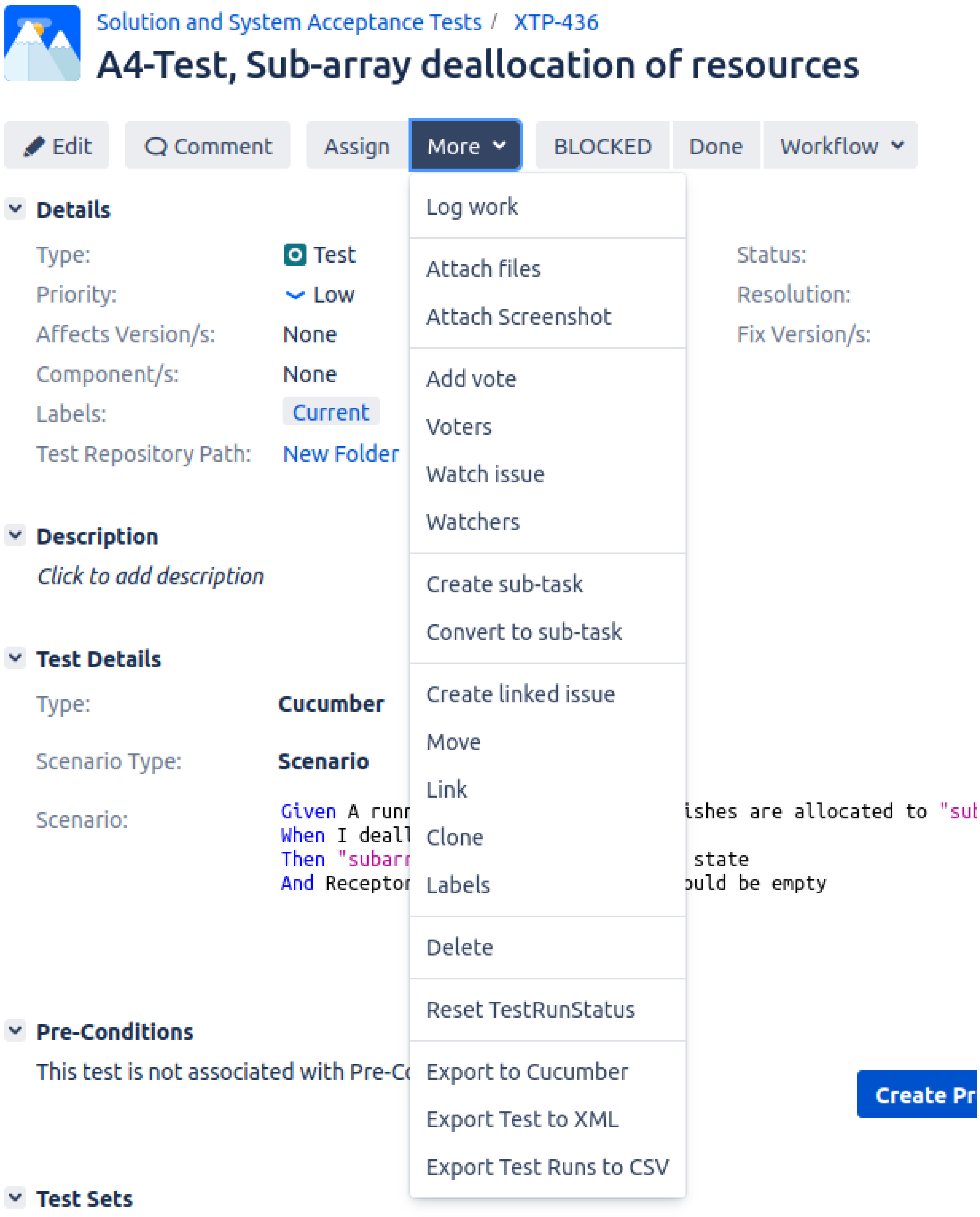
Cucumber Scenario

Enter the Cucumber Scenario clauses in Gherkin

Then provide the test details:

- Test type: Cucumber
- Cucumber type: scenario
- Cucmber scenario: write your Gherkin (given, when, then) steps here.
- Link your test to the relevent Test Set or Test Sets. If you wish to link an existing test to a new Test Set, that's encouraged, and you can skip the test creation steps.

- **Once all the tests for the Test Set are defined, you can export the `.feature` file.**
  - Find the relevant Test Set.
  - Go to the More dropdown menu.
  - Select Export to Cucumber from the menu. You'll need to do this for each Test Set you wish to exercise.


Solution and System Acceptance Tests / XTP-436

A4-Test, Sub-array deallocation of resources

Details

Type:	Test
Priority:	Low
Affects Version/s:	None
Component/s:	None
Labels:	Current
Test Repository Path:	New Folder
Status:	
Resolution:	
Fix Version/s:	

Description

Click to add description

Test Details

Type:	**Cucumber**
Scenario Type:	**Scenario**

Scenario:

```
Given A runr                  ishes are allocated to "sul
When I deall
Then "subarr                   state
And Receptor                  ould be empty
```

Pre-Conditions

This test is not associated with Pre-C

Test Sets

- Add the `.feature` file to the relevant GitLab repository. We recommend placing this in the same directory as your tests; you may want to create a directory for your .feature files so that they are placed close to the test code, but so they're not confused with it.
- Implement your tests using `pytest-bdd`.

- Import `pytest-bdd` to your test module.
- Define a pytest fixture. This creates an empty dictionary that is used to communicate data between steps.
- Annotate the test case with the relevant scenario.
- Write your tests, annotating the methods with the Gherkin keywords. These methods can be reused by your tests (e.g. the same "given" step can be reused by several tests).

```
// import the relevant libraries
from pytest-bdd import (given, parsers, scenarios, then, when)

// load the scenarios from the .feature file. If there are multiple scenarios, add␣
↪the scenario name after the path.
scenarios(path/to/.feature/file)

//you can create a pytest fixture to allow you to pass data between steps via a␣
↪dictionary
@pytest.fixture
def result():
   return {}

//then write your test steps, annotating them appropriately:
@given('I have an SDPSubarry device')
def subarray_device(devices):
   //code to get a subarray device
   result = devices.get_device(DEVICE_NAME)
   return result

// note that this given step can be reused for many tests that need an SDP subarray␣
↪device.

@when('Test step goes here')
def set_device_state(device):
    // more test code goes here

@then('Result step goes here')
def test_result():
    // test the result of your when steps here
```

This code is loosely based on https://gitlab.com/ska-telescope/sdp/ska-sdp-lmc/-/blob/master/tests/test_subarray.py.

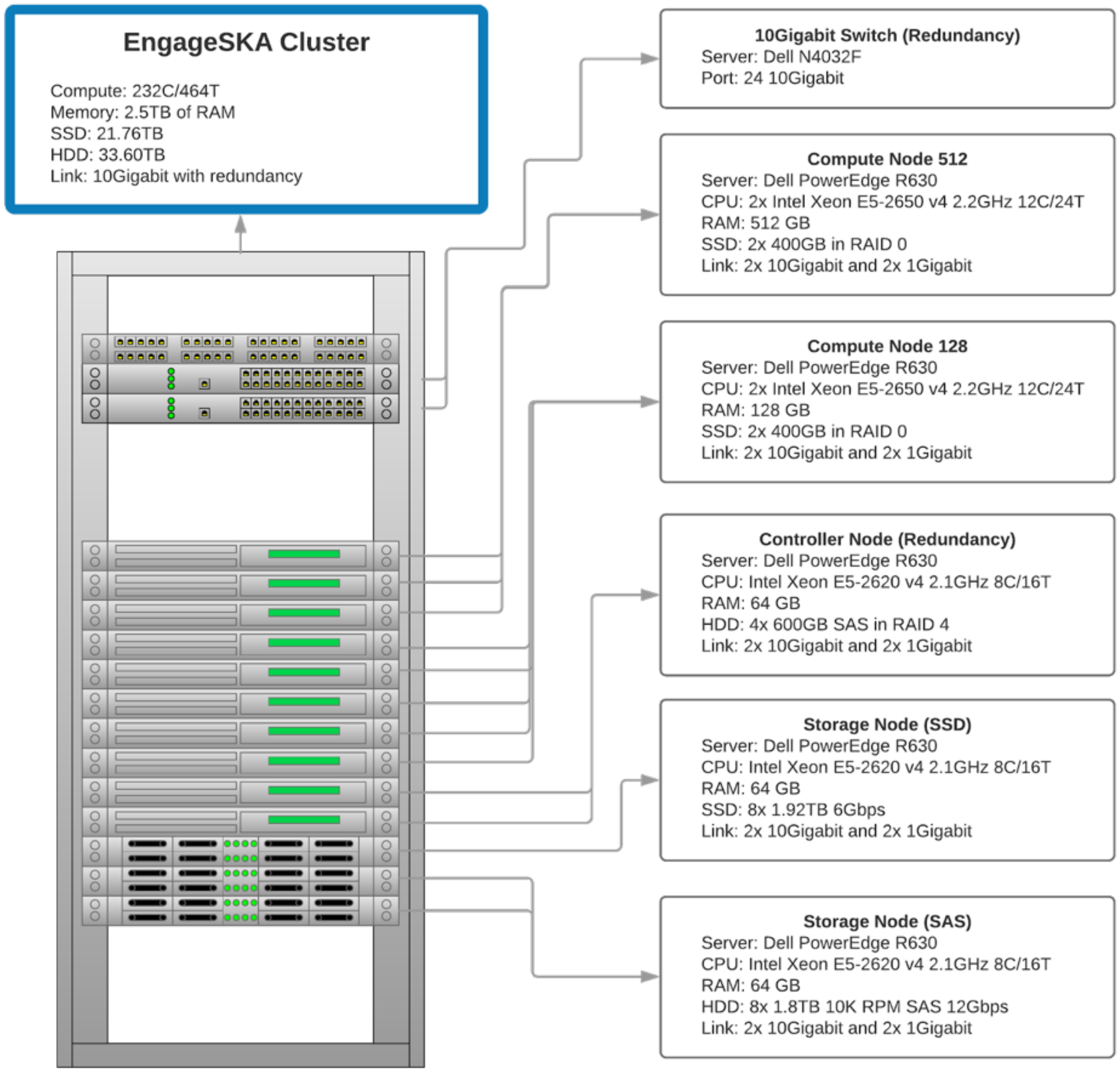
Note: We strongly recommend only using the JIRA integration on repositories such as skampi, that do a lot of integration. We further recommend only using the JIRA integration on the main/master branch. If you like the BDD testing style, you can just use `pytest-bdd` and get test outcomes as part of the usual CI/CD pipeline.

2.10 EngageSKA: CI/CD & Testing Infrastructure

Our CI/CD takes place on the EngageSKA cluster. Its architecture and instructions on how to access it are described in *EngageSKA cluster*. *Monitoring Dashboards* to help you discover the status of various projects are also available.

2.10.1 EngageSKA cluster

Cluster specs


EngageSKA Cluster
Compute: 232C/464T
Memory: 2.5TB of RAM
SSD: 21.76TB
HDD: 33.60TB
Link: 10Gigabit with redundancy
10Gigabit Switch (Redundancy)
Server: Dell N4032F
Port: 24 10Gigabit
Compute Node 512
Server: Dell PowerEdge R630
CPU: 2x Intel Xeon E5-2650 v4 2.2GHz 12C/24T
RAM: 512 GB
SSD: 2x 400GB in RAID 0
Link: 2x 10Gigabit and 2x 1Gigabit
Compute Node 128
Server: Dell PowerEdge R630
CPU: 2x Intel Xeon E5-2650 v4 2.2GHz 12C/24T
RAM: 128 GB
SSD: 2x 400GB in RAID 0
Link: 2x 10Gigabit and 2x 1Gigabit
Controller Node (Redundancy)
Server: Dell PowerEdge R630
CPU: Intel Xeon E5-2620 v4 2.1GHz 8C/16T
RAM: 64 GB
HDD: 4x 600GB SAS in RAID 4
Link: 2x 10Gigabit and 2x 1Gigabit
Storage Node (SSD)
Server: Dell PowerEdge R630
CPU: Intel Xeon E5-2620 v4 2.1GHz 8C/16T
RAM: 64 GB
SSD: 8x 1.92TB 6Gbps
Link: 2x 10Gigabit and 2x 1Gigabit
Storage Node (SAS)
Server: Dell PowerEdge R630
CPU: Intel Xeon E5-2620 v4 2.1GHz 8C/16T
RAM: 64 GB
HDD: 8x 1.8TB 10K RPM SAS 12Gbps
Link: 2x 10Gigabit and 2x 1Gigabit

Access the cluster

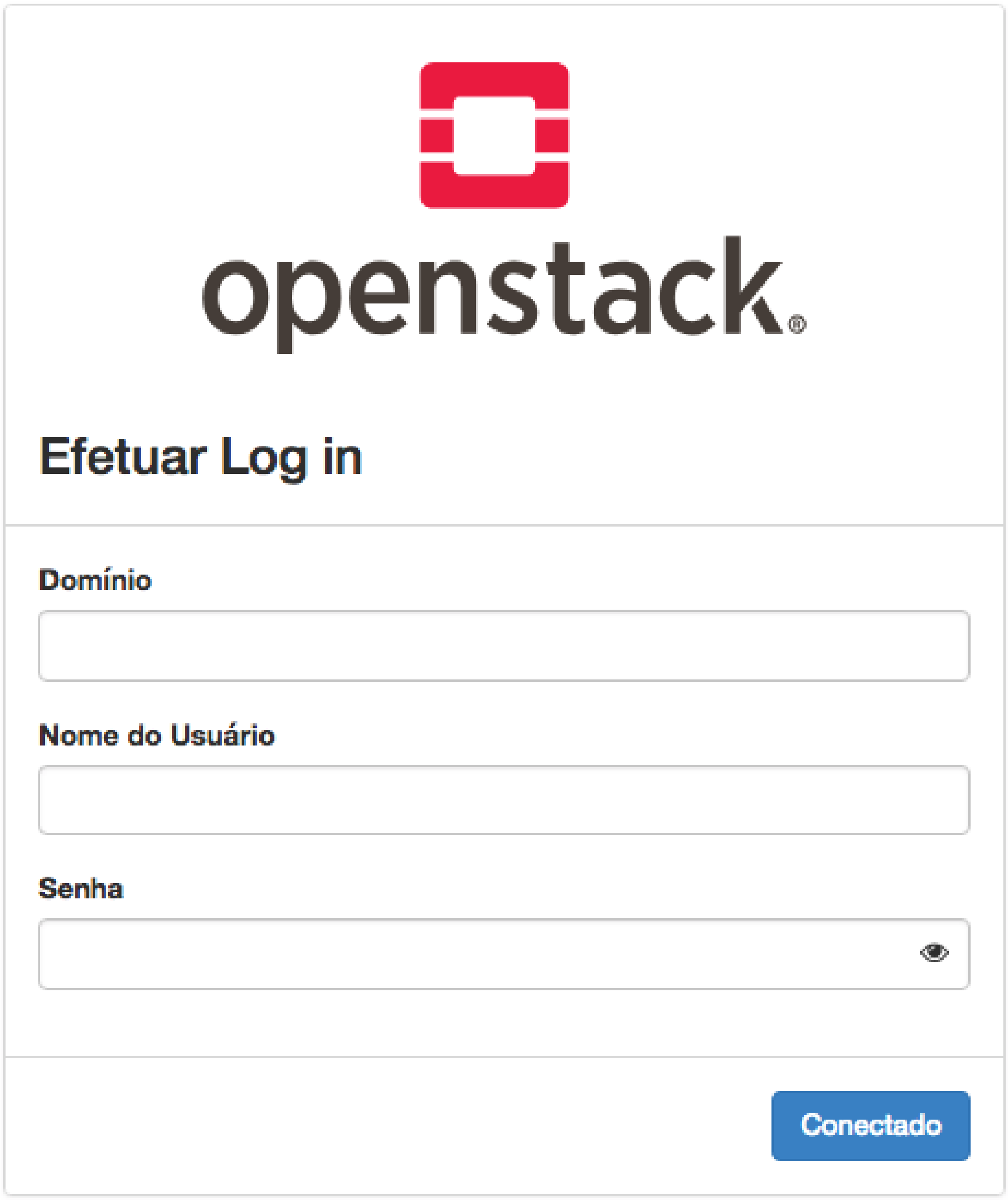
The EngageSKA cluster locates at the Datacenter of Institute of Telecommunication (IT) in Aveiro. To have access to the cluster, it is required to be inside the facilities or have VPN credentials to access the IT network remotely.

Access to the network using VPN

At the moment, VPN credentials are sent individually and is required to send an email to Domingos Nunes (dfsn@ua.pt) with the knowledge from Piers Harding (P.Harding@skatelescope.org).

- Guide on how to connect to the private network using VPN: https://engageska-portugal.pt/theme/files/vpn-guide.pdf

Access to the OpenStack platform (Virtualization)



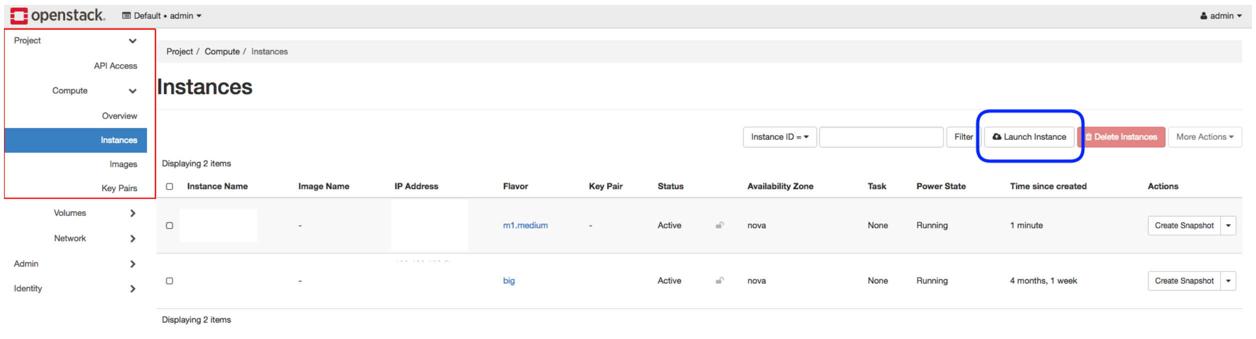
The OpenStack platform requires authentication in order to use it.

At the moment, OpenStack credentials are sent individually and it is required to send an email to Domingos Nunes (dfsn@ua.pt) with the knowledge from Piers Harding (piers@ompka.net). In the next phase, OpenStack could support Gitlab authentication.

To access the OpenStack platform go to http://192.168.93.215/dashboard (requires VPN) and login with your credentials. These credentials should be the same used for your VPN authentication. The **Domain** to use is **default**.

Virtual machine deployment

- At the sidebar, go to Project -> Compute -> Instances and click on the "Launch Instance" button:



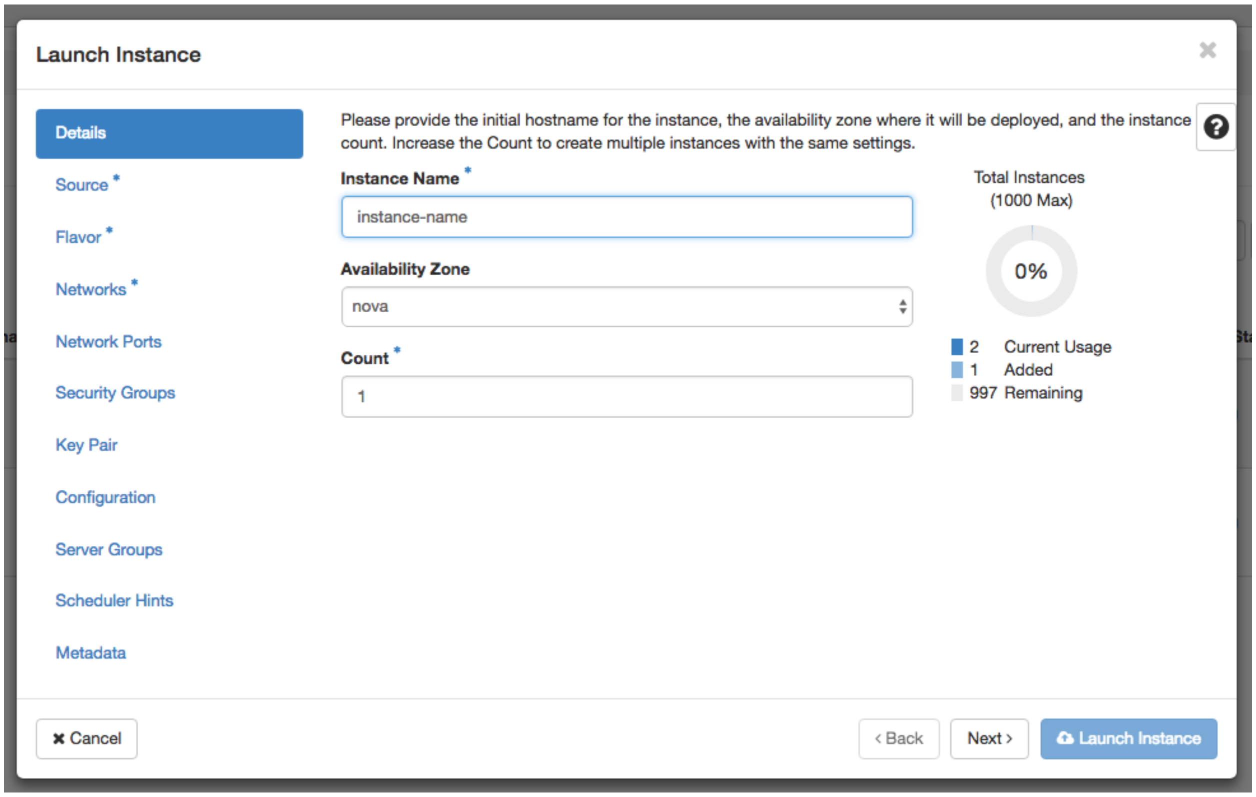
Instances

Displaying 2 items

	Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Time since created	Actions
☐	[redacted]	-	[redacted]	m1.medium	-	Active	nova	None	Running	1 minute	Create Snapshot
☐	[redacted]	-	[redacted]	big		Active	nova	None	Running	4 months, 1 week	Create Snapshot

Displaying 2 items

- At this stage a menu will pop-up and will ask to specify virtual machine characteristics, chose an name for virtual machine:



Please provide the initial hostname for the instance, the availability zone where it will be deployed, and the instance count. Increase the Count to create multiple instances with the same settings.

Instance Name *: instance-name

Availability Zone: nova

Count *: 1

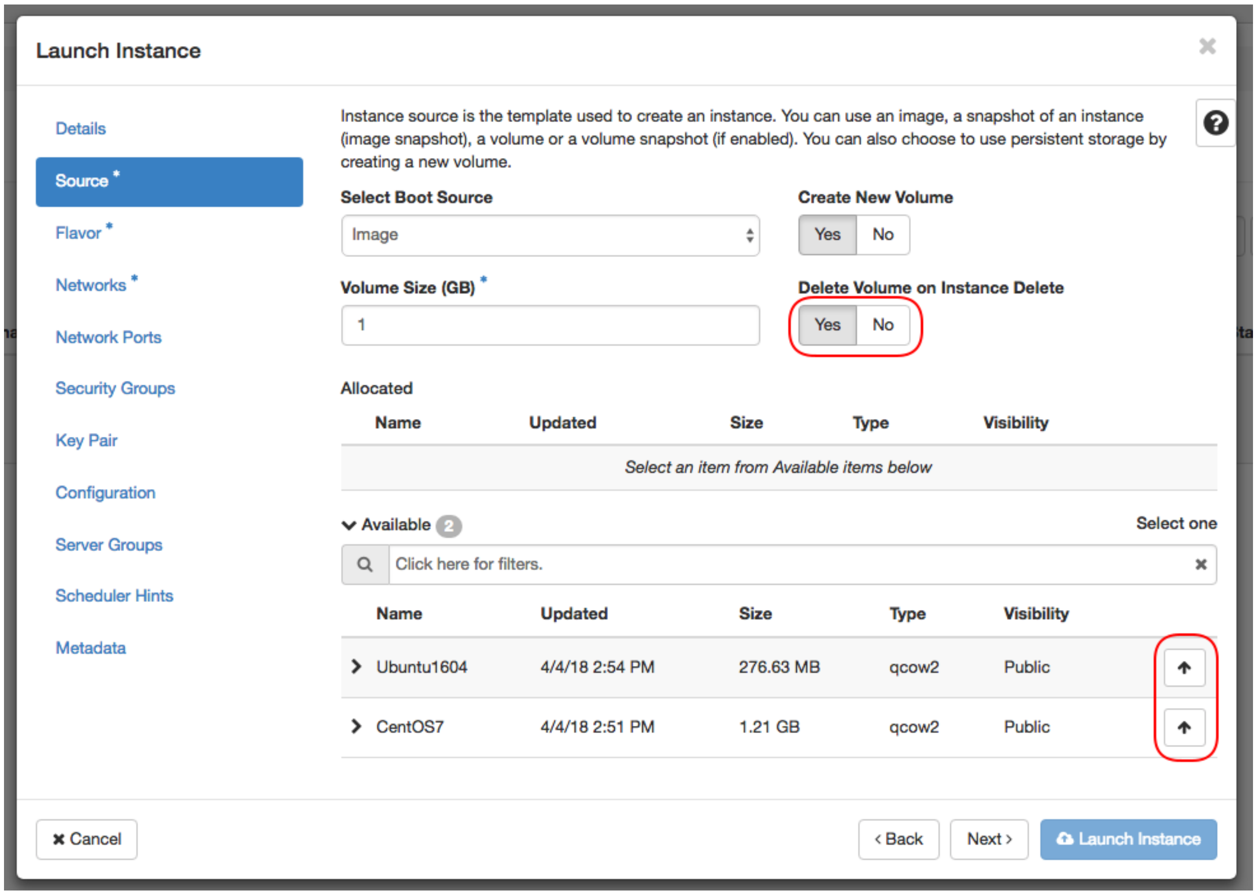
Total Instances (1000 Max)

0%

■	2	Current Usage
■	1	Added
□	997	Remaining

- Select the Operating System you want your VM to have:

NOTE: Please choose the option "Yes" at "Delete Volume on Instance Delete" so when you decide to delete the instance the volume will be also deleted and not occupy unnecessary space



Launch Instance

Details
Source *
Flavor *
Networks *
Network Ports
Security Groups
Key Pair
Configuration
Server Groups
Scheduler Hints
Metadata

Instance source is the template used to create an instance. You can use an image, a snapshot of an instance (image snapshot), a volume or a volume snapshot (if enabled). You can also choose to use persistent storage by creating a new volume.

Select Boot Source: Image

Create New Volume: [✓ Yes] No

Volume Size (GB) *: 1

Delete Volume on Instance Delete: [✓ Yes] No

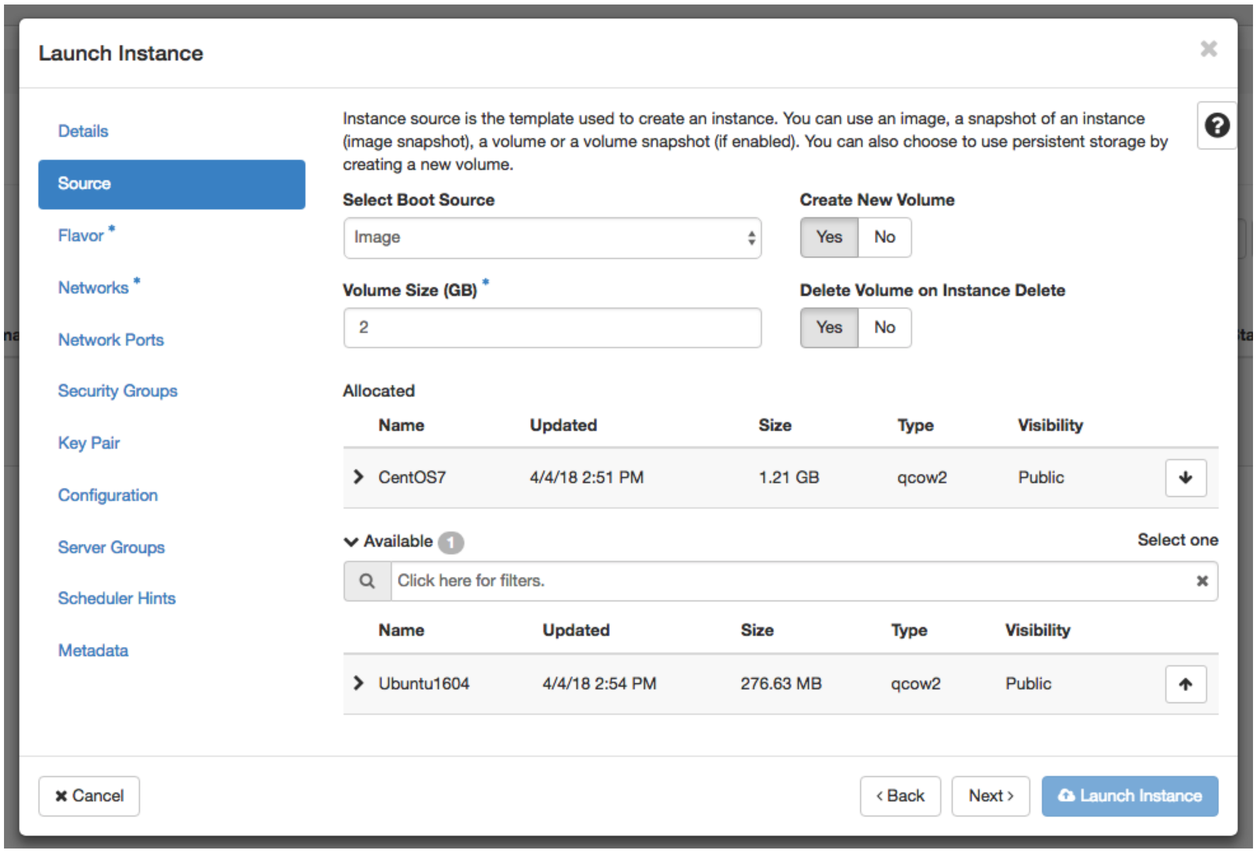
Allocated

Name	Updated	Size	Type	Visibility
Select an item from Available items below				

Available 2 — Select one

Click here for filters.

Name	Updated	Size	Type	Visibility
Ubuntu1604	4/4/18 2:54 PM	276.63 MB	qcow2	Public
CentOS7	4/4/18 2:51 PM	1.21 GB	qcow2	Public



Instance source is the template used to create an instance. You can use an image, a snapshot of an instance (image snapshot), a volume or a volume snapshot (if enabled). You can also choose to use persistent storage by creating a new volume.

Select Boot Source: Image

Create New Volume: [✓ Yes] No

Volume Size (GB) *: 2

Delete Volume on Instance Delete: [✓ Yes] No

Allocated

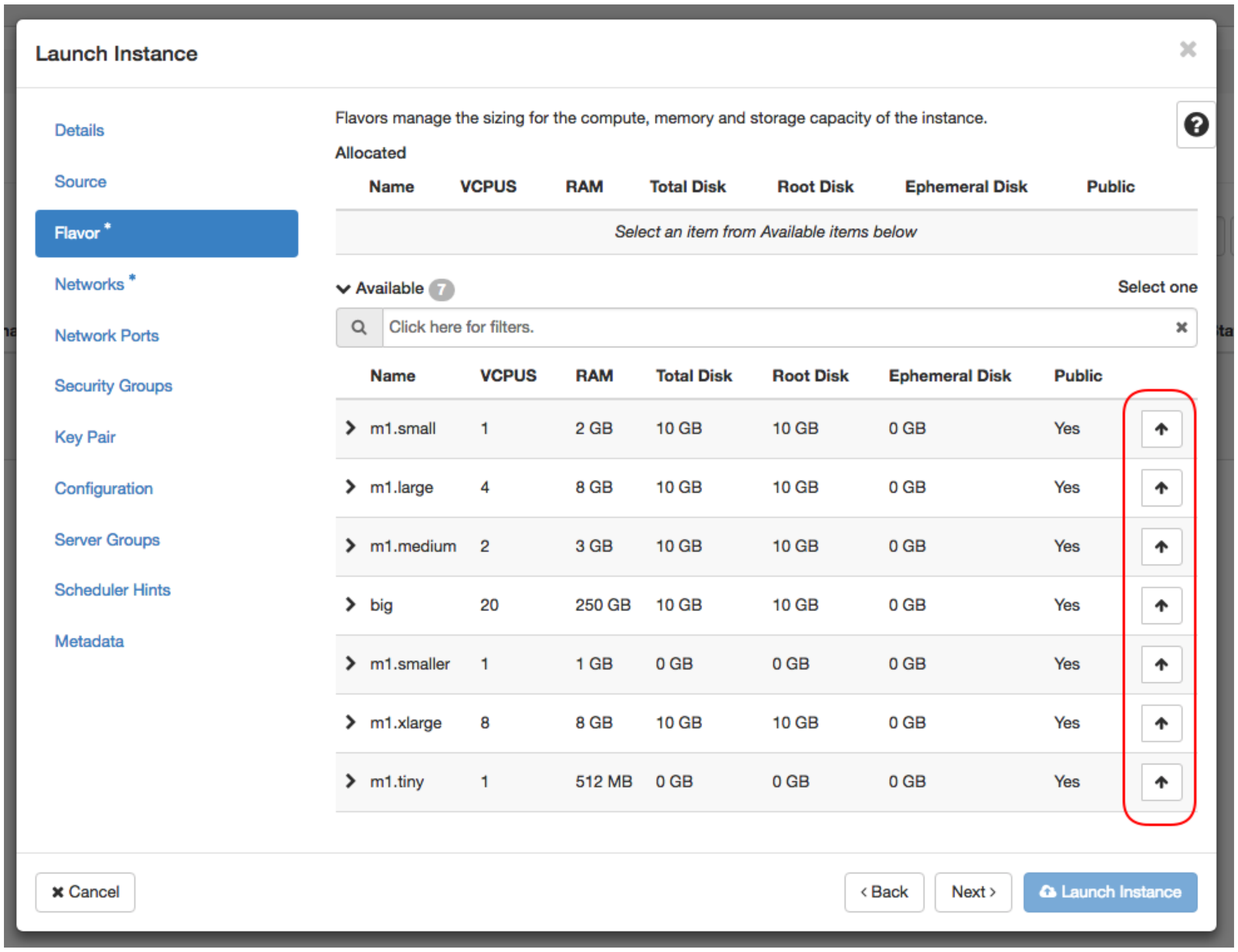
Name	Updated	Size	Type	Visibility
CentOS7	4/4/18 2:51 PM	1.21 GB	qcow2	Public

Available 1 — Select one

Click here for filters.

Name	Updated	Size	Type	Visibility
Ubuntu1604	4/4/18 2:54 PM	276.63 MB	qcow2	Public

- Select the flavor which you want your VM to have:



Flavors manage the sizing for the compute, memory and storage capacity of the instance.

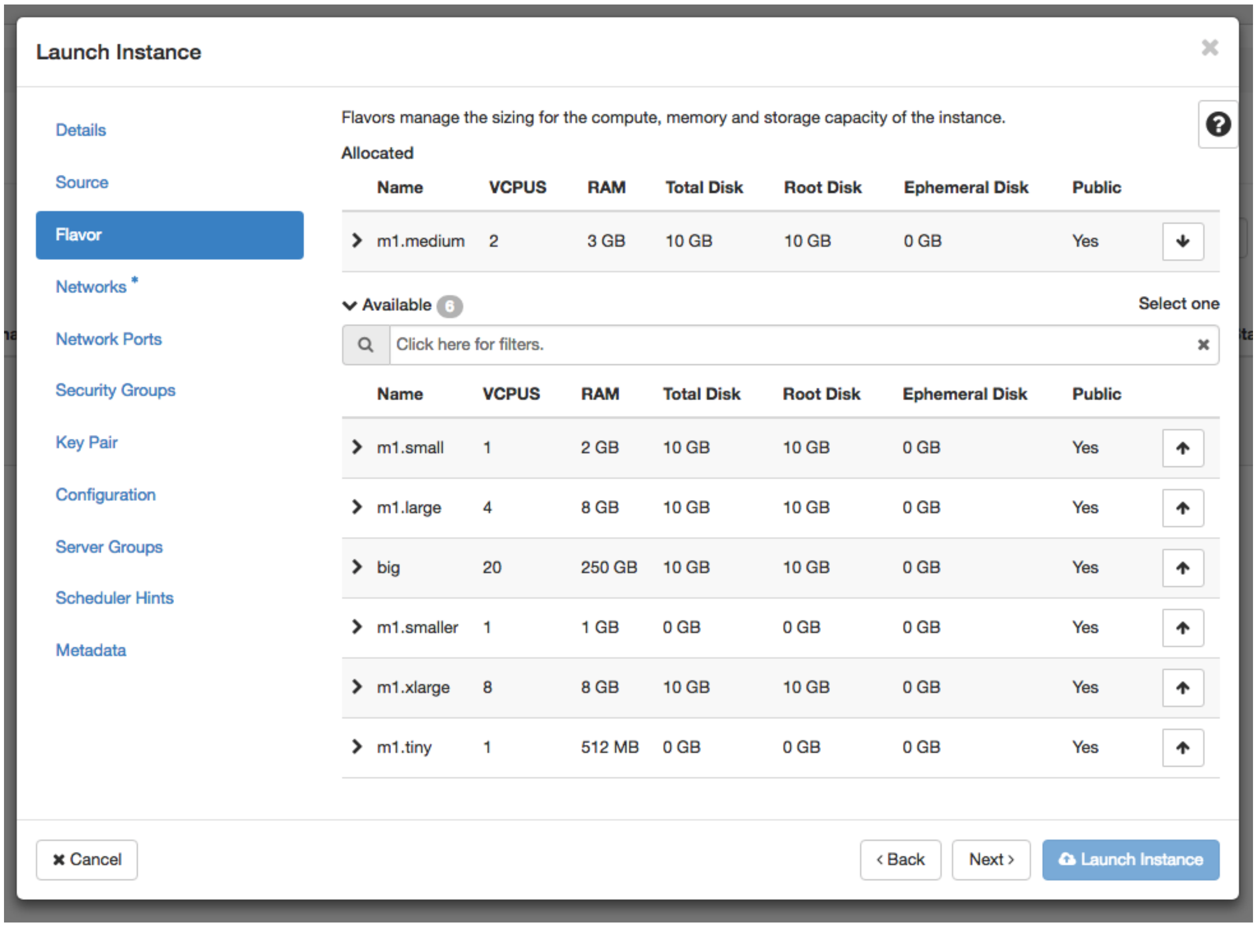
Allocated

Name	VCPUS	RAM	Total Disk	Root Disk	Ephemeral Disk	Public
Select an item from Available items below						

Available 7 — Select one

Click here for filters.

Name	VCPUS	RAM	Total Disk	Root Disk	Ephemeral Disk	Public
m1.small	1	2 GB	10 GB	10 GB	0 GB	Yes
m1.large	4	8 GB	10 GB	10 GB	0 GB	Yes
m1.medium	2	3 GB	10 GB	10 GB	0 GB	Yes
big	20	250 GB	10 GB	10 GB	0 GB	Yes
m1.smaller	1	1 GB	0 GB	0 GB	0 GB	Yes
m1.xlarge	8	8 GB	10 GB	10 GB	0 GB	Yes
m1.tiny	1	512 MB	0 GB	0 GB	0 GB	Yes



Flavors manage the sizing for the compute, memory and storage capacity of the instance.

Allocated

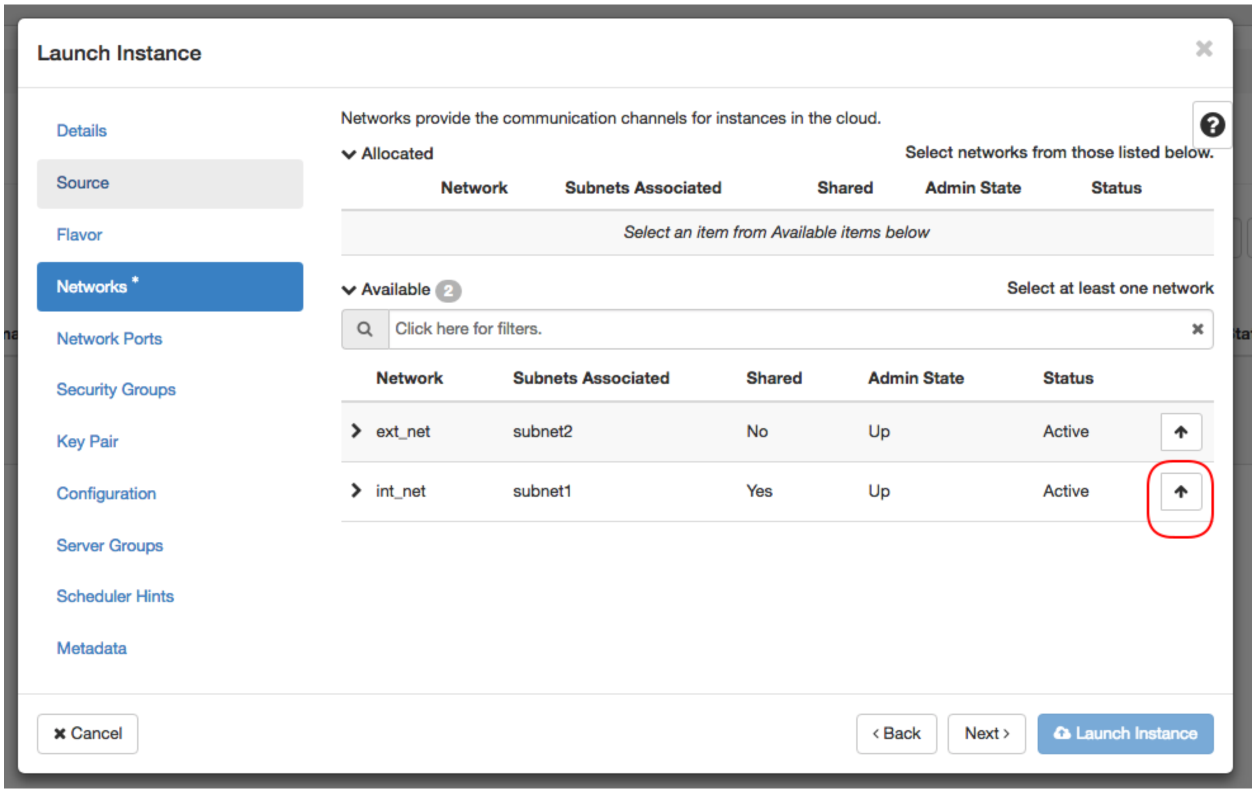
Name	VCPUS	RAM	Total Disk	Root Disk	Ephemeral Disk	Public
m1.medium	2	3 GB	10 GB	10 GB	0 GB	Yes

Available 6 Select one

Click here for filters.

Name	VCPUS	RAM	Total Disk	Root Disk	Ephemeral Disk	Public
m1.small	1	2 GB	10 GB	10 GB	0 GB	Yes
m1.large	4	8 GB	10 GB	10 GB	0 GB	Yes
big	20	250 GB	10 GB	10 GB	0 GB	Yes
m1.smaller	1	1 GB	0 GB	0 GB	0 GB	Yes
m1.xlarge	8	8 GB	10 GB	10 GB	0 GB	Yes
m1.tiny	1	512 MB	0 GB	0 GB	0 GB	Yes

- Select private network (int-net):



Networks provide the communication channels for instances in the cloud.

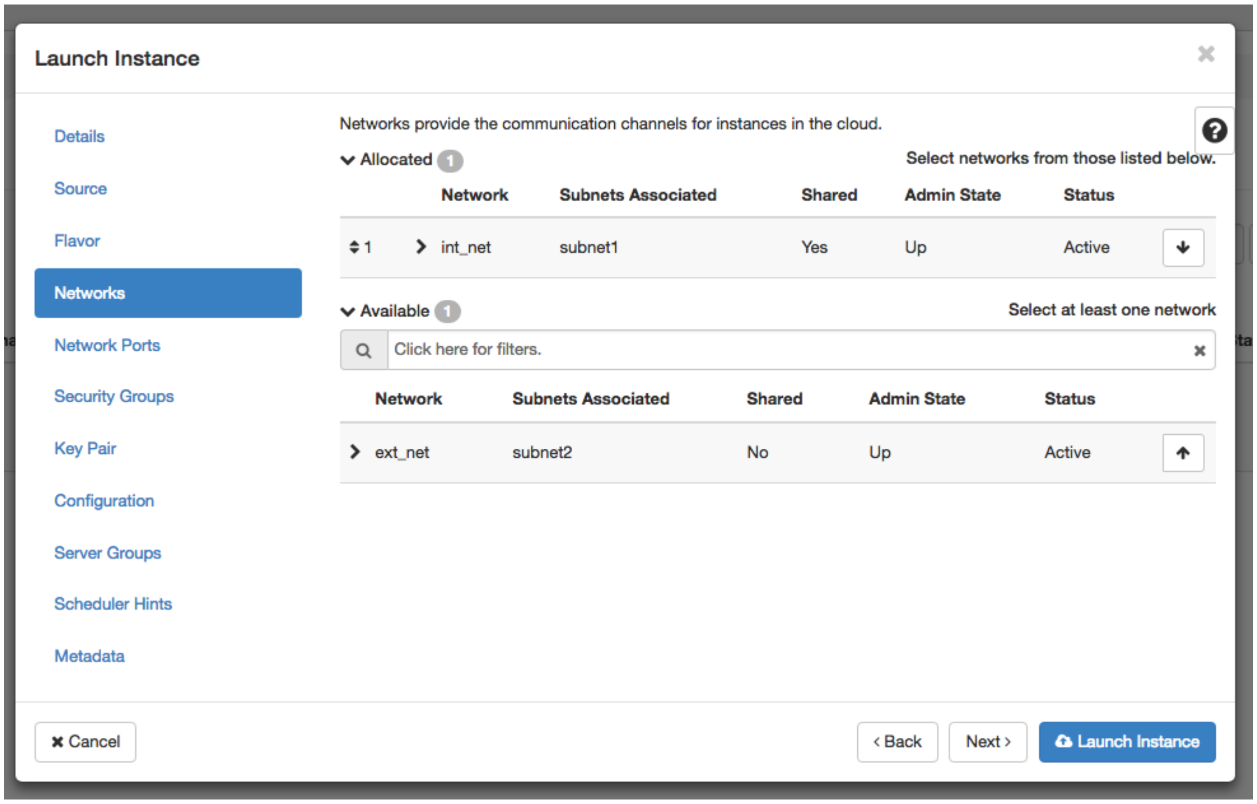
Allocated — Select networks from those listed below.

Network	Subnets Associated	Shared	Admin State	Status
Select an item from Available items below				

Available 2 — Select at least one network

Click here for filters.

Network	Subnets Associated	Shared	Admin State	Status
ext_net	subnet2	No	Up	Active
int_net	subnet1	Yes	Up	Active



Networks provide the communication channels for instances in the cloud.

Allocated 1 — Select networks from those listed below.

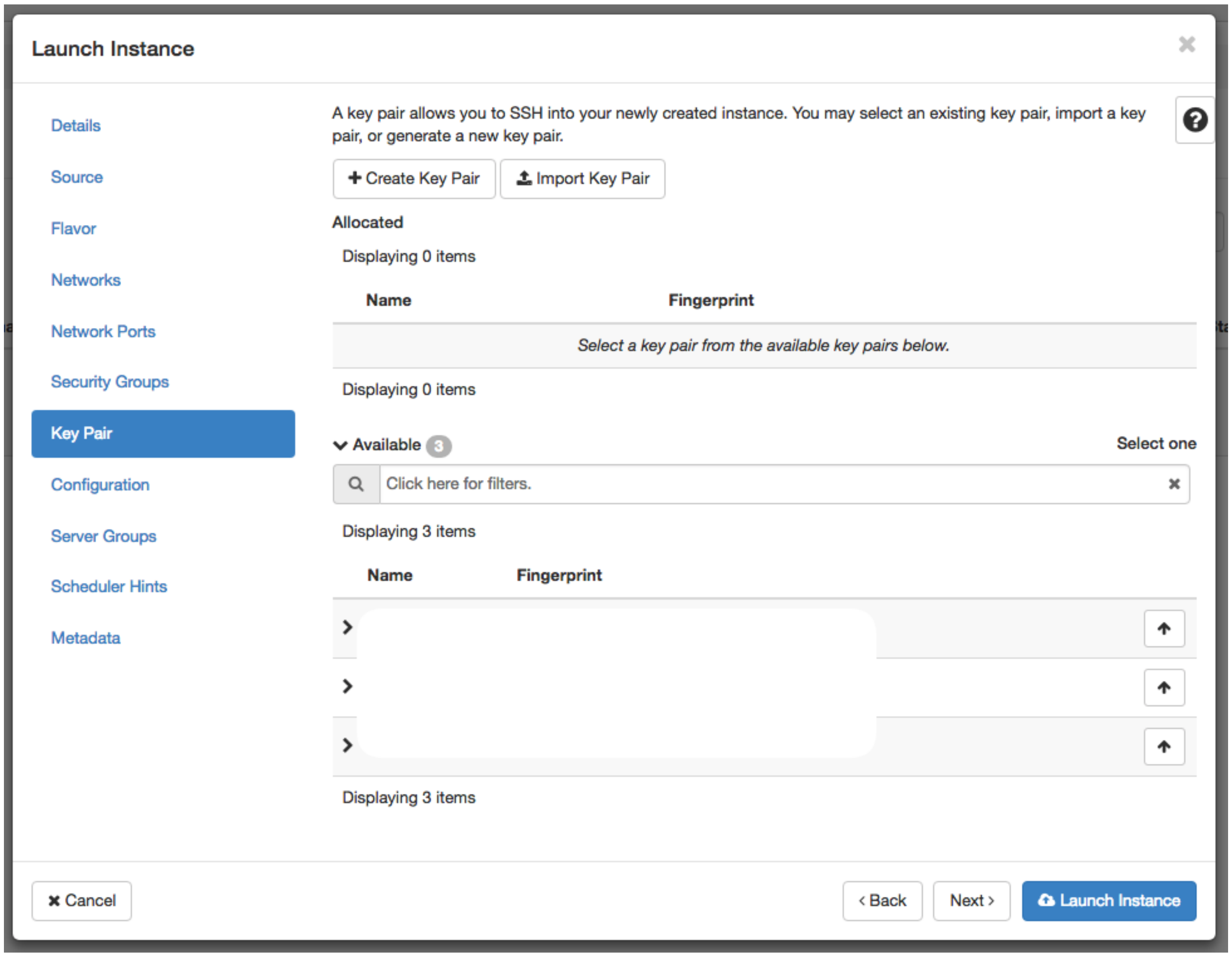
	Network	Subnets Associated	Shared	Admin State	Status
1	int_net	subnet1	Yes	Up	Active

Available 1 — Select at least one network

Click here for filters.

Network	Subnets Associated	Shared	Admin State	Status
ext_net	subnet2	No	Up	Active

- Create or use ssh key to enable ssh access to the VM:



Launch Instance

A key pair allows you to SSH into your newly created instance. You may select an existing key pair, import a key pair, or generate a new key pair.

Allocated

Displaying 0 items

Name	Fingerprint
Select a key pair from the available key pairs below.	

Displaying 0 items

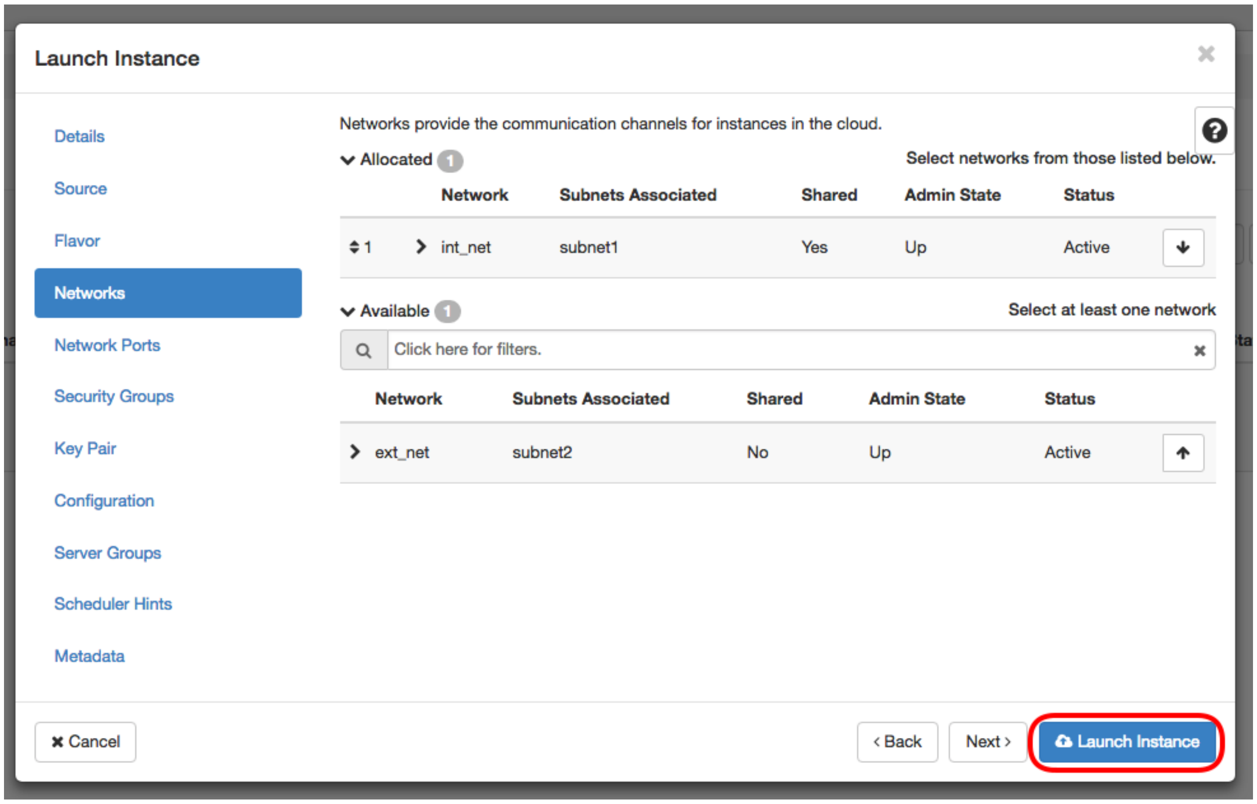
Available 3 — Select one

Click here for filters.

Displaying 3 items

Displaying 3 items

- In the end press on “Launch Instance” button at the bottom. This initiates the virtual machine deployment. It could take a while:



Networks provide the communication channels for instances in the cloud.

Allocated 1 — Select networks from those listed below.

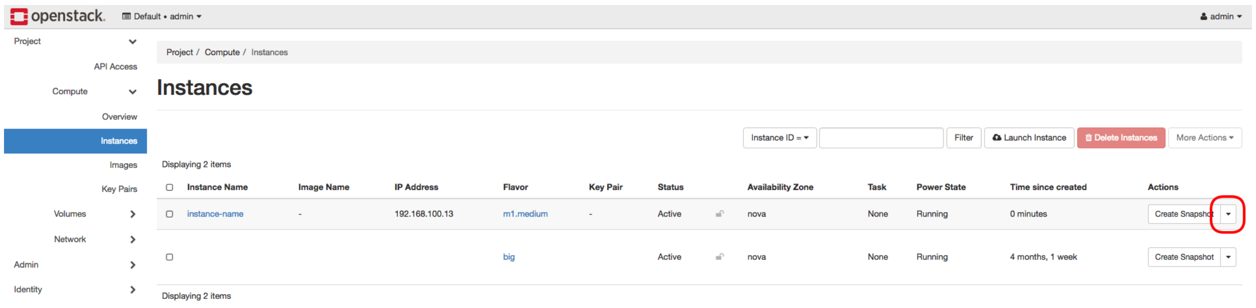
	Network	Subnets Associated	Shared	Admin State	Status
1	int_net	subnet1	Yes	Up	Active

Available 1 — Select at least one network

Click here for filters.

Network	Subnets Associated	Shared	Admin State	Status
ext_net	subnet2	No	Up	Active

- When the Power State become "Running", the virtual machine has been successfully deployed and is ready to be used:



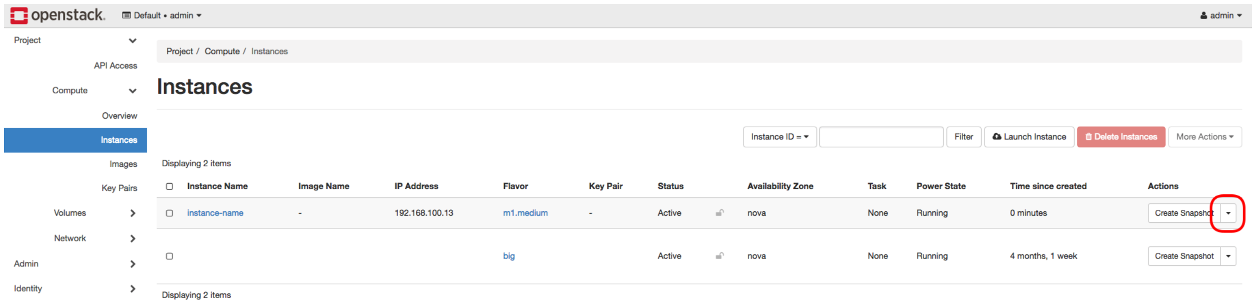
Instances

Displaying 2 items

Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Time since created	Actions
instance-name	-	192.168.100.13	m1.medium	-	Active	nova	None	Running	0 minutes	Create Snapshot
			big		Active	nova	None	Running	4 months, 1 week	Create Snapshot

Displaying 2 items

- Since the VM is deployed inside private network you will need to associate Floating IP from your network have the access:

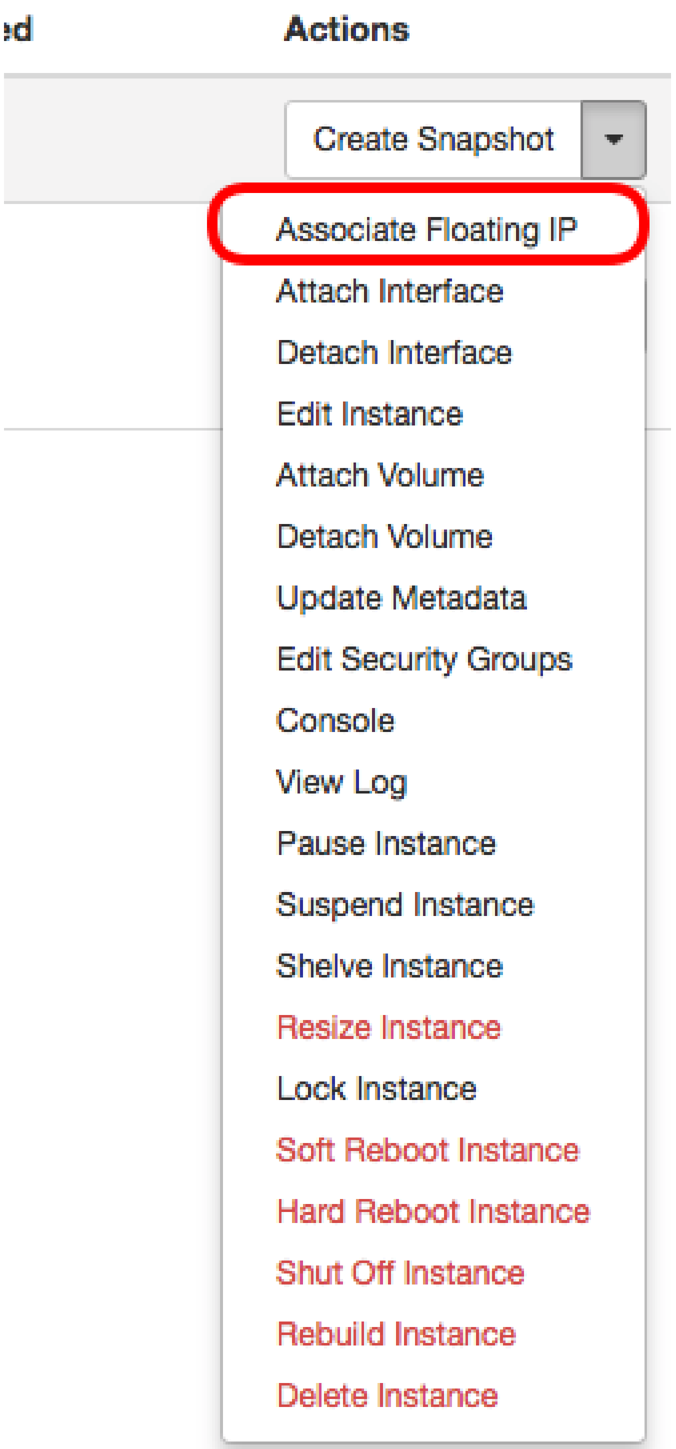


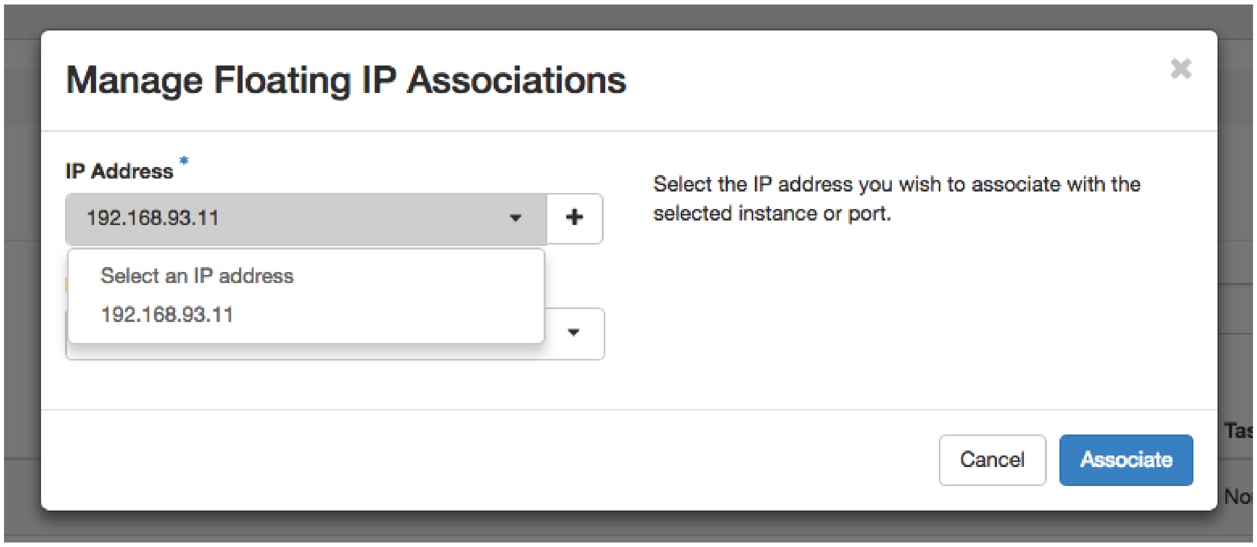
Instances

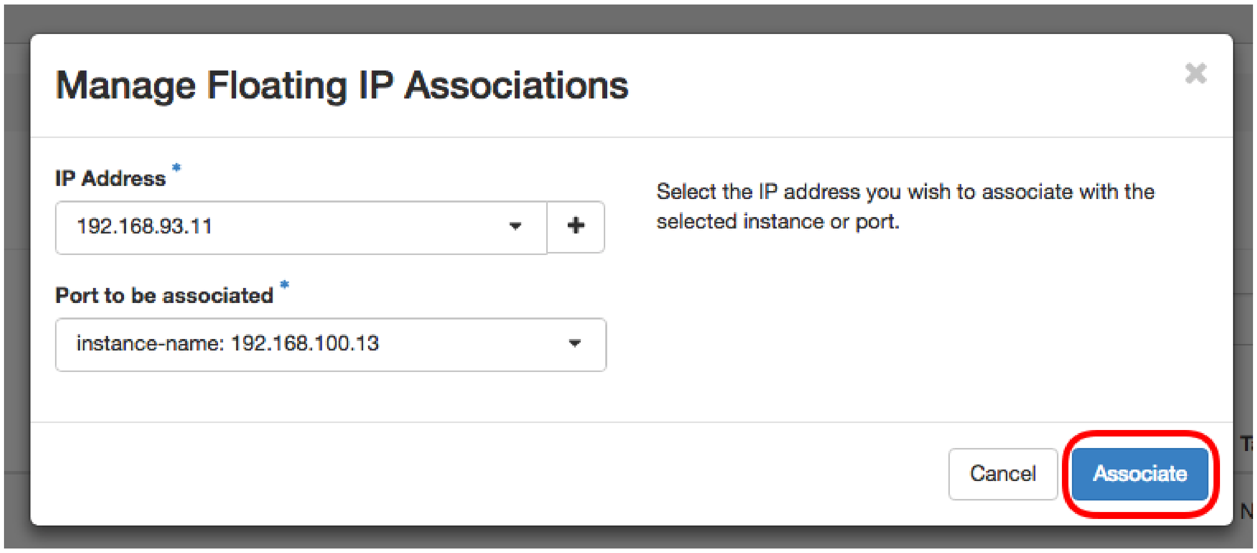
Displaying 2 items

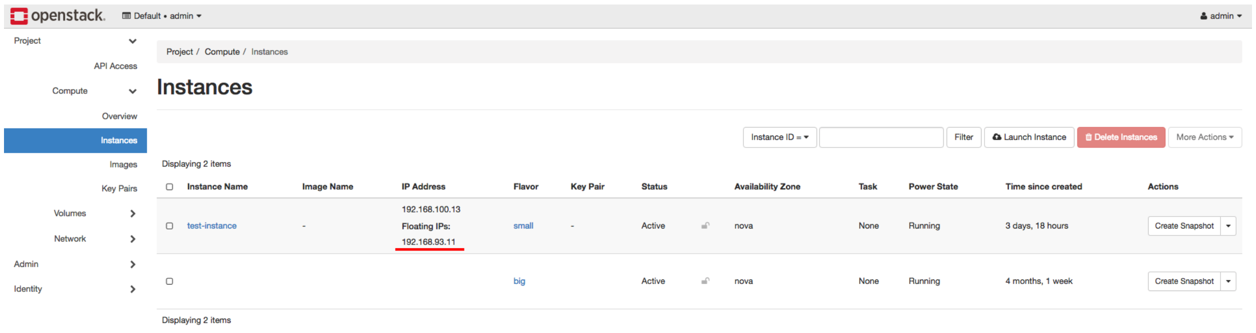
Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Time since created	Actions
instance-name	-	192.168.100.13	m1.medium	-	Active	nova	None	Running	0 minutes	Create Snapshot
			big		Active	nova	None	Running	4 months, 1 week	Create Snapshot

Displaying 2 items


ed
Actions
Create Snapshot
Associate Floating IP
Attach Interface
Detach Interface
Edit Instance
Attach Volume
Detach Volume
Update Metadata
Edit Security Groups
Console
View Log
Pause Instance
Suspend Instance
Shelve Instance
Resize Instance
Lock Instance
Soft Reboot Instance
Hard Reboot Instance
Shut Off Instance
Rebuild Instance
Delete Instance


Manage Floating IP Associations
IP Address *
192.168.93.11
Select an IP address
192.168.93.11
Select the IP address you wish to associate with the selected instance or port.
Cancel
Associate


Manage Floating IP Associations
IP Address *
192.168.93.11
Port to be associated *
instance-name: 192.168.100.13
Select the IP address you wish to associate with the selected instance or port.
Cancel
Associate



Displaying 2 items

Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Time since created	Actions
test-instance	-	192.168.100.13 Floating IPs: 192.168.93.11	small	-	Active	nova	None	Running	3 days, 18 hours	Create Snapshot
			big		Active	nova	None	Running	4 months, 1 week	Create Snapshot

Displaying 2 items

- Now using any SSH client connect to the instance through VPN using the Floating IP address. The login user is **ubuntu** when using the Ubuntu base images and **centos** for the CentOS ones.

Docker machine deployment

Official docker-machine documentation: https://docs.docker.com/machine/overview/

1. Installation

Guide: https://docs.docker.com/machine/install-machine/

2. Configuration

In order to use the OpenStack integration you need to export OpenStack Authentication credentials.

For the future use, create an executable file which will export environmental variables automatically. For example you can call file "openstackrc" and the content of the file be:

```
# VM CONFIG
export OS_SSH_USER=ubuntu
export OS_IMAGE_NAME=Ubuntu1604
export OS_FLAVOR_NAME=m1.medium
export OS_FLOATINGIP_POOL=ext_net
export OS_SECURITY_GROUPS=default
export OS_NETWORK_NAME=int_net

# AUTH
export OS_DOMAIN_NAME=default
export OS_USERNAME=<OPENSTACK_USER>
export OS_PASSWORD=<OPENSTACK_PASS>
export OS_TENANT_NAME=geral
export OS_AUTH_URL=http://192.168.93.215:5000/v3
```

OS_SSH_USER Default ssh user, usually it is ubuntu (if operating system is ubuntu)

OS_IMAGE_NAME OS image to be used during virtual machine deployment

OS_FLAVOR_NAME Virtual machine specification (vCPU, RAM, storage, . . .)

Flavor	vCPU	Root Disk	RAM
m1.tiny	1	0	0.5GB
m1.smaller	1	0	1GB
m1.small	1	10GB	2GB
m1.medium	2	10GB	3GB
m1.large	4	10GB	8GB
m1.xlarge	8	10GB	8GB
ska1.full	46	10GB	450GB

OS_FLOATINGIP_POOL Floating IP external network pool is the "ext_net"

OS_SECURITY_GROUPS Security groups, default is "default"

OS_NETWORK_NAME Private network, default is "int_net"

OS_DOMAIN_NAME OpenStack domain region, default is "default"

OS_USERNAME OpenStack username

OS_PASSWORD OpenStack password

OS_TENANT_NAME OpenStack project name, default is "geral"

OS_AUTH_URL OpenStack Auth URL, default is "http://192.168.93.215:5000/v3"

3. Usage

Complete documentation about docker-machine CLI commands can be found here: https://docs.docker.com/machine/reference/

3.1 Run the enviromental variable file

```
$ . openstackrc
```

3.2 Create docker-machine

Create a machine. Requires the –driver flag to indicate which provider (OpenStack) the machine should be created on, and an argument to indicate the name of the created machine.

```
$ docker-machine create --driver=openstack MACHINE-NAME

Creating CA: /root/.docker/machine/certs/ca.pem
Creating client certificate: /root/.docker/machine/certs/cert.pem
Running pre-create checks...
Creating machine...
(MACHINE-NAME) Creating machine...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with ubuntu(systemd)...
Installing Docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on
this virtual machine, run: docker-machine env MACHINE-NAME
```

In this step docker-machine will create VM inside OpenStack. As soon as the ssh connection to VM is available the Docker service will be installed.

3.3 Set docker-machine environment

Set environment variables to dictate that docker should run a command against a particular machine.

```
$ docker-machine env MACHINE-NAME

export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.93.23:2376"
export DOCKER_CERT_PATH="/root/.docker/machine/machines/MACHINE-NAME"
export DOCKER_MACHINE_NAME="MACHINE-NAME"
# Run this command to configure your shell:
# eval $(docker-machine env MACHINE-NAME)
```

3.4 Configure shell to use your docker-machine

After this, when you execute "docker" command it will be executed remotely.

```
$ eval $(docker-machine env MACHINE-NAME)
```

Now if you run "docker-machine ls" you see that your machine is active and ready to use.

```
$ docker-machine ls

NAME           ACTIVE   DRIVER      STATE     URL                          SWARM   ␣
↪DOCKER     ERRORS
MACHINE-NAME   *        openstack   Running   tcp://192.168.93.23:2376             v18.
↪09.0
```

3.5 Use "docker" command to remotely deploy docker containers

```
$ docker run -d -p 80:80 nginx

Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
a5a6f2f73cd8: Pull complete
67da5fbcb7a0: Pull complete
e82455fa5628: Pull complete
Digest: sha256:98b06873ea9c87d5df1bb75b650926cfbcc4c53f675dfabb158830af0b115f99
Status: Downloaded newer image for nginx:latest
889a1ab275ba072980fe4fd3ec58094513cf41330c3698b226c239ba490a24a6
```

3.6 Remove docker-machine

Remove a machine. This removes the local reference and deletes it on the cloud rr or virtualization management platform.

```
$ docker-machine rm MACHINE-NAME (-f if need force)
```

3.7 Docker-machine IP

Get the IP address of one or more machines.

```
$ docker-machine ip MACHINE-NAME

192.168.93.23
```

3.8 Docker-machine list

List currently deployed docker-machines.

```
$ docker-machine ls

NAME           ACTIVE   DRIVER      STATE     URL                        SWARM  ␣
↪DOCKER     ERRORS
MACHINE-NAME   *        openstack   Running   tcp://192.168.93.23:2376           v18.
↪09.0
```

3.9 Docker-machine upgrade

Upgrade a machine to the latest version of Docker. How this upgrade happens depends on the underlying distribution used on the created instance.

```
$ docker-machine upgrade MACHINE-NAME

Waiting for SSH to be available...
Detecting the provisioner...
Upgrading docker...
Restarting docker...
```

3.10 Docker-machine stop

Stops running docker-machine.

```
$ docker-machine stop MACHINE-NAME

Stopping "MACHINE-NAME"...
Machine "MACHINE-NAME" was stopped.
```

3.11 Docker-machine restart

Restarts docker-machine.

```
$ docker-machine restart MACHINE-NAME

Restarting "MACHINE-NAME"...
Waiting for SSH to be available...
Detecting the provisioner...
Restarted machines may have new IP addresses. You may need to re-run the
`docker-machine env` command.
```

3.12 Docker-machine start

Starts docker-machine.

```
$ docker-machine start MACHINE-NAME

Starting "MACHINE-NAME"...
Machine "MACHINE-NAME" was started.
Waiting for SSH to be available...
Detecting the provisioner...
```

(continues on next page)

(continued from previous page)

```
Started machines may have new IP addresses. You may need to re-run the
`docker-machine env` command.
```

3.13 Docker-machine ssh

Log into or run a command on a machine using SSH.

```
$ docker-machine ssh MACHINE-NAME

Welcome to Ubuntu 16.04.4 LTS (GNU/Linux 4.4.0-116-generic x86_64)

* Documentation:  https://help.ubuntu.com
* Management:     https://landscape.canonical.com
* Support:        https://ubuntu.com/advantage

Get cloud support with Ubuntu Advantage Cloud Guest:
http://www.ubuntu.com/business/services/cloud

153 packages can be updated.
81 updates are security updates.

New release '18.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


ubuntu@MACHINE-NAME:~$
```

Access to the bare metal

In this stage, this option is very restrictive and only in a well-justified situation is allowed.

2.10.2 Monitoring Dashboards

There are several dashboards that can be used to monitor the SKA infrastructure and projects' statuses.

In order to access the dashboards, first you should add below lines to your *hosts* file. *Note that IP addresses of the dashboards are subject to change anytime. If there are any problems with the below dashboards please contact the system team.*

```
# hosts file ("/etc/hosts" for linux/unix, "%WINDIR%\System32\drivers\etc\hosts" for␣
↪Windows)
# Dashboards
192.168.93.102 integration.engageska-portugal.pt
192.168.93.102 tango.rest.integration.engageska-portugal.pt
192.168.93.102 tangogql-proxy.integration.engageska-portugal.pt
193.136.93.245 kibana.engageska-portugal.pt monitoring.engageska-portugal.pt
```

Some of the dashboards are password protected. To get the current passwords please contact the system team.

- SKA Infrastructure: Monitoring metrics such as Kubernetes, GitLab Runners, Elasticstack, cadvisor etc.
- Prometheus Alerts: Prometheus alerts for the core kubernetes cluster and infrastructure VMs
- Kibana Logs: Kibana Logs for the core cluster

- Webjive: Webjive dashboard for the Tango Devices
- Argos: Argos grafana dashboard for Testing purposes for the repositories test metrics such as coverage, number of executed tests, mean values etc. Note: this dashboard has only some repositories at the moment for prototyping

2.11 Containers: containerisation and deployment

2.11.1 Containerisation Standards

This section describes a set of standards, conventions and guidelines for building, integrating and maintaining container technologies.

Table of Contents

Overview of Standards

These standards, best practices and guidelines are based on existing industry standards and tooling. The main references are:

- Docker v2 Registry API Specification.
- Dockerfile best practices.
- Container Network Interface.
- Container Storage Interface.
- Open Container Initiative image specification.
- Open Container Initiative run-time specification.

The standards are broken down into the following areas:

- Structuring applications in a containerised environment - general guidelines for breaking up application suites for running in a containerised way
- Defining and building container images - how to structure image definitions, and map your applications onto the image declaration
- Running containerised applications - interfacing your application with the container run time environment

Throughout this documentation, Docker is used as the reference implementation with the canonical version being Docker 18.09.4 CE API version 1.39, however the aim is to target compliance with the OCI specifications so it is possible to substitute in alternative Container Engines, and image build tools that are compatible.

Cheatsheet

A *Container Standards CheatSheet* is provided at the end of this document as a quick guide to standards and conventions elaborated on throughout this document.

Structuring applications in a Containerised Environment

How does containerisation work

Containerisation is a manifestation of a collection of features of the Linux kernel and OS based on:

- Capabilities (CAPS) - POSIX 1003.1e capabilities - predate namespaces, but genesis for Linux unknown - approximately Kernel 2.2 onwards
- Cgroups - introduced in January 2008
- File-system magic - such as pivot_root, and bind mounting first appeared in Linux 2.4 - circa 2001
- Namespaces - introduced in 2002

These features combine to give a form of lightweight virtualisation that runs directly in the host system Kernel of Linux, where the container is typically launched by a Container Engine such as Docker.

Namespaces create the virtualisation effect by switching the init process (PID 1) of a container into a separate namespace of the Kernel for processes, network stacks and mount tables so as to isolate the container from all other running processes in the Kernel. **Cgroups** provide a mechanism for controlling resource allocation eg: Memory, CPU, Net, and IO quotas, limits, and priorities. **Capabilities** are used to set the permissions that containerised processes have for performing system calls such as IO. The **file-system magic** performed with pivot_root recasts the root of the file-system for the container init process to a new mount point, typically the root of the container image directory tree. Then, bind mounting enables sharing file-system resources into a container.

Container Image

The Linux Kernel features make it possible for the container virtualisation to take place in the Kernel, and to have controls placed on the runtime of processes within that virtualisation. The container image is the first corner stone of the software contract between the developer of a containerised application and the Container Engine that implements the Virtualisation. The image is used to encapsulate all the dependencies of the target application including executables, libraries, static configuration and sometimes static data.

The OCI Image specification defines a standard for constructing the root file-system that a containerised application is to be launched from. The file-system layout of the image is just like the running application would expect and need as an application running on a virtual server. This can be as little as an empty `/` (root) directory for a fully statically linked executable, or it could be a complete OS file-system layout including `/etc`, `/usr`, `/bin`, `/lib` etc. - whatever the target application needs.

According to the OCI specification, these images are built up out of layers that typically start with a minimal OS such as AlpineLinux, with successive layers of modification that add libraries, configuration and other application dependencies.

At container launch, the image layers of the specified image are stacked up in ascending order using a Union File-System. This creates a complete virtual file-system view, that is read only (if an upper layer has the same file as a

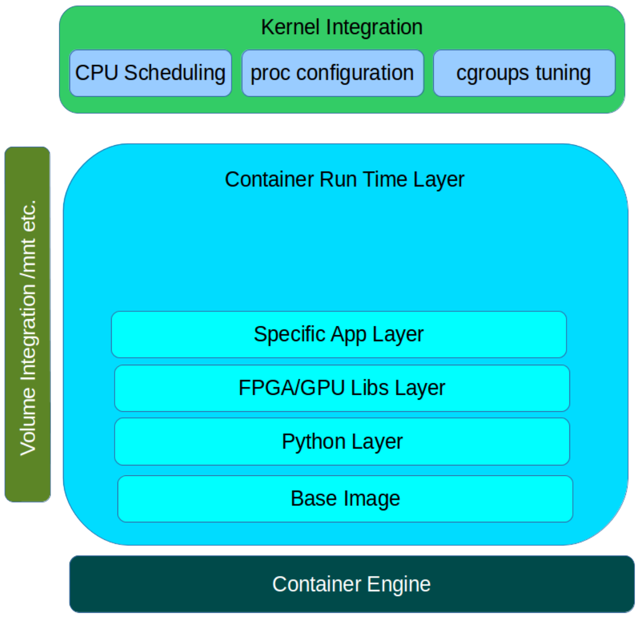


Fig. 7: The basic anatomy of a container and how it interfaces with host at run time.

lower layer, the lower layer is masked). Over the top of this file-system pancake stack a final read/write layer is added to complete the view that is passed into the container as it's root file-system at runtime.

Network Integration

Different Container Engines deal with networking in varying ways at runtime, but typically it comes in two flavours:

- host networking - the host OS network stack is pushed into the container
- a separate virtual network is constructed and bridged into the container namespace

There are variations available within Docker based on overlay, macvlan and custom network plugins that conform to the CNI specification.

Hostname, and DNS resolution is managed by bind mounting a custom /etc/hosts and /etc/resolv.conf into the container at runtime, and manipulating the UTS namespace.

Storage Integration

External storage required at runtime by the containerised application is mapped into the container using bind mounting. This takes a directory location that is already present on the host system, and maps it into the specified location within the container file-system tree. This can be either files or directories. The details of how specialised storage is made available to the container is abstracted by the Container Engine which should support the CSI specification for drivers integrating storage solutions. This is the same mechanism used to share specialised devices eg: `/dev/nvidia0` into a container.

Structuring Containerised Applications

Each containerised application should be a single discrete application. A good test for this is:

- is there a single executable entry point for the container?
- is the running process fulfilling a single purpose?
- is the process independently maintainable and upgradable?
- is the running process independently scalable?

For example, `iperf`, or `apache2` as separate containerised applications are correct, but putting `NGiNX` and `PostgreSQL` in a single container is wrong. This is because `NGiNX` and `PostgreSQL` should be independently maintained, upgraded and scaled, an init process handler would be required to support multiple parenet processes, and signals would not be correctly propagated to these parent processes (eg: Postgres and NGiNX) from the Container Engine.

A containerised application should not need a specialised multi-process init process such as `supervisord`. As soon as this is forming part of the design, there should almost always be an alternative where each application controlled by the `init` process is put into a separate container. Often this can be because the design is trying to treat a container like a full blown Virtual Machine through adding `sshd`, `syslog` and other core OS services. This is not an optimal design because these services will be multiplied up with horizontal scaling of the containerised application wasting resources. In both these example cases, `ssh` is not required because a container can be attached to for diagnostic purposes eg: `docker exec ...`, and it is possible to bind mount `/dev/log` from the host into a container or configure the containerised application to point to `syslog` over TCP/UDP.

Take special care with signal handling - the Container Engine propagates signals to init process which should be the application (using the EXEC for of entry point). If not it will be necessary to ensure that what ever wrapper (executable, shell script etc.) is used propagates signals correctly to the actual application in the container. This is particularly important at termination time where the Engine will typically send a SIGHUP waiting for a specified timeout and then following up with a SIGKILL. This could be harmful to stateful applications such as databases, message queues, or anything that requires an orderly shutdown.

A container image among other things, is a software packaging solution, so it is natural for it to follow the same Software Development Life Cycle as the application held inside. This also means that it is good practice for the released container image versions to map to the released application versions. An example of this in action is the NGiNX Ingress Controller releases. By extension, this also leads to having one Git repository and container image per application in order to correctly manage independent release cycles.

Defining and Building Container Images

The core of a containerised application is the image. According to the OCI specification, this is the object that encapsulates the executable and dependencies, external storage (VOLUME) and the basics of the launch interface (the ENTRYPOINT and ARGS).

The rules for building an image are specified in the `Dockerfile` which forms a kind of manifest. Each rule specified creates a new layer in the image. Each layer in the image represents a kind of high watermark of an image state which can ultimately be shared between different image builds. Within the local image cache, these layer points can be shared between running containers because the image layers are stacked as a read only UnionFS. This Immutability is a key concept in containers. containers should not be considered mutable and therefore precious. The goal is that it should be possible to destroy and recreate them with (little or) no side effects.

If there is any file-system based state requirement for a containerised application, then that requirement should be satisfied by mounting in external storage. This will mean that the container can be killed and restarted at anytime, giving a pathway to upgrade-ability, maintainability and portability for the application.

The Image

When structuring the image build eg: `Dockerfile`, it is important to:

- minimise the size of the image, which will speed up the image pull from the repository and the container launch
- minimise the number of layers to speed up the container launch through speeding up the assembly process
- order the layers from most static to least static so that there is less churn and depth to the image rebuild process eg: why rebuild layers 1-5 if only 6 requires building.

Image Definition Syntax

Consistency with `Dockerfile` syntax will make code easier to read. All directives and key words should be in upper case, leaving a clear distinction from image building tool syntax such as Unix commands.

All element names should be in lower case eg: image labels and tags, and arguments (`ARG`). The exception is environment variables (`ENV`) as it is customary to make them all upper case within a shell environment.

Be liberal with comments (lines starting with `#`). These should explain each step of the build and describe any external dependencies and how changes in those external dependencies (such as a version change in a base image, or included library) might impact on the success of the build and the viability of the target application.

```
# This application depends on type hints available only in 3.7+
# as described in PEP-484
ARG base_image="python:3.7"
FROM $base_image
...
```

Where multi-line arguments are used, sort them for ease of reading, eg:

```
RUN apt-get install -y \
        apache2-bin \
        binutils \
        cmake
...
```

Build Context

The basic build process is performed by:

```
docker build -t <fully qualified tag for this image> \
             -f path/to/Dockerfile \
             project/path/to/build/context
```

The build context is a directory tree that is copied into the image build process (just another container), making all of the contained files available to subsequent `COPY` and `ADD` commands for pushing content into the target image. The size of the build context should be minimised in order to speed up the build process. This should be done by specifying a path within the project that contains only the files that are required to be added to the image.

Always be careful to exclude unnecessary and sensitive files from the image build context. Aside from specifying a build context directory outside the root of the current project, it is also possible to specify a `.dockerignore` file which functions like a `.gitignore` file listing exclusions from the initial copy into the build context. Never use `ADD`, `COPY` or `ENV` to include secret information such as certificates and passwords into an image eg: `COPY id_rsa .ssh/id_rsa`. These values will be permanently embedded in the image (even buried in lower layers), which may then be pushed to a public repository creating a security risk.

Minimise Layers

Image builds tend to be highly information dense, therefore it is important to keep the scripting of the build process in the `Dockerfile` short and succint. Break the build process into multiple images as it is likely that part of your proposed image build is core and common to other applications. Sharing base images (and layers) between derivative images will improve download time of images, and reduce storage requirements. The Container Engine should only download layers that it does not already have - remember, the UnionFS shares the layers between running containers as it is only the upper most layer that is writable. The following example illustrates a parent image with children:

```
FROM python:latest
RUN apt-get install -y libpq-dev \
                postgresql-client-10
RUN pip install psycopg2 \
                sqlalchemy
```

The image is built with `docker build -t python-with-postgres:latest .`. Now we have a base image with Python, Postgres, and SQLalchemy support that can be used as a common based for other applications:

```
FROM  python-with-postgres:latest
COPY ./app /app
...
```

Minimising layers also reduces the build and rebuild time - `ENV`, `RUN`, `COPY`, and `ADD` statements will create intermediate cached layers.

Multi-stage Builds

Within a `Dockerfile` it is possible to specify multiple dependent build stages. This should be used to reduce the final size of an image. For example:

```
FROM python-builder:latest AS builder
COPY requirements.txt .
RUN pip3 install -r requirements.txt

FROM python-runtime:latest
COPY --from=builder /usr/local /usr/local
...
```

This uses an imaginary Python image with all the development tools, and necessary compilers as a named intermediate image called `builder` where dependent libraries are compiled, and built and then the target image is created from an imaginary streamlined Python runtime image which has the built libraries copied into it from the original build, leaving behind all of the nolonger required build tools.

Encapsulation of Data with Code

Avoid embedding configuration and data that your application requires in the container image. The only exceptions to this should be:

- The configuration or data is guaranteed to be static
- The configuration or data is tiny (kilo-bytes to few mega-bytes), well defined, and forms sensible defaults for the running application

To ignore this, will likely make your container implementation brittle and highly specific to a use case, as well as bloating the image size. It is better practice to mount configuration and data into containers at runtime using environment variables and volumes.

Base Images

Base images and image provenance will need to be checked in order to maintain the security and integrity of the SKA runtime systems. This is will include (but not limited to) automated processes for:

- Code quality checking for target applications
- Vulnerability scanning
- Static application security testing
- Dependency scanning
- License scanning
- Base image provenance tree

Ensuring that the base images and derivative images are safe and secure with verifiable provenance wll be important to the security of the entire platform, so it will be important to choose a base image that will pass these tests. To assist with this, the SKA will curate a set of base images for the supported language environments so that developers can have a supported starting position. Discuss your requirements with the Systems Team, so that they can be captured and supported in advance.

As a general rule, stable image tags should be used for base images that at least include the Major and Minor version number of Semantic Versioning eg: `python:3.7`. As curated base images come from trusted sources, this ensures that the build process gets a functionally stable starting point that will still accrue bug fixing and security patching. Do not use the `latest` tag as it is likely that this will break your application in future, and it gives no indication of the container developers last tested environment specification.

Reduce Image Size

Avoid installing unnecessary packages in your container image. Your production container should not automatically require a debugger, editor or network analysis tools. Leave these out, or if they are truly required, then create a derivative image from the standard production one explicitly for the purposes of debugging, and problem resolution. Adding these unnecessary packages will bloat the image size, and reduce the efficiency of image building, and shipping as well as unnecessarily expose the production container to potential further security vulnerabilities by increasing the attack surface.

RUN and packages

When installing packages with the `RUN` directive, always clean the package cache afterwards to avoid the package archives and other temporary files unnecessarily becoming part of the new layer - eg:

```
...
RUN \
    apt-get update && \
    apt-get install -y the-package && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
...
```

Ordering

Analyse the order of the build directives specified in the `Dockerfile`, to ensure that they are running from the lowest frequency changing to the highest.

Consider the following:

```
FROM python:latest
ARG postgres_client "postgresql-client-10 libpq-dev"
RUN apt-get install -y $postgres_client
COPY requirements.txt .
RUN pip3 install -r requirements.txt
COPY ./app /app
...
```

Looking at the example above, during the intensive development build phase of an application, it is likely that the most volitile element is the `./app` itself, followed by the Python dependencies in the `requirements.txt` file, then finally the least changeable element is the specific postgresql client libraries (the base image is always at the top).

Laying out the build process in this way ensures that the build exploits as much as possible the build cache that the Container Engine holds locally. The cache calculates a hash of each element of the `Dockerfile` linked to all the previous elements. If this hash has not changed then the build process will skip the rebuild of that layer and pull it from the cache instead. If in the above example, the `COPY ./app /app` step was placed before the `RUN apt-get install`, then the package install would be triggered every time the code changed in the application unnecessarily.

Labels

Use the `LABEL` directive to add ample metadata to your image. This metadata is inherited by child images, so is useful for provenance and traceability.

```
...
LABEL \
      author="A Developer <a.developer@example.com>" \
      description="This image illustrates LABELs" \
      license="Apache2.0" \
      registry="acmeincorporated/imagename" \
      vendor="ACME Incorporated" \
      org.skatelescope.team="Systems Team" \
      org.skatelescope.version="1.0.0" \
      org.skatelescope.website="http://gitlab.com/ACMEIncorporate/widget"
...
```

The following are recommended labels for all images:

- author: name and email address of the author
- description: a short description of this image and it's purpose.
- license: license that this image and contained software are released under
- registry: the primary registry that this image should be found in
- vendor: the owning organisation of the software component
- org.skatelescope.team: the SKA team responsible for this image.
- org.skatelescope.version: follows Semantic Versioning, and should be linked to the image version tag discussed below.
- org.skatelescope.website: where the software pertaining to the building of this image resides

Arguments

Use arguments via the ARG directive to parameterise elements such as the base image, and versions of key packages to be installed. This enables reuse of the build recipe without modification. Always set default values, as these can be overridden at build time, eg:

```
ARG base_image="python:latest"
FROM $base_image
RUN apt-get install -y binutls cmake
ARG postgres_client="postgresql-client-10 libpq-dev"
RUN apt-get install -y $postgres_client
...
```

The ARGs referenced above can then be addressed at build time with:

```
docker build -t myimage:latest \
             --build-arg base_image="python:3" \
             --build-arg postgres_client="postgresql-client-9 libpq-dev"
             -f path/to/Dockerfile \
             project/path/to/build/context
```

Note: the `ARG postgres_client` is placed after the `apt-get install -y binutls cmake` as this will ensure that the variable is bound as late as possible without invalidating the layer cache of that package install.

Environment Variables

Only set environment variables using ENV if they are required in the final image. ENV directives create layers and a permanent record of values that are set, even if they are overridden by a subsequent ENV directive. If an environment variable is required by a build step eg: `RUN gen-myspecial-hash`, then chain the `export` of the variable in the RUN statement, eg:

```
...
RUN export THE_HASH="wahoo-this-should-be-secret" \
    && gen-myspecial-hash \
    && unset THE_HASH
...
```

This ensures that the value is ephemeral, at least from the point of view of the resultant image.

ADD or COPY + RUN vs RUN + curl

ADD and COPY are mostly interchangeable, however `ADD my-fancy.tar.gz /tmp` might not do what you expect in that it will auto-extract the archive at the target location. COPY is the preferred mechanism as this does not have any special behaviours.

Be clear of what the purpose of the COPY or ADD statement is. If it is a dependency only for a subsequent build requirement, then consider replacing with RUN eg:

```
...
RUN \
    mkdir /usr/local/dist && cd /usr/local/dist && \
    curl -O https://shibboleth.net/downloads/identity-provider/3.2.1/shibboleth-
↪identity-provider-3.2.1.tar.gz && \
    tar -zxf shibboleth-identity-provider-3.2.1.tar.gz && \
```

(continues on next page)

(continued from previous page)

```
    rm shibboleth-identity-provider-3.2.1.tar.gz
...
```

The above example downloads and installs the software archive, and then removes it within the same image layer, meaning that the archive file is not left behind to bloat the resultant image.

USER and WORKDIR

It is good practice to switch the container user to a non privelleged account when possible for the application, as this is good security practice, eg: `RUN groupadd -r userX && useradd --no-log-init -r -g userX userX`, and then specify the user with `USER userX[:userX]`.

Never use sudo - there should never be a need for an account to elevate permissions. If this seems to be required then please revisit the architecture, discuss with the Systems Team and be sure of the reasoning.

`WORKDIR` is a helper that sets the default directory at container launch time. Aside from being good practice, this is often helpful when debugging as the path and context is already set when using `docker exec -ti ....`

ENTRYPOINT and CMD

`ENTRYPOINT` and `CMD` are best used in tandem, where `ENTRYPOINT` is used as the default application (fully qualified path) and `CMD` is used as the default set of arguments passed into the default application, eg:

```
...
ENTRYPOINT ["/bin/cat"]
CMD ["/etc/hosts"]
...
```

It is best to use the `["thing"]` notation as this is the `exec` format ensuring that proper signal propagation occurs to the containerised application.

It is often useful to create an entry point script that encapsulates default flags and settings passed to the application, however, still ensure that the final application launch in the script uses `exec /path/to/my/app ...` so that it becomes PID 1.

ONBUILD and the undead

ONBUILD is a powerful directive that enables the author of an image to enforce an action to occur in a subsequent derivative image build, eg:

```
FROM python:latest
RUN pip3 install -r https://example.com/parent/image/requirements.txt
ONBUILD COPY ./app ./app
ONBUILD RUN chmod 644 ./app/bin/*
...
```

Built with `docker build -t myimage:1.0.0-onbuild .`

In any child image created `FROM myimage:1.0.0-onbuild ...`, the parent image will seemingly call back from the dead and execute statement `COPY ./app ./app` and `RUN chmod 644 ./app/bin/*` as soon as the `FROM` statement is interpreted. As there is no obvious way to tell whether an image has embedded `ONBUILD` statements (without `docker inspect myimage:1.0.0-onbuild`), it is customary to add an indicator to the tag name as above: `myimage:1.0.0-onbuild` to act as a warning to the developer. Use the `ONBUILD` feature sparingly, as it can easily cause unintended consequences and catch out dependent developers.

Naming and Tagging

Image names should reflect the application that will run in the resultant container, which ideally ties in directly with the repository name eg: `tango-example/powersupply:latest`, is the image that represents the Tango powersupply device from the tango-example repository.

Images should be tagged with:

- short commit hash as derived by `git rev-parse --verify --short=8 HEAD` from the parent repository eg: bbedf059. This is useful on each feature branch build as it uniquely identifies branch HEAD on each push when used in conjunction with Continuous Integration.
- When an image version for an application is promoted to production, it should be tagged with the application version (using Semantic Versioning). For the latest most major.minor.patch image version the 'latest' tag should be added eg: for a tango device and a released image instance with hash tag: 9fab040a, added version tags are: `1.13.2`, `1.13`, `1`, `latest` - where major/minor/patch version point to the latest in that series.
- A production deployment should always be made with a fully qualified semantic version eg: `tango-example/powersupply:1.13.2`. This will ensure that partial upgrades will not inadvertently make their way into a deployment due to historical scheduling. The `latest` tag today might point to the same hash as `1.13.2`, but if a cluster recovery was enacted next week, it may now point to `1.14.0`.

While it is customary for th Docker community at large to support image variants based on different image OS bases and to denote this with tags eg: `python:<version>-slim` which represents the Debian Slim (A trimmed Debian OS) version of a specific Python release, the SKA will endeavour to support only one OS base per image, removing this need as it does not strictly follow Semantic Versioning, and creates considerable maintenance overhead.

Within the SKA hosted Continuous Integration infrastructure, development and test images will be periodically purged from the repository after N months, leaving the last version built. All production images are kept indefinitely.

This way anyone who looks at the image repository will have an idea of the context of a particular image version and can trace it back to the source.

Image Tools

Any image build tool is acceptable so long as it adheres to the OCI image specification v1.0.0. The canonical tool used for this standards document is Docker 18.09.4 API version 1.39, but other tools maybe used such as BuildKit and img.

Development tools

Debuging tools, profilers, and any tools not essential to the running of the target application should not be included in the target application production image. Instead, a derivative image should be made solely for debugging purposes that can be swapped in for the running application as required. This is to avoid image bloat, and to reduce the attack surface of running containers as a security consideration. These derivative images should be named explicitly `dev` eg: `tango-example/powersupply-dev:1.13.2`.

Image Storage

All images should be stored in a Docker v2 Registry API compliant repository, protected by HTTPS. The SKA supported and hosted repositories are based on the Central Artefact Repository Container Registry available at artefact.skatelescope.org .

All containerised software used within the SKA, will be served out of the hosted repository service. This will ensure that images are quality assured and always remain available beyond the maintenance life-cycle of third party and COTs software.

Image signing and Publishing

All images pushed to the SKA hosted repository must be signed. This will ensure that only trusted content will be launched in containerised environments. Docker Content Trust signatures can be checked with:

```
$docker trust inspect --pretty \
   artefact.skatelescope.org/ska-tango-images/ska-python-runtime:1.2.3

Signatures for artefact.skatelescope.org/ska-tango-images/ska-python-runtime:1.2.3

SIGNED TAG          DIGEST                                                             ␣
↪ SIGNERS
1.2.3               3f8bb7c750e86d031dd14c65d331806105ddc0c6f037ba29510f9b9fbbb35960 ␣
↪ (Repo Admin)

Administrative keys for artefact.skatelescope.org/ska-tango-images/ska-python-
↪runtime:1.2.3

  Repository Key:   abdd8255df05a14ddc919bc43ee34692725ece7f57769381b964587f3e4decac
  Root Key: a1bbec595228fa5fbab2016f6918bbf16a572df61457c9580355002096bb58e1
```

Running Containerised Applications

As part of the development process for a containerised application, the developer must determine what **the application interface contract** is. Referring back to the *Container Anatomy* diagram above, a containerised application has a number of touch points with the underlying host through the Container Engine. These touch points form the interface and include:

- Network - network and device attachment, hostname, DNS resolution
- Volumes - persistent data and configuration files
- Ports
- Environment variables
- Permissions
- Memory
- CPU
- Devices
- OS tuning, and ulimits
- IPC
- Signal handling
- Command and arguments
- Treatment of StdIn, StdOut, and StdErr

Usage documentation for the image must describe the intended purpose of each of these configurable resources where consumed, how they combine and what the defaults are with default behaviours.

Container Resources

Management of container resources is largely dependent on the specific Container Engine in use. For example, Docker by default runs a container application in it's own namespace as the root user, however this is highly configurable. The following example shares devices, and user details with the host OS, effectively transparently running the application as the current user of the command line:

```
cat <<EOF | docker build -t mplayer -
FROM ubuntu:18.04
ENV DEBIAN_FRONTEND noninteractive
RUN \
    apt-get update && \
    apt-get install mplayer -y && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

ENTRYPOINT ["/usr/bin/mplayer"]
CMD ["--help"]
EOF

docker run --rm --name the-morepork-owl \
  --env HOME=${HOME} \
  --env DISPLAY=unix$DISPLAY \
  --volume /etc/passwd:/etc/passwd:ro \
  --user $(id -u) \
  --volume ${HOME}:${HOME} \
  --workdir ${HOME} \
  --volume /tmp/.X11-unix:/tmp/.X11-unix:ro \
  --volume /etc/machine-id:/etc/machine-id:ro \
  --volume /run/user/$(id -u):/run/user/$(id -u):ro \
  -ti mplayer /usr/bin/mplayer https://www.doc.govt.nz/Documents/conservation/native-
↪animals/birds/bird-song/morepork-song.mp3
```

Storage

As previously stated, all storage shared into a container is achieved through bind mounting. This is true for both directory mount points and individual files. While it is not mandatory to use the `VOLUME` directive in the image `Dockerfile`, it is good practice to do this for all directories to be mounted as it provides annotation of the image requirements. These volumes and files can be populated with default data, but be aware they are completely masked at runtime when overlayed by a mount.

When adding a volume at runtime, consider whether write access is really required. As with the example above `--volume /etc/passwd:/etc/passwd:ro` ensures that the `/etc/passwd` file is read only in the container reducing the security concerns.

Network

containerised applications should avoid using `--net=host` (host only) based networking as this will push the container onto the running host network namespace monopolising any ports that it uses. This means that another instance of this container or any other that uses the same ports cannot run on the same host severely impacting on scheduling and resource utilisation efficiencies.

Permissions

Where possible, a containerised application should run under a specific UIG/GID to avoid privilege escalation as an attack vector.

It should be a last resort to run the container in privileged mode `docker run --privileged ...`, as there are very few use cases that will require this. The most notable are when a container needs to load kernel modules, or a container requires direct host resource access (such as network stack, or specialised device) for performance reasons. Running a container in this mode will push it into the host OS namespace meaning that the container will monopolise any resources such as network ports (see *Network*).

Configuration

Configuration of a containerised application should be managed primarily by:

- *Environment Variables*
- configuration files

Avoid passing large numbers of configuration options on the command line, and service connection information that could contain secrets such as keys and passwords should not be passed as options, as these can appear in the host OS process table.

Configuration passed into a container should not directly rely on a 3rd party secret/configuration service integration such as vault, consul or etcd. If integration with these services are required, then a sidecar configuration provider architecture should be adopted that specifically handles these environment specific issues.

Appropriate configuration defaults should be defined in the image build as described in the earlier section on *image environment variables*, along with default configuration files. These defaults should be enough to launch the application into it's minimal state unaided by specifics from the user. If this is not possible then the default action of the container should be to run the application with the `--help` option to start the process of informing the user what to do next.

Memory and CPU

Runtime constraints for Memory and CPU should be specified, to ensure that an application does not exhaust host resources, or behave badly next to other co-located applications, for example with Docker:

```
docker run --rm --name postgresdb --memory="1g" --cpu-shares="1024" --cpuset-cpus="1,3
↪" -d postgres
```

In the above scenario, the PostgreSQL database would have a 1GB of memory limit before an Out Of Memory error occurred, and it would get a 100% share of CPUs 1 and 3. This example also illustrates CPU pinning.

Service Discovery

Although Container Orchestration is not covered by these standards, it is important to note that the leading Orchestration solutons (Docker Swarm, Kubernetes, Mesos) use DNS as the primary service discovery mechanism. This should be considered when designing containerised applications so that they inherrently expect to resolve dependent services by DNS, and in return expose their own services over DNS. This will ensure that when in future the containerised application is integrated as part of an Orchestrated solution, it will conform to that architecture seamlessly.

Standard input, output, and errors

Container Engines such as Docker are implemented on the fundamental premise that the containerised application behaves as a standard UNIX application that can be launched (`exec'ed`) from the commandline. Because of this, the application is expected to respond to all the standard inputs and outputs including:

- stdin
- stdout
- stderr
- signals
- commandline parameters

The primary use case for stdin is where the container is launched replacing the entry point with a shell such as `bash`. This enables a DevOps engineer to enter into the container namespace for diagnostic and debug purposes. While it is possible to do, it is not good practice to design a containerised application to read from stdin as this will make an assumption that any scheduling and orchestration service that executes the container will be able to enact UNIX pipes which is not the case.

stdout and stderr are sent straight to the Container Engine logging system. In Docker, this is the logging sub-system which combines the output for viewing purposes with `docker logs ....` Because these logging systems are configurable, and can be syndicated into unviversal logging solutions, using stdout/stderr is used as a defacto standard for logging.

Logging

The SKA has adopted *SKA Log Message Format* as the logging standard to be used by all SKA software. This should be considered a base line standard and will be decorated with additional data by an integrated logging solution (eg: ElasticStack).

The following recommendations are made:

- when developing containerised applications, the development process should scale from the individual unit on the desktop up to the production deployment. In order to do this, logging should be implemented so that stdout/stderr is used, but is configurable to switch the emission to syslog
- log formatting must adhere to *SKA Log Message Format*
- testing should include confirmation of integration with the host syslog, which is easily achieved through bind mounting `/dev/log`
- within the syslog standard, the message portion should be enriched with JSON structured data so that the universal logging solution integrated with the Container Engine and/or Orchestration solution can derive greater semantic meaning from the application logs

Sharing

Aside from communication over TCP/UDP sockets between processes, it is possible to communicate between containers in other ways, including:

- SHMEM/IPC
- Named pipes
- Shared volumes

SysV/POSIX shared memory segments, semaphores and message queues can be shared using the `--ipc=host|container-id` option for `docker run ....` However, this is specific to the runtime enviroment and the orchestration solution. The `host` option is a security risk that must be evaluated as any joining containers will be pushed into the host OS namespace.

Named pipes, are straight forward as these are achieved through shared hostpath mounts between the containers where the pipe can be created using `mkfifo`.

2.11.2 Container Standards CheatSheet

A summary of the standards to be used as a checklist.

Reference Implementation

Throughout Docker is used as the reference implementation with the canonical version being Docker 18.09.4 CE API version 1.39.

Structuring applications in a Containerised Environment

- Each containerised application should be a single discrete application.
- A containerised application should not need a specialised multi-process init process such as `supervisord` as there should not be multiple parent processes.
- Ensure that signal handling is correctly propagated from PID 1 to the containerised application so that container engine SIGHUP and SIGKILL are correctly handled.
- There should be one container image per application with one application per Git repository in order to correctly manage independent release cycles.

Defining and Building Container Images

- Containers are immutable by design - it should be possible to destroy and recreate them with (little or) no side effects.
- Do not store state inside a containerised application - always mount in storage for this purpose keeping containers ephemeral.
- Minimise the size and number of layers of the image to speed up image transfer and container launch.
- Order the layers from most static to least static to reduce churn and depth to the image rebuild process.
- All directives and key words should be in upper case.
- All element names should be in lower case - image labels and tags, and arguments (`ARG`) apart from environment variables (`ENV` - upper case).
- Liberally use comments (lines starting with `#`) to explain each step of the build and describe any external dependencies.
- Where multi-line arguments are used such as `RUN apt-get install ...`, sort them for ease of reading.
- The size of the build context should be minimised in order to speed up the build process.
- Always be careful to exclude unnecessary and sensitive files from the image build context.
- Break the build process into multiple images so that core and common builds can be shared with other applications.

- Use multi-stage builds (with `COPY --from...`) to reduce the final size of an image.
- Avoid embedding configuration and data in the container image unless it is small, guaranteed to be static, and forms sensible defaults for the running application.
- Base images and image provenance must be checked in order to maintain the security and integrity of the SKA runtime systems.
- Stable image tags should be used for base images that include the Major and Minor version number of Semantic Versioning eg: `python:3.7`.
- Avoid installing unnecessary packages in your container image.
- Create a derivative image from the standard production one explicitly for the purposes of debugging, and problem resolution.
- Always clean the package cache afterward use of `apt-get install ...` to avoid the package archives and other temporary files becoming part of the new layer.
- Order the build directives specified in the `Dockerfile`, to ensure that they are running from the lowest frequency changing to the highest to exploit the build cache.
- Use the `LABEL` directive to add metadata to your image.
- Use arguments (`ARG`) to parameterise elements such as the base image, and versions of key packages to be installed.
- Only set environment variables using `ENV` if they are required in the final image to avoid embedding unwanted data.
- Prioritise use of `RUN + curl` over `ADD/COPY + RUN` to reduce image size.
- use `USER` and `WORKDIR` to switch the user at execution time and set directory context.
- Never use sudo - there should never be a need for an account to elevate permissions.
- set `ENTRYPOINT` to the full path to the application and `CMD` to the default command line arguments.
- Use the `["thing"]` which is the `exec` notation ensuring that proper signal propagation occurs to the containerised application.
- Use the `ONBUILD` feature sparingly, as it can cause unintended consequences.

aming and Tagging

- Image names should reflect the application that will run in the resultant container eg: `tango-example/powersupply:1.13.2`.
- Images should be tagged with short commit hash as derived by `git rev-parse --verify --short=8 HEAD` from the parent Git repository.
- When an image version for an application is promoted to production, it should be tagged with the application version (using Semantic Versioning).
- For the most current major.minor.patch image version the 'latest' tag should be added.
- Application version tags should be added eg: `1.13.2`, `1.13`, `1` - where major/minor/patch version point to the latest in that series.
- A production deployment should always be made with a fully qualified semantic version eg: `tango-example/powersupply:1.13.2`.
- The SKA will endeavour to support only one OS base per image as the practice of multi-OS bases does not strictly follow Semantic Versioning, and creates considerable maintenance overhead.

- Within the SKA hosted Continuous Integration infrastructure, all production images are kept indefinitely.
- Images with debuging tools, profilers, and any tools not essential to the running of the target application should be contained in a derivative image that is named explicitly `dev` eg: `tango-example/powersupply-dev:1.13.2`.
- All images should be stored in a Docker v2 Registry API compliant repository, protected by HTTPS.
- All containerised software used within the SKA, will be served out of the hosted repository service.

Image Signing and Publishing

- All images pushed to the SKA hosted repository must be signed. This will ensure that only trusted content will be launched in containerised environments.

Image Signing and Publishing

- All images pushed to the SKA hosted repository must be signed. This will ensure that only trusted content will be launched in containerised environments.

Running Containerised Applications

- The containerised application developer must determine what **the application interface contract** based on the *touch points with resources* from the underlying host through the Container Engine.
- Usage documentation for the image must describe the intended purpose of each configured resource, how they combine and what the defaults are with default behaviours.
- Use `VOLUME` statements for all directories to be mounted as it provides annotation of the image requirements.
- When adding a volume at runtime, consider whether write access is really required - add `:ro` liberally.
- Containerised applications should avoid using `--net=host` (host only) based networking as this will push the container onto the running host network namespace monopolising any ports that it uses.
- Where possible, a containerised application should run under a specific UIG/GID to avoid privilege escalation as an attack vector.
- It should be a last resort to run the container in privileged mode `docker run --privileged ...` as this has security implications.
- Configuration of a containerised application should be managed primarily by *Environment Variables* and configuration files.
- Avoid passing large numbers of configuration options on the command line or secrets such as keys and passwords.
- Configuration passed into a container should not directly rely on a 3rd party secret/configuration service integration.
- Appropriate configuration defaults should be defined in the image build using *image environment variables*, along with default configuration files. These defaults should be enough to launch the application into it's minimal state unaided by specifics from the user.
- Runtime constraints for Memory and CPU should be specified, to ensure that an application does not exhaust host resources, or behave badly with co-located applications.

- Although Container Orchestration is not covered by these standards, it is important to note that the leading Orchestration solutons (Docker Swarm, Kubernetes, Mesos) use DNS as the primary service discovery mechanism. This should be considered when designing containerised applications so that they inherrently expect to resolve dependent services by DNS, and in return expose their own services over DNS. This will ensure that when in future the containerised application is integrated as part of an Orchestrated solution, it will conform to that architecture seamlessly.

Logging

- Stdout and stderr are sent straight to the Container Engine logging system. In Docker, this is the logging subsystem which combines the output for viewing purposes with `docker logs ....` This is used as a defacto standard for containerised application logging.
- Logging should be implemented so that stdout/stderr is used, but is configurable to switch the emission to syslog
- Logging to *stdout* or console so that the routing and handling of log messages can be handled by the container runtime (*dockerd*, *containerd*) or dynamic infrastructure platform (*Kubernetes*).
- The SKA has adopted *SKA Log Message Format* as the logging standard to be used by all SKA software.
- Within the this standard, the message portion should be enriched with JSON structured data so that the universal logging solution integrated with the Container Engine and/or Orchestration solution can derive greater semantic meaning from the application logs.

2.11.3 Working with Docker: Proxy Cache

Docker (the Corporation) has implemented download rate limiting or throttling effective from the 1st of November, 2020, the details of which can be found in the company FAQ and related technical documentation. This can create problems for heavy users of OCI Container Images hosted on the the Docker Hub public registry service, such as CI/CD pipeline build processes, and Institutional environments that aggregate their network traffic through single points of presence.

Table of Contents

The Symptoms

The result of these changes is a generic problem for any individual or organisation that were previously relying on an essentially free public OCI Container Image library service. The problem manifests itself when the introduced threshold is reached for pulling Images, the pull requests will fail and return an HTTP error code 429, where you may see logging messages like the following:

```
HTTP/1.1 429 Too Many Requests
```

The basic rules of the new thresholds (as at 15 Dec 2020) are that:

- anonymous pull requests - 100 over 6 hours
- logged in pull requests - 200 over 6 hours
- subscription thresholds - vary as selected

Mitigation

In response to this, all affected parties have a few options:

- migrate all OCI Images required to another service provider such as gcr.io (Google), quay.io (RedHat), and GitLab
- migrate to an internally hosted service such as Nexus, Harbor, or the Docker registry
- implement a caching proxy such as Nexus, Harbor, or the Docker registry
- purchase a subscription from Docker

Solution for CI/CD and Integration Testing Environments

For the supporting services hosted on the Engage SKA OpenStack cluster including the CI/CD GitLab pipelines, shared Kubernetes cluster, and supporting shared services (storage, monitoring, logging, database services), the approach has been to roll out a caching proxy integrated with any deployment of DockerD - which is effectively every VM Host in the cluster supporting these shared services.

When an OCI Image (UN-qualified URL pointing to Docker Hub) is requested eg: via the client `docker pull`, the request is first communicated with the local `dockerd` daemon (the container runtime environment). The `dockerd` consults it's local library and the pull request policy (eg: `IfNotPresent`, `Always`, `Never`), and then makes the request based on the daemon configuration. When the daemon is configured to use a cache, the request is forwarded to the cache first. If the image exists in the cache, it is returned, else the request is forwarded to Docker Hub. Docker Hub returns the result to the cache, the cache returns the result to `dockerd`, and `dockerd` notifies the original requesting client.

HowTo: Proxy Cache

To deploy a Caching proxy, there are effectively two steps:

- Deploy the caching service exposed on an addressable port (protecting this service is an exercise left for the reader)
- Configuring the client host dockerd to use the Proxy Cache

The following offers two solutions based on using the Docker supplied `registry:v2` container, and the Sonatype Nexus Repository Manager 3.

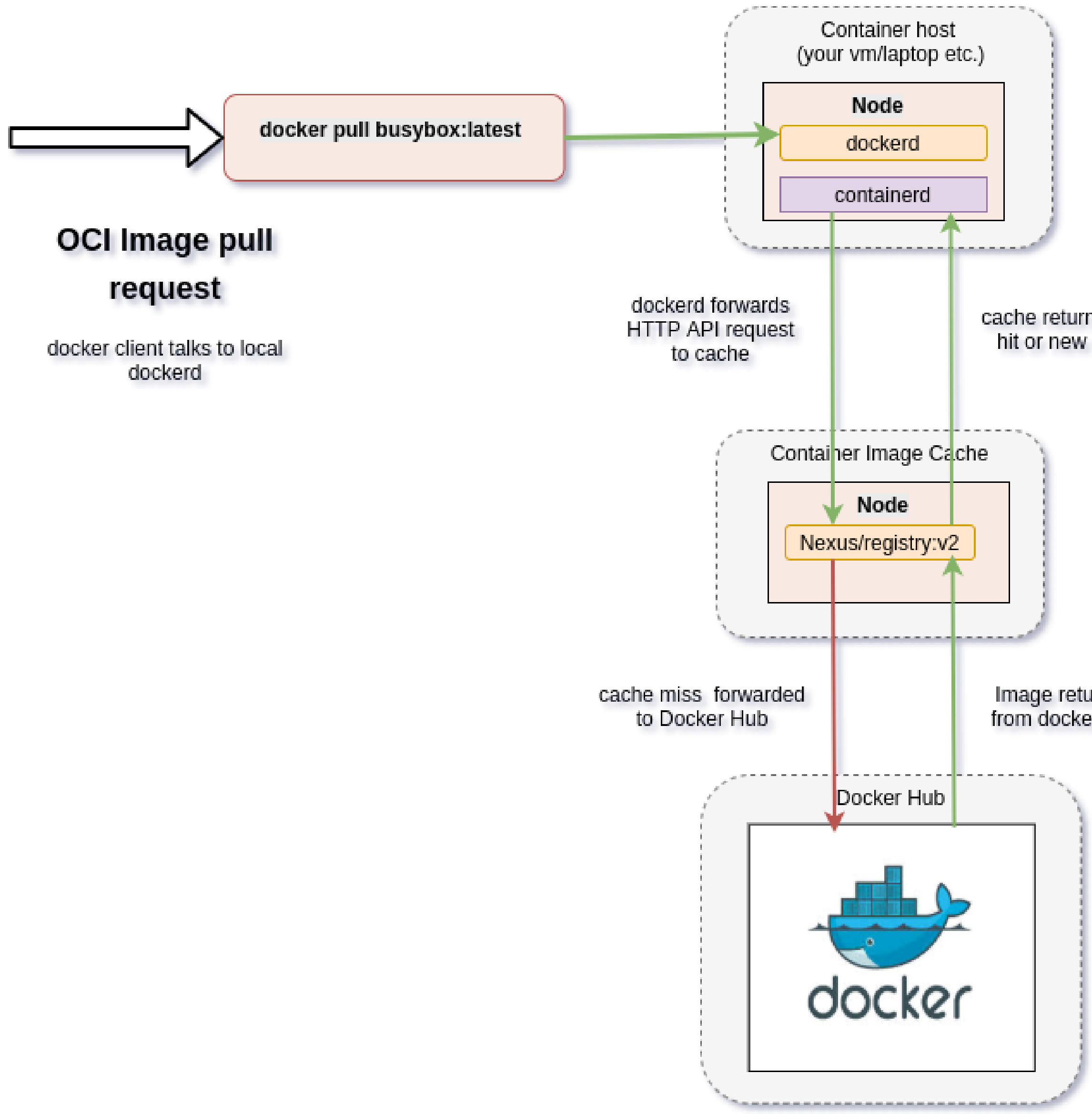

docker pull busybox:latest
OCI Image pull request
docker client talks to local dockerd
Container host (your vm/laptop etc.)
Node
dockerd
containerd
dockerd forwards HTTP API request to cache
cache returns direct hit or new cache
Container Image Cache
Node
Nexus/registry:v2
cache miss forwarded to Docker Hub
Image returned from docker hub
Docker Hub
docker

Fig. 8: Pulling an OCI Image - interaction with cache.

Caching proxy solutions

Two caching solutions are demonstrated - the first is a minimal caching proxy based on the Docker registry:v2 daemon. The second is using the Nexus Repository Manager 3 Docker registry as a caching proxy.

registry:v2

Docker provides a local private registry implementation called the `registry`. This can be converted into a caching pull-through registry by passing in the necessary configuration described below. The key things are to set the `REGISTRY_PROXY_USERNAME` variable to your Docker Hub account name, and the `REGISTRY_PROXY_PASSWORD` to a Docker Hub application security token for your account which can be configure at the address https://hub.docker.com/settings/security.

```
docker run \
-d \
-p 5000:5000 \
--restart=always \
--name=through-cache-secure \
-e REGISTRY_PROXY_REMOTEURL="https://registry-1.docker.io" \
-e REGISTRY_PROXY_USERNAME=<your docker username> \
-e REGISTRY_PROXY_PASSWORD=<registry application token> \
registry
```

This will expose the registry container on the host port `5000`, so this mirror can be referenced with `http://<ip or dns name for host>:5000`. This solution is easy to setup and is something that could be used by individuals or small collections of developers. (reference https://circleci.com/docs/2.0/docker-hub-pull-through-mirror/)

Nexus

Running Nexus (https://help.sonatype.com/repomanager3) is a valuable service as a Software Artefact Repository, but does have some overhead and disaster recovery considerations especially if it is used for more than a caching proxy (not only for OCI Images, but also for PyPi, NPM etc.). However, the initial setup for running a caching proxy is reasonably simple. The following are a guiding set of scripts and pointers to documentation to help with this process, focusing on how to establish an anonymous Docker caching proxy.

Boot the Nexus server in a docker container and mount caching space storage (from the host path `/data/nexus-data` in this example) into the container. The configuration for Nexus is updated to enable the Groovy scripting API, enable the default password, and disable the startup wizard (`${DATA}/etc/nexus.properties`).

```
#!/bin/bash

TAG=3.28.1
DATA=/data/nexus-data

docker stop --time=120 nexus || true
docker rm -f nexus|| true

sudo mkdir -p ${DATA}/etc

# write out a Nexus configuration that will allow automated setup to run
sudo cat <<EOF | sudo tee ${DATA}/etc/nexus.properties
# Jetty section
# application-port=8081
# application-host=0.0.0.0
```

(continues on next page)

(continued from previous page)

```
# nexus-args=\${jetty.etc}/jetty.xml,\${jetty.etc}/jetty-http.xml,\${jetty.etc}/jetty-
↪requestlog.xml
# nexus-context-path=/\${NEXUS_CONTEXT}

# Nexus section
# nexus-edition=nexus-pro-edition
# nexus-features=\
#  nexus-pro-feature
# nexus.clustered=false

# activate scripting
nexus.scripts.allowCreation=true

# disable the wizard.
nexus.onboarding.enabled=false

# disable generating a random password for the admin user.
# default is: admin123
nexus.security.randompassword=false

EOF
# set permissions to that expected by the container
sudo chown -R 200 ${DATA}

# launch the Nexus service
docker run -d -p 8081:8081 -p 8181:8181 --name nexus \
-v ${DATA}:/nexus-data \
--net=host \
-e INSTALL4J_ADD_VM_PARAMS="-Xms2g -Xmx2g -XX:MaxDirectMemorySize=3g " \
sonatype/nexus3:${TAG}
```

Once the Nexus server has completed starting up (use `docker logs nexus -f` to follow what has happened), you then need to run the following Groovy script (packaged inside a bash shell script) to complete the installation of setting up the Docker Proxy Repository.

```
#!/bin/bash
# Set the environment variables to your server eg: BASE_URL, and PASSWORD
set -x
BASE_URL=http://localhost:8081
GROOVY_SCRIPT=/tmp/docker-script.json
SCRIPT_NAME=dockerRepositories
USER=admin
DEFAULT_PASSWORD=admin123 # do not change - required for boot up
PASSWORD=admin # set this password to what you want it to be

# write out the Groovy script for configuring the Nexus repository manager
# note the \n's are required for JSON payloads
cat <<EOF > ${GROOVY_SCRIPT}
{
"name": "${SCRIPT_NAME}",
"content": "import groovy.json.JsonOutput\n
import org.sonatype.nexus.security.realm.RealmManager\n
import org.sonatype.nexus.blobstore.api.BlobStoreManager\n
def user = security.securitySystem.getUser('admin')\n
user.setEmailAddress('admin@example.com')\n
security.securitySystem.updateUser(user)\n
```

(continues on next page)

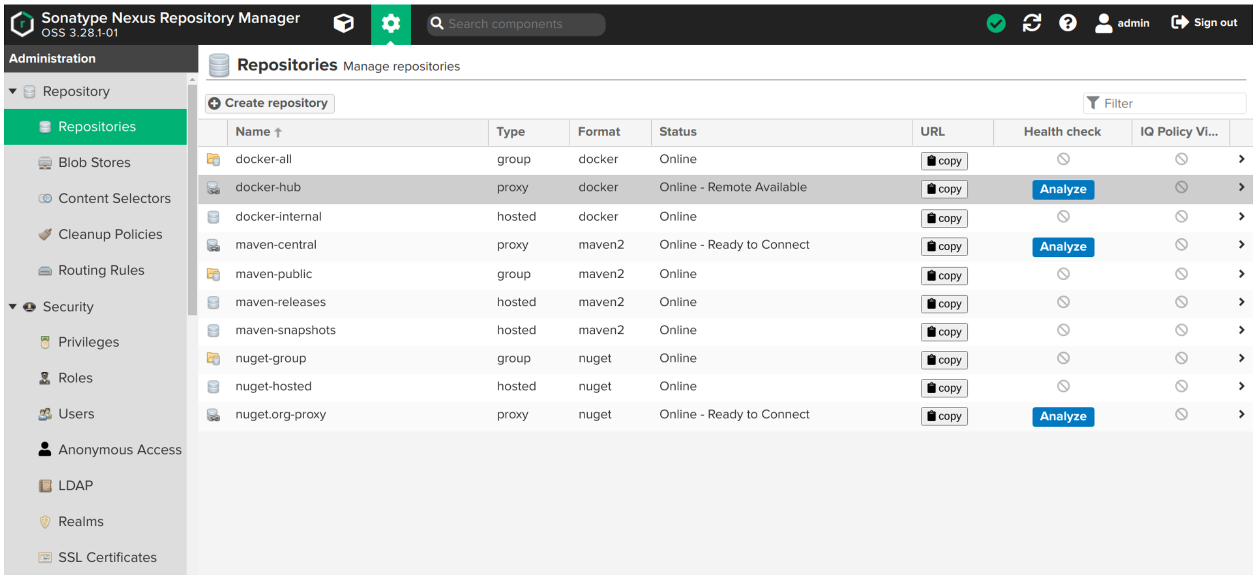
(continued from previous page)

```
security.securitySystem.changePassword('admin','${PASSWORD}')\n
log.info('default password for admin changed')\n
\n
//enable Docker Bearer Token\n
realmManager = container.lookup(RealmManager.class.name)\n
realmManager.enableRealm('DockerToken')\n
\n
//Enable anonymois access which we above disabled\n
security.anonymousAccess = true\n
security.setAnonymousAccess(true)\n
// create hosted repo and expose via https to allow deployments\n
repository.createDockerHosted('docker-internal', null, null)\n
\n
// create proxy repo of Docker Hub and enable v1 to get search to work\n
// no ports since access is only indirectly via group\n
repository.createDockerProxy('docker-hub',                   // name\n
                             'https://registry-1.docker.io', // remoteUrl\n
                             'HUB',                          // indexType\n
                             null,                           // indexUrl\n
                             null,                           // httpPort\n
                             null,                           // httpsPort\n
                             BlobStoreManager.DEFAULT_BLOBSTORE_NAME, //␣
↪blobStoreName\n
                             true, // strictContentTypeValidation\n
                             true)\n
\n
// create group and allow access via https\n
def groupMembers = ['docker-hub', 'docker-internal']\n
repository.createDockerGroup('docker-all', 8181, null, groupMembers, true,␣
↪BlobStoreManager.DEFAULT_BLOBSTORE_NAME, false)\n
log.info('Script dockerRepositories completed successfully')\n
",
"type": "groovy"
}
EOF

# upload the Groovy script
curl -v -u ${USER}:${DEFAULT_PASSWORD} -X POST --header 'Content-Type: application/
↪json' \
"${BASE_URL}/service/rest/v1/script" \
-d @${GROOVY_SCRIPT}

# run the Groovy script
curl -v -X POST -u ${USER}:${DEFAULT_PASSWORD} --header "Content-Type: text/plain" "$
↪{BASE_URL}/service/rest/v1/script/${SCRIPT_NAME}/run"
```

This shell (bash) script is an automation of the process described in this article and this help documentation.

Once the script has completed, you can then login to the Repository Manager at http://localhost:8081. Navigate to the Repository configuration screen and check that you have something like the following repositories setup.

The Docker group repository `docker-all` is now available on the mirror URL `http://localhost:8181`.



Name	Type	Format	Status	URL	Health check	IQ Policy Vi...
docker-all	group	docker	Online	copy		
docker-hub	proxy	docker	Online - Remote Available	copy	Analyze	
docker-internal	hosted	docker	Online	copy		
maven-central	proxy	maven2	Online - Ready to Connect	copy	Analyze	
maven-public	group	maven2	Online	copy		
maven-releases	hosted	maven2	Online	copy		
maven-snapshots	hosted	maven2	Online	copy		
nuget-group	group	nuget	Online	copy		
nuget-hosted	hosted	nuget	Online	copy		
nuget.org-proxy	proxy	nuget	Online - Ready to Connect	copy	Analyze	

Fig. 9: The Docker group repository.

Configuration of dockerd

The local `dockerd` agent must be configured to point the cache. This is done either in the command line arguments using the switch `--registry-mirror`, or using the daemon configuration file `/etc/docker/daemon.json`. A list of mirrors can be supplied, and they are tried in order, with the final attempt being to bypass using a cache/mirror at all.

```
{
...
"registry-mirrors" : [
    "http://192.168.178.22:8181"
],
...
}
```

Test using an image pull such as `docker pull busybox`. Check your local system logs to see whether there are any messages about skipping to the next end point. This can be found on Linux using `journalctl -f` and the messages might look like the following:

```
Dec 15 08:58:57 wattle dockerd[920572]: time="2020-12-14T19:58:57.826247120Z"␣
↪level=info msg="Attempting next endpoint for pull after error: Get http://192.168.
↪93.12:8181/v2/library/ubuntu/manifests/20.04: unauthorized: authentication required"
```

Configuration of podman

If using podman for dockerless builds then the configuration file */etc/containers/registries.conf* needs to be updated as follows:

```
unqualified-search-registries = ["docker.io"]

[[registry]]
location = "docker.io"
```

(continues on next page)

(continued from previous page)

```
[[registry.mirror]]
location = "192.168.93.12:8181"
insecure = true
```

2.11.4 A Quick Introduction to Kubernetes

This page is based on a talk given by Piers Harding in July 2020, and provides an SKA-specific overview of Kubernetes. Further details can be found in *Container Standards* and *Container Orchestration Guidelines*.

What is Kubernetes?

Kubernetes is an abstraction layer to allow you to manage containerised applications. To run an application, you need compute, network and storage, so Kubernetes defines resources that allow you to manage these things. The Kubernetes resources are a mechanism whereby you can refer to compute, network, storage in terms of abstract names, without having to address the underlying hardware directly.

These defintions work in terms of applied semantics and eventual consistency. If you create a resource such as a Pod (a Pod is the smallest unit of compute that you can manage in Kubernetes), and then update that resource, the Kubernetes system will do its best to nudge or coerce that Pod into the new state that you've defined. It works out the delta betwen the old and new resource definitions, and then tries to apply changes to update to the new state. So all you need to know is how you want the Pod to look at some point – you don't need to calculate and implement the deltas yourself.

Similarly, as part of the inbuilt health checking, Kubernetes will, if it finds a Pod in an unexpected state, try to nudge it back to the expected state. So if for some reason a node (an actual server) on your Kubernetes cluster goes away, the Pods scheduled on those nodes will be re-load-balanced onto the remaining nodes on the cluster, once the problem is detected.

Compute

From a compute point of view, the basic unit is the container, and the Pod is made up of one or more containers. Those containers in that Pod are effectively blessed into the same network namespace (the cluster namespace), and the actual process namespace can be shared as well. When a Pod is defined, kubernetes starts an initial process within the namespace, and this is the "head" of what a Pod is. So if any containers attached within this namespace fail, the whole application doesn't fall over, because that head process is guaranteed to remain. So the list of the containers you define in a Pod gets booted up in the associated namespace given to that Pod.

Quick Note on Kubernetes Namespaces

Namespacing allows you to define the scope of things, in a general computer science sense. A Kubernetes cluster has a single networking address space, to allow the Kubernetes scheduler (`kube-scheduler`) to place jobs throughout the cluster. You can also deploy Pods to a specified Namespace (a Kubernetes construct as opposed to a Linux Kernel namespace for OS level network issolation), and that Namespace is isolated from other Namespaces in the cluster. So this allows you to deploy (for example) different versions of the same container, with the same Pod and Service (akin to a DNS name) names for testing purposes.

Scheduling Compute

For a horizontally scaling process, such as a web application, you deploy more of these pods to get horizontal scaling. You define how to deploy your Pods. You can use a `Deployment` (which in turn uses a `ReplicaSet`) to define how many of these Pods you want running, and what the scheduling constraintes are – such as where in the cluster you wish to permit them to run or not run. (A good use case here is if you have some parts of your application GPU-enabled, you want those Pods to run on GPU nodes, and the non-GPU parts of the application to run on nodes that don't have GPUs.) Kubernetes then takes that ReplicaSet definition and figures out where in the cluster it can run, boots up the Pods, and sorts out the networking between the Pods.

As well as Deployments, we have StatefulSets. This is essentially a specialisation of a Deployment. It guarantees the ordering and naming of the individual Pods within the set. So if you have a StatefulSet with replicas set to >1, the Pods will have an incremental number attached to the name you gave the StatefulSet (so my-app01, my-app02, my-app03. . .). These are guaranteed network stability, so within the cluster, they are guaranteed to keep their IP address when restarted. This is ideal for things like databases and services where you want stability (i.e. where you wish to maintain state!)

The other major thing guaranteed by StatefulSets is that if you scale them up and down, they'll behave like a popping stack. So having created Pods 01, 02, 03. . . , 03 is removed first. With a Deployment, order is not meant to be important, so if you scale down, Kubernetes will remove a Pod from the Deployment as it sees fit.

Then there are DaemonSets. These are very good for some of the key OS/unix style services you want to emulate in the cluster. A DaemonSet will guarantee to have onely one of these scheduled on each physical node within the cluster, based on scheduling rules and affinity rules. (So you can deploy DaemonSets to a subset of your cluster if you wish.) This is particularly good for when you need some specific plugin service running, such as logging or monitoring; thus you can use a Daemonset to get logs from each host on the cluster.

The last scheduling solution is a Job. Jobs are one-shot pods that run to completion. You can basically allocate Pods in the same way as sets – you can create and tear down services also. The Pod can contain any number of containers, and the Pod will run to completion. A 0 exit code indicates success. All other deployment types expect to be up at all times, and they don't complete unless there's an error, in which case the error will be caught by the health monitoring. You can also cron your Jobs (just like on Unix), so they run at a regular cadence.

We have some configurator jobs to set up services, such as a Job to set up the Tango database server for TMC.

Note that failed Pods aren't immediately garbage collected. In a production environment, you set rules that mean Pods are garbage collected after 24 hours, so you have a chance to grab the Pod logs (if it's not already collected by your logging system), and then you might see why the Pod failed.

Beware! Some Pods return an exit code of 2 on successful update. This is incorrect from a Unix point of view. We've tried to wrap and catch it in most places (eg: `dsconfig`).

With skampi at the moment there are complex dependencies and some pods needs to come up in a particular order. At the moment, we're using a StatefulSet to enforce ordering.

Networking

So you've deployed a lot of Pods to the cluster to compute somthing. How do you expose these Pods you've scheduled to the cluster, so they can talk to each other?

There are two main methods for doing this: a Service resource and an Ingress resource. A Service is a way of exposing particular services, load-balanced across the associated Pods, usually related to a single deployment or StatefulSet (or even a DaemonSet).

A Service specifies a to-and-from port (or a series of them) that is to be exposed, translating the port(s) that are open to data on the associated Pods, and associating it with a Pod on the Service. The way it works out which Pods to provide this service for is based on label matching. So you need to get this right, otherwise you could mash together two different deployments! So you need to label your Pods and Sets properly, otherwise your Service will break.

Services help translate requests (such as a database request or a push to a message queue) in to the internal Kubernetes DNS for the cluster. The first scope is the namespace of the cluster you're running in. This becomes the first level of the DNS name. This first level is the Service name itself. So if you're running Jupyter, you'll define (say) a ReplicaSet and put a Service resource in front of it. That Service maps the ports of the jupyter web service listening within the

Pods to a port and IP it exposes. You might name the Service `jupyter` and if it's running the in `j-hub` namespace, inside the cluster it's now referred to as `jupyter.j-hub`. This is the primary method of service discovery within the Kubernetes cluster.

A Pod *can* communicate with another pod without putting a service in front of it, but there's no stability in the reference names. You might have deployed a bare Pod yourself, outside a ReplicaSet. That does have a fixed name, but then there's no health checking or monitoring associated with that bare Pod, and thus there's no auto-healing. So to get auto-healing *and* reference stability, you need a ReplicaSet/some other schedulable deployment mechanism, and a Service. Bare Pods are vigorously discouraged in Kubernetes as an anti-pattern.

Because there are no guarantees of naming within Pods, the labelling schemes allow Services to provide a bridge between the Pods and the fixed IP front end within the cluster. Typically, a Service will have a cluster IP address (there are other ways of doing it, but we'll stick to this method) which you can use to communicate with the Pods managed by the Service. This address is resolved by DNS (Domain Name Services) within the cluster. On top of this you get load-balancing schemes, such as random, round-robin, or even sticky (this isn't usually a good idea, but may be needed for some legacy applications like R ShinyApps).

The Service load-balancer means that if a Pod fails, the Kubernetes components in the Service will automatically notice (via health checking) and drop it from the load balander, so you don't get dead endpoints.

However, Services are primarily for communication within the cluster. So to communicate with the outside world (whether the internet, a VPN, basically anything that isn't your Kubernetes cluster), you need an Ingress Controller. An Ingress Controller is a point of entry or exit to the outside world within the cluster. You do a further mapping exercise based on the Service name and port names together with the URI to define which Services within the cluster should be exposed to the outside world, and how they should be exposed. This is HTTP-based. For TCP based applications, a Service type of Loadbalancer can be used to create a network mapping from the internally running Pods to the outside world. However this is only possible if the underlying Kubernetes infrastructure supports these kinds of Load Balancers eg: OpenStack Octavia.

How does this impact latency? Historically, this was done with `iptables` rules (rules that control communication and routing for the Pod network), and is moving to IPVS (basically, like iptables, but faster). This is because the iptables rulesets get very big. The bigger the cluster, and the more stuff you're running, the more enormous they get.

Mostly, the cluster network is controlled by 3rd party solutions. We're using Calico, which works quite efficiently with a flat network (it is configured to bind at Layer2). These solutions mean that if you try to route between two Pods on the same node, the iptables should encode this, and make sure that the route between those two Pods never goes off-node. If there's a hop between hosts, iptables should also encode this. Calico is an intelligent routing service, and it will route in the most efficient way it can. It's referred to as an overlay network.

If latency is a problem, you can use affinity rules to place the Pods on the same node. This is most important if you don't have control over node placement (e.g. when you're working on the public cloud). From a Pod perspective, it's dealing with a local subnet in the data centre. So in one way, the Pod is its own little computer, with a network, compute, and storage – hence the comparisons between pods, containers, and VMs.

Storage

A PersistentVolume is an abstraction from the actual physical implementation of the underlying storage solution. This abstraction is manifested though StorageClass names. So when you create a PersistentVolumeClaim, you specify a StorageClass, which is an abstract concept and the underlying storage engine will go away and allocate that piece of storage and then mount it wherever it needs to be (Node). Then the Pod which wants to use that storage can find it and access it as a filesystem.

The StorageClasses can have different characteristics. So within the syscore of the Kubernetes cluster on EngageSKA, we have two fundamental storage types. One is block, and you can only mount that for *What is read-write once versus read-write many?*, so its primary use case would be running something like a database, and then you write to storage from within the database engine.

The other storage type we're supporting is NFS (network file system). Currently, we implement both types using Ceph, but there's nothing stopping us taking our deployment to AWS or the Google Cloud Platform, and using their storage solutions and creating alias StorageClass names for NFS and block, and deploying our workloads there. We wouldn't have to change our resource descriptions, because we have this abstraction between what we call storage and the characteristics we want to have for that type of storage, and how it's actually physically implemented on the platform.

Some older storage engines require you to define a PersistentVolume, which is a low-level addressing of a lump of storage (e.g. a StorageClass). A PersistentVolumeClaim is a claim to mount that lump of storage which turns the abstract StorageClass into a reality. This may not have been the best idea. The Ceph implementation doesn't use the PersistentVolume concept at all – you just describe a PersistentVolumeClaim, declaring the StorageClass you want and how much storage you want there to be, and basically it does it all in one operation.

Then the volume you've created becomes available to the Pod. So the NFS storage is read-write many. That's ideal for web-based or horizontally scaling applications, where you need many instances of the application running, all needing concurrent access to that storage to read-write (like they all need to access the content for web pages). Block storage gives raw access, NFS is through a posix-style interface. So there are tradeoffs to the different types of storage and performance characteristics.

What is read-write once versus read-write many?

Read-write once/many refers to the number of times you can mount that piece of storage into a running container. So for a database, it makes sense to mount that storage once, to the container running the database engine. Read-write many means that multiple containers can mount the storage, so you can have multiple Pods all reading out content for your web page, for example.

Resource Management

We can put limits on CPU, memory, and storage, so that we can control resource usage across the cluster.

At the Pod level, you can set two things: a request, and a limit. Request is usually set to lower than the limit. The request is what you expect the Pod to need in normal usage – i.e. the Pod's normal consumption of resources – and the limit is the upper bound. If the Pod hits the limit, you expect that there is something wrong, and it's thus a Pod health issue. So if the Pod exceeds those limits, the Kubernetes scheduler would mark the Pod for eviction and then evict it. The kubelet on each host monitors this. When the kubelet gets the scheduling requirements from the kube-controller, it knows what the limits are for the Pod it's about to launch, and then it monitors that Pod.

Eviction doesn't happen instantaneously. There are global policies about when something gets evicted. But things that do exceed their resource limits will get evicted in a certain amount of time. We do have monitoring, so you can look at the resources your Pod is using.

2.11.5 Multitenancy

The SKA adopts Kubernetes as the orchestration layer. The Kubernetes Clusters Managed by the Systems Team serve multiple projects and can host multiple deployments of the same project at the same time. Those clusters are thus configured to ensure that each job runs isolated from the others and that the cluster resources are fairly allocated among the different jobs. This is what is called multitenancy.

Multitenancy implemented at the Namespace level

An important characteristic of a Namespace in a Kubernetes Cluster is that it is isolated from other Namespaces in the cluster. This means that you can isolate each separate job and its Kubernetes resources by deploying to a different Namespace. Within a given Namespace we constrain resource usage by defining Limit Ranges and Resource Quotas.

A Resource Quota constrains aggregate resource usage. It can limit:

- the quantity of objects that can be created in a namespace by type
- the total amount of compute resources available in that namespace including:
  - total cpu
  - total memory
  - total ephemeral storage

An example of the content of a `yaml` file defining Resource Quotas is:

ResourceQuota description

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    pods: "250"
    requests.cpu: 12000m
    requests.memory: 16Gi
    requests.ephemeral-storage: 25Gi
    limits.cpu: 24000m
    limits.memory: 32Gi
    limits.ephemeral-storage: 50Gi
```

Enabling a Resource Quota in a namespace means that users **must** specify requests and limits for all the containers inside the pods deployed in that Namespace. If requests or limits are not defined in the Helm Charts used for deployment, we can nonetheless prevent the quota system from rejecting pod creation by specifying default values at the container level, the Limit Ranges. An example of the content of a `yaml` file defining Limit Ranges for a Namespace is:

LimitRange description

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limit-range
spec:
  limits:
  - default:
      memory: 256Mi
      cpu: 200m
      ephemeral-storage: 256Mi
    defaultRequest:
      memory: 64Mi
      cpu: 65m
      ephemeral-storage: 256Mi
    type: Container
```

If requests and limits are defined in the Helm Charts used for deployment they will override the values in the Limit Ranges for those containers.

Currently in the Kubernetes Clusters maintained by the Systems Team multitenancy is implemented only in the SKAMPI pipelines, but the scripts developed for the SKAMPI project can be easily adapted for use with other pipelines. Multitenancy is implemented for SKAMPI not only in the *permanent* namespaces used to run the integration and staging environments, but also on the temporary pipelines used in feature branch development.

Access Pipeline Namespaces

Branch pipelines in the Kubernetes Clusters maintained by the System Team are short lived; they are erased after 24 hours. Also, they are named automatically and as such users must be aware of the naming scheme. The name for the pipeline Namespace is of the form `ci-<project name>-<branch name>`. For SKAMPI a `-low` or a `-mid` is appended at the end of the name depending on the telescope. For example, for a SKAMPI project branch named *at-51* and for a deployment involving the MID telescope the corresponding Namespace name would be `ci-skampi-at-51-mid`. We note that it is important to keep branch names reasonably short since Kubernates truncates Namespace names at 63 characters.

Note: It is important for users to develop appropriately for the multitenant environment. The Helm Charts used for deploying SKAMPI should avoid accessing resources such as Ingresses, PersistentVolumes, and CustomResourceDefinitions in other namespaces. Cluster globals should also be avoided, for example ingress hostnames should be globally unique. This can be achieved by using a url which includes the namespace designation.

`url: "http://$INGRESS_HOST/ci-$CI_PROJECT_NAME/taranta"` is not multi-tenant, all namespaces would share the same url. `url: "http://$INGRESS_HOST/ci-$CI_PROJECT_NAME-$CI_COMMIT_BRANCH-mid/taranta"` insures multitenancy.

Multitenancy of the branch pipelines allows for the owners of a given job to access logs, investigate problems, test things, without worrying that the performance of other jobs running in the cluster is affected. In order to achieve this users need to be able to retrieve a kubeconfig file giving access to the cluster. Such a file is generated automatically by the pipelines running on SKAMPI providing access only to the namespace specific for that pipeline, thus assuring that users will not interfere with other jobs running in the cluster.

Retrieving the kubeconfig file is easy, you'll see a `curl` in the job output in gitlab towards the end:

```
Example:

You can get the kubeconfig file from the url:
"https://nexus.engageska-portugal.pt/repository/k8s-ci-creds/ci-skampi-st-559-publish-
↪credentials-low"
with the following command into your current directory in a file called KUBECONFIG:
       curl https://nexus.engageska-portugal.pt/repository/k8s-ci-creds/ci-skampi-st-
↪559-publish-credentials-low --output KUBECONFIG
```

Once this file is copied to your local machine, and the adequate enviroment variables are set you should be able to access the namespace within the kubernetes cluster. A more detailed description on how this is implemented in the pipeline and how it works is found in the README file at the SKAMPI project repository https://gitlab.com/ska-telescope/skampi/-/blob/master/README.md

Assumptions/Additional Notes

- `SERVICE_ACCOUNT` and `KUBE_NAMESPACE` variables must be set.
- `CI_PROJECT_NAME` and `CI_COMMIT_BRANCH` variables must be accessible. Note: These are already available in gitlab pipelines.
- The namespaces are deleted 24 hours after they are created hence the kubeconfig is only valid for 24 hours
- The namespaces are deleted if there is a recent commit on the branch; the previous namespaces for the same branch/MR are deleted so that there is only one namespace which is pointing to the recent commit in the branch

2.11.6 Container Orchestration Guidelines

This section describes a set of standards, conventions and guidelines for deploying application suites on Container Orchestration technologies.

Table of Contents

Overview of Standards

These standards, best practices and guidelines are based on existing industry standards and tooling. The main references are:

- Cloud Native Computing Foundation.
- Docker v2 Registry API Specification.
- Container Network Interface.
- Container Storage Interface.

- Open Container Initiative image specification.
- Open Container Initiative run-time specification.

The standards are broken down into the following areas:

- Structuring application suites for orchestration - general guidelines for breaking up application suites for running in a container orchestration
- Defining and building cloud native application suites - resource definitions, configuration, platform resource integration
- Kubernetes primitives - a more detailed look at key components: Pods, Services, Ingress
- Scheduling and running cloud native application suites - scheduling, execution, monitoring, logging, diagnostics, security considerations

Throughout this documentation, Kubernetes in conjunction with Helm is used as the reference implementation with the canonical versions being Kubernetes v1.16.2 and Helm v3.1.2, however the aim is to target compliance with the OCI specifications and CNF guidelines so it is possible to substitute in alternative Container Orchestration solutions, and tooling.

A set of example Helm Charts are provided in the repository container-orchestration-chart-examples. These can be used to get an overall idea of how the components of a chart function together, and how the life cycle and management of a chart can be managed with `make`.

Structuring application suites for Orchestration

In order to understand how to structure applications suites for orchestration, we first need to understand what the goals of Cloud Native software engineering are.

what is Cloud Native

It is the embodiment of modern software delivery practices supported by tools, frameworks, processes and platform interfaces.

These capabilities are the next evolution of Cloud Computing, raising the level of abstraction for all actors against the architecture from the hardware unit to the application component.

What does this mean? Developers and system operators (DevOps) interface with the platform architecture using abstract resource concepts, and should have next to no concern regarding the plumbing or wiring of the platform, while still being able to deploy and scale applications according to cost and usage.

Cloud Native exploits the advantages of the Cloud Computing delivery model:

- PaaS (Platform as a Service) layered on top of IaaS (Infrastructure as a Service)
- CI/CD (Continuous Integration/Delivery) – fully automated build, test, deploy
- Modern DevOps – auto-scaling, monitoring feedback loop to tune resource requirements
- Software abstraction from platform compute, network, storage
- Portability across Cloud Services providers

Why Cloud Native SDLC (Software Development Life Cycle)?

Kubernetes provides cohesion for distributed projects:

- Codify standards through implementing testing gates
- Ensures code quality, consistency and predictability of deployment success – CI/CD

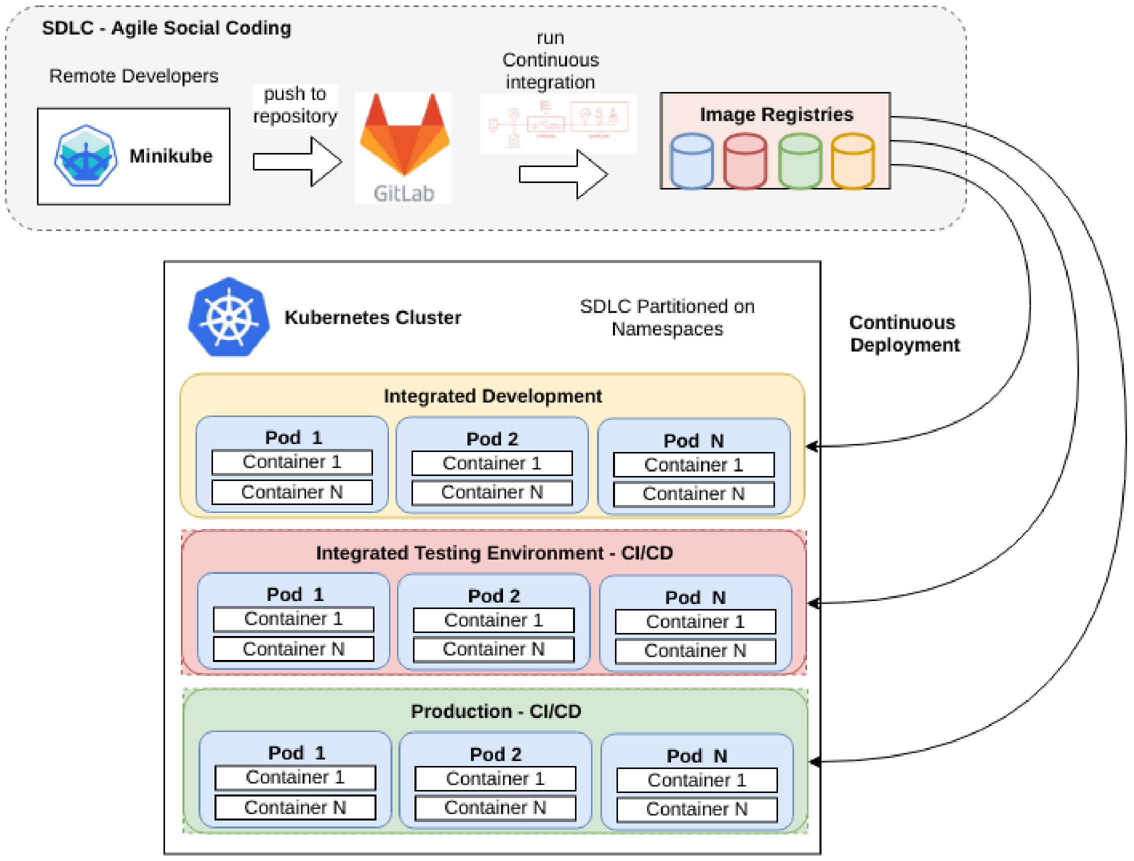


Fig. 10: How Kubernetes fits into the Cloud Native SDLC

- Automation – build AND rebuild for zero day exploits at little cost
- Portability of SDI (Software Defined Infrastructure) as well as code
- Provides a codified reference implementation of best practices, and exemplars
- Enables broad engagement – an open and collaborate system - a “Social Coding Platform”
- Consistent set of standards for integration with SRC (SKA Regional Centres), and other projects – the future platform of integrated science projects through shared resources enabled by common standards

How does orchestration work

At the core of Cloud Native is the container orchestration platform. For the purposes of these guidelines, this consists of Kubernetes as the orchestration layer, over Docker as the container engine.

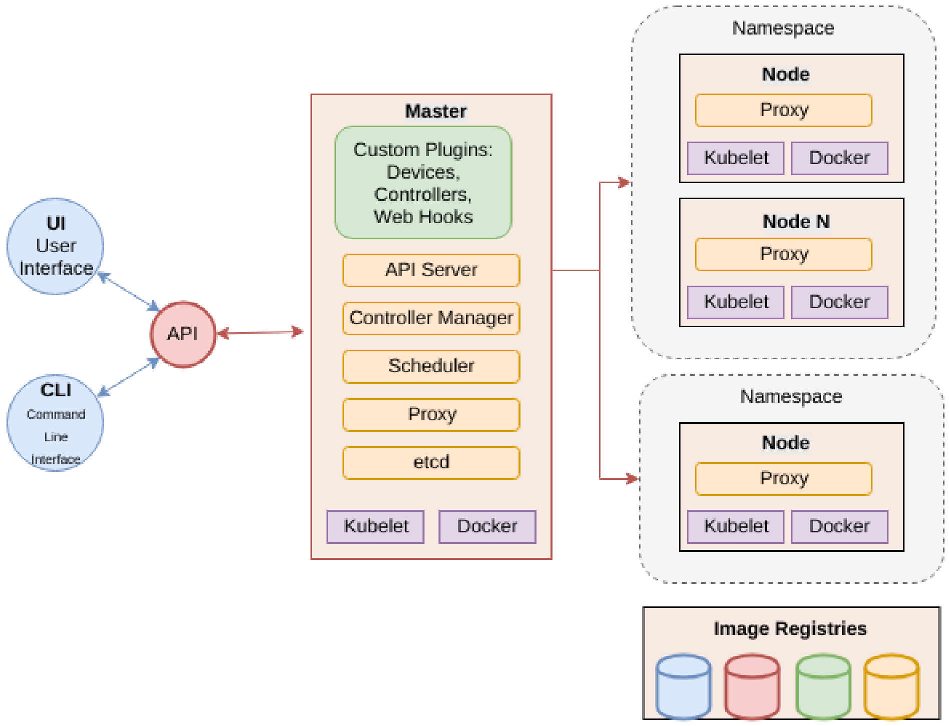


Fig. 11: The architecture of Kubernetes at the centre of the Cloud Native platform

Kubernetes provides an abstraction layer from hardware infrastructure resources enabling compute, network, storage, and other dependent services (other applications) to be treated as abstract concepts. A computing cluster is not a collection of machines but instead is an opaque pool of resources, that are advertised for availability through a consistent REST based API. These resources can be customised to provide access to and accounting of specialised devices such as GPUs.

Through the Kubernetes API, the necessary resources that make up an application suite (compute, network, storage) are addressed as objects in an idempotent way that declares the desired state eg: this number of Pods running these containers, backed by this storage, on that network. The scheduler will constantly move the cluster towards this desired state including in the event of application or node/hardware failure. This builds in robustness and auto-healing. See *A Quick Introduction to Kubernetes* for a general introduction.

Both platform and service resources can be classified by performance characteristics and reservation criteria using labelling, which in turn are used by scheduling algorithms to determine optimum placement of workloads across the cluster. All applications are deployed as sets of one or more containers in a minimum configuration called a Pod.

Pods are the minimum scalable unit that are distributed and replicated across the cluster according to the scheduling algorithm. A Pod is essentially a single Kernel namespace holding one or more containers. It only makes sense to put together containers that are essentially tightly coupled and logically indivisible by design. These Pods can be scheduled in a number of patterns using Controllers (full list) including bare Pod (a single Pod instance), Deployment (a replicated Pod set), StatefulSet (a Deployment with certain guarantees about naming and ordering of replicated units), DaemonSets (one Pod per scheduled compute node), and Job/CronJob (run to completion applications).

A detailed discussion of these features can be found in the main Kubernetes documentation under Concepts.

Structuring Application Suites

Architecting software to run in an orchestration environment builds on the guidelines given in the *Container Standards 'Structuring Containerised Applications'* section. The key concepts of treating run time containers as immutable and atomic applications where any application state is explicitly dealt with through connections to storage mechanisms, is key.

The application should be broken into components that represent:

- an application component has an independent development lifecycle
- individual process that performs a discrete task such as a micro service, specific database/web service, device, computational task etc.
- component that exposes a specific service to another application eg. a micro service or database
- a reusable component that is applicable to multiple application deployments eg. a co-routine or proximity depdendent service (logger, metrics collector, network helper, private database etc)
- an independently scalable unit that can be replicated to match demand
- the minimum unit required to match a resource profile at scheduling time such as storage, memory, cpu, specialised device

Above all, design software to scale horizontally through a UNIX process model so that individual components that have independent scaling characteristics can be replicated independently.

The application interface should be through the standard *container run time* interface contract:

- inputs come via a configurable Port
- outputs go to a configurable network service
- logging goes to stdout/stderr and syslog and uses JSON to enrich metadata (see *Container Standards 'Logging'*)
- metrics are advertised via a standard such as Prometheus Exporters, or emit metrics in a JSON format over TCP consumable by ETL services such as LogStash
- configuration is passed in using environment variables, and simple configuration files (eg: ini, or key/value pairs).
- POSIX compliant storage IO is facilitated by bind mounted volumes.
- connections to DBMS, queuing technologies and object storage are managed through configuration.
- applications should have builtin recoverability so that prior state and context is automatically discovered on restart. This enables the cluster to auto-heal by re-launching workloads on other resources when nodes fail (critical aspect of a micro-services architecture).

By structuring an application in this fashion, it can scale from the single instance desktop development environment up to a large parallel deployment in production without needing to have explicit understanding builtin for the plumbing and wiring of each specific environment because this is handled through external configuration at the Infrastructure management layer.

Example: Tango Controls

To help illustrate the Cloud Native application architecture concepts, a walk through of a Tango application suite is used.

A Tango Controller System environment is typically made up of the following:

- Database containing the system state eg: MySQL.
- DatabaseDS Tango device server.
- One or more Tango devices.
- Optional components - Tango REST interface, Tango logviewer, SysAdmin and debugging tools such as Astor and Jive.

These components map to the following Kubernetes resources:

- MySQL Database == StatefulSet.
- DatabaseDS == Deployment or StatefulSet.
- Tango REST interface == Deployment.
- Tango Device == bare Pod, or single replica Deployment.

This example does not take into consideration an HA deployment of MySQL, treating MySQL as a single instance StatefulSet. Using a StatefulSet in this case gives the following guarantees above a Deployment:

- Stable unique network identifiers.
- Stable persistent storage.
- Ordered graceful deployment and scaling.
- Ordered automated rolling updates.

These characteristics are useful for stable service types such as databases and message queues.

DatabaseDS is a stateless and horizontally scalable service in it's own right (state comes from MySQL). This makes it a fit for the Deployment (which in turn uses a ReplicaSet) or the StatefulSet. Deployments are a good fit for stateless components that require high availability through mechanisms such as rolling upgrades.

The Tango Devices are single instance applications that act as a proxy between the 'real' hardware being controlled and the DatabaseDS service that provides each Tango Device with a gateway to the Tango cluster state database (MySQL). Considering that in most cases, an upgrade to a Device Pod is likely to be a delete and replace, we can use the simplest case of a bare Pod which will enable us to name each Pod after it's intended device without the random suffix generated for Deployments.

Example: MPI jobs

A typical MPI application consists of a head node, and worker nodes with the (run to completion) job being launched from the head node, which in turn controls the work distribution over the workers.

This can be broken in to:

- a generic component type that covers head node and worker nodes.
- a launcher that triggers the application on the designated head node.

These components map to the following Kubernetes resources:

- Worker node == DaemonSet or StatefulSet.
- Launcher and Head node == Job.

MPI jobs typically only require a single instance per physical compute node, and this is exactly the use case of DaemonSets where Kubernetes ensures exactly one instance of a Pod is running on each designated node. Using Jobs enables the launcher and the head node to be combined. Both Job and DaemonSet Pods will most likely need the same library and tools from MPI, so can be combined into a single container image.

Linking Components Together

Components of an application suite or even between suites should use DNS for service discovery. This is achieved by using the Service resource. Services should always be declared before Pods so that the automatic generation of associated Environment Variables happens in time for the subsequent Pods to discover them. Service names are permanent and predictable, and are tied to the Namespace that a application suite is deployed in, for example in the namespace `test`, the DatabaseDS Tango component can find the MySQL database `tangodb` using the name `tangodb` or `tangodb.test` which is distinctly different to the instance running in the `qa` namespace also named `tangodb` but addressable by `tangodb.qa`. This greatly simplifies configuration management for software deployment.

Defining and building cloud native application suites

All Kubernetes resource objects are described through the REST based API. The representations of the API documents are in either JSON or YAML, however the preference is for YAML as the description language as this tends to be more human readable. The API representations are declarative, specifying the end desired state. It is up to the Kubernetes scheduler to make this a reality.

It is important to use generic syntax and Kubernetes resource types. Specialised resource types reduce portability of resource descriptors and templates, and increase dependency on 3rd party integrations. This could lead to upgrade paralysis because the SDLC is out of our control. An example of this might be using a non-standard 3rd party Database Operator for MySQL instead of the official Oracle one.

Metadata

Each resource is described with:

- apiVersion - API version that this document should invoke
- kind - resource type (object) that is to be handled
- metadata - descriptive information including name, labels, annotations, namespace, ownership, references
- spec(ification) - the body of the specification for this resource type denoted by *kind*

The following is an example of the start of a StatefulSet for the Tango DatabaseDS:

Resource description

```
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: databaseds-integration-tmc-webui-test
  labels:
    app.kubernetes.io/name: databaseds-integration-tmc-webui-test
    helm.sh/chart: integration-tmc-webui-0.1.0
    app.kubernetes.io/instance: test
    app.kubernetes.io/managed-by: helm
spec:
  ...
```

Namespaces

Even though it is possible to specify the namespace directly in the Metadata, it **SHOULD NOT** be, as this reduces the flexibility of any resource definition and templating solution employed such as Helm. The namespace can be specified at run time eg: `kubectl --namespace test apply -f resource-file.yaml`.

Name and Labels

Naming and labelling of all resources associated with a deployment should be consistent. This ensures that deployments that land in the same namespace can be identified along with all inter-dependencies. This is particulaly useful when using the `kubectl` command line tool as label based filtering can be employed to sieve out all related objects.

Labels are entirely flexible and free form, but as a minimum specify:

- the `name` and `app.kubernetes.io/name` with the same identifier with sufficient precision that the same application component deplyed in the same namespace can be distinguished eg: a concatenation of <application>-<suite>-<release>. `name` and `app.kubernetes.io/name` are duplicated because label filter interaction between resources relies on labels eg: `Service` exposing `Pods` of a `Deployment`.
- the labels of the deployment suite such as the `helm.sh/chart` for Helm, including the version.
- the `app.kubernetes.io/instance` (which is `release`) of the deployment suite.
- `app.kubernetes.io/managed-by` what tooling is used to manage this deployment - most likely `helm`.

Optional extras which are also useful for filtering are:

- `app.kubernetes.io/version` the component version.
- `app.kubernetes.io/component` the component type (most likely related to the primary container).
- `app.kubernetes.io/part-of` what kind of application suite this component belongs to.

The recommended core label set are described under Kubernetes common labels.

```
metadata:
  name: databaseds-integration-tmc-webui-test
  labels:
    app.kubernetes.io/name: databaseds-integration-tmc-webui-test
    helm.sh/chart: integration-tmc-webui-0.1.0
    app.kubernetes.io/instance: test
    app.kubernetes.io/version: "1.0.3"
    app.kubernetes.io/component: databaseds
    app.kubernetes.io/part-of: tango
    app.kubernetes.io/managed-by: helm
```

Using this labelling scheme enables filtering for all deployment related objects eg: `kubectl get all -l helm.sh/chart=integration-tmc-webui-0.1.0,app.kubernetes.io/instance=test`.

kubectl label filtering

```
$ kubectl get all,configmaps,secrets,pv,pvc -l helm.sh/chart=integration-tmc-webui-0.
↪1.0,app.kubernetes.io/instance=test
NAME                                           READY   STATUS     RESTARTS   AGE
pod/databaseds-integration-tmc-webui-test-0    1/1     Running    0          55s
pod/rsyslog-integration-tmc-webui-test-0       1/1     Running    0          55s
pod/tangodb-integration-tmc-webui-test-0       1/1     Running    0          55s
pod/tangotest-integration-tmc-webui-test       1/1     Running    0          55s
pod/webjive-integration-tmc-webui-test-0       0/6     Init:0/1   0          55s
```

(continues on next page)

(continued from previous page)

```

NAME                                            TYPE        CLUSTER-IP    EXTERNAL-IP␣
↪  PORT(S)                                       AGE
service/databaseds-integration-tmc-webui-test   ClusterIP   None          <none>       ␣
↪  10000/TCP                                     55s
service/rsyslog-integration-tmc-webui-test      ClusterIP   None          <none>       ␣
↪  514/TCP,514/UDP                               55s
service/tangodb-integration-tmc-webui-test      ClusterIP   None          <none>       ␣
↪  3306/TCP                                      55s
service/webjive-integration-tmc-webui-test      ClusterIP   10.97.135.8   <none>       ␣
↪  80/TCP,5004/TCP,3012/TCP,8080/TCP,27017/TCP   55s

NAME                                                     READY   AGE
statefulset.apps/databaseds-integration-tmc-webui-test   1/1     55s
statefulset.apps/rsyslog-integration-tmc-webui-test      1/1     55s
statefulset.apps/tangodb-integration-tmc-webui-test      1/1     55s
statefulset.apps/webjive-integration-tmc-webui-test      0/1     55s

NAME                                                  CAPACITY   ACCESS MODES  ␣
↪RECLAIM POLICY   STATUS   CLAIM                                          STORAGECLASS␣
↪  REASON   AGE
persistentvolume/rsyslog-integration-tmc-webui-test   10Gi       RWO           ␣
↪Retain           Bound    default/rsyslog-integration-tmc-webui-test   standard     ␣
↪           56s
persistentvolume/tangodb-integration-tmc-webui-test   1Gi        RWO           ␣
↪Retain           Bound    default/tangodb-integration-tmc-webui-test   standard     ␣
↪           55s
persistentvolume/webjive-integration-tmc-webui-test   1Gi        RWO           ␣
↪Retain           Bound    default/webjive-integration-tmc-webui-test   standard     ␣
↪           55s

NAME                                                       STATUS   VOLUME             ␣
↪                    CAPACITY   ACCESS MODES   STORAGECLASS   AGE
persistentvolumeclaim/rsyslog-integration-tmc-webui-test   Bound    rsyslog-
↪integration-tmc-webui-test   10Gi       RWO            standard       56s
persistentvolumeclaim/tangodb-integration-tmc-webui-test   Bound    tangodb-
↪integration-tmc-webui-test   1Gi        RWO            standard       55s
persistentvolumeclaim/webjive-integration-tmc-webui-test   Bound    webjive-
↪integration-tmc-webui-test   1Gi        RWO            standard       55s
```

Templating the Application

While it is entirely possible to define all the necessary resources for an application suite to be deployed on Kubernetes in individual or a single YAML file, this approach is static and quickly reveals it's limitations in terms of creating reusable and composable application suites. This is where Helm Charts have been adopted by the Kubernetes community as the leading templating solution for deployment. Helm provides a mechanism for generically describing an application suite, separating out configuration, and rolling out deployment releases all done in a declarative 'configuration as code' style. All Helm Charts should target a minimum of three environments:

- Minikube - the standalone developer environment.
- CI/CD - the Continuous Integration testing environment which is typically the same benchmark as Minikube.
- Production Cluster - the target production Kubernetes environment.

Minikube should be the default target environment for a Chart, as this will have the largest audience and should be optimised to work without modification of any configuration if possible.

When designing a Chart it is important to have clear separation of concerns:

- the application - essentially the containers to run.
- configuration - any variables that influence the application run time.
- resources - any storage, networking, configuration files, secrets, ACLs.

The general structure of a Chart should follow:

```
charts/myapp/
        Chart.yaml          # A YAML file containing information about the chart and␣
↪listing
                            # dependencies for the chart (refer to Helm 2 vs Helm 3␣
↪differences).
        LICENSE             # OPTIONAL: A plain text file containing the license for␣
↪the chart
        README.md           # OPTIONAL: A human-readable README file
        values.yaml         # The default configuration values for this chart
        charts/             # A directory containing any charts upon which this chart␣
↪depends.
        templates/          # A directory of templates that, when combined with␣
↪values,
                            # will generate valid Kubernetes manifest files.
        templates/NOTES.txt # OPTIONAL: A plain text file containing short usage notes
        templates/tests     # A directory of test templates for running with 'helm␣
↪test'
```

All template files in the `templates/` directory should be named in a readily identifiable way after the component that it contains, and if further clarification is required then it should be suffixed with the `Kind` of resource eg: `tangodb.yaml` contains the `StatefulSet` for the Tango database, and `tangodb-pv.yaml` contains the `PersistentVolume` declaration for the Tango database. `ConfigMaps` should be clustered in `configmaps.yaml` and `Secrets` in `secrets.yaml`. The aim is to make it easy for others to understand the layout of application suite being deployed.

Helm sub-chart architecture

Introduction to subcharts

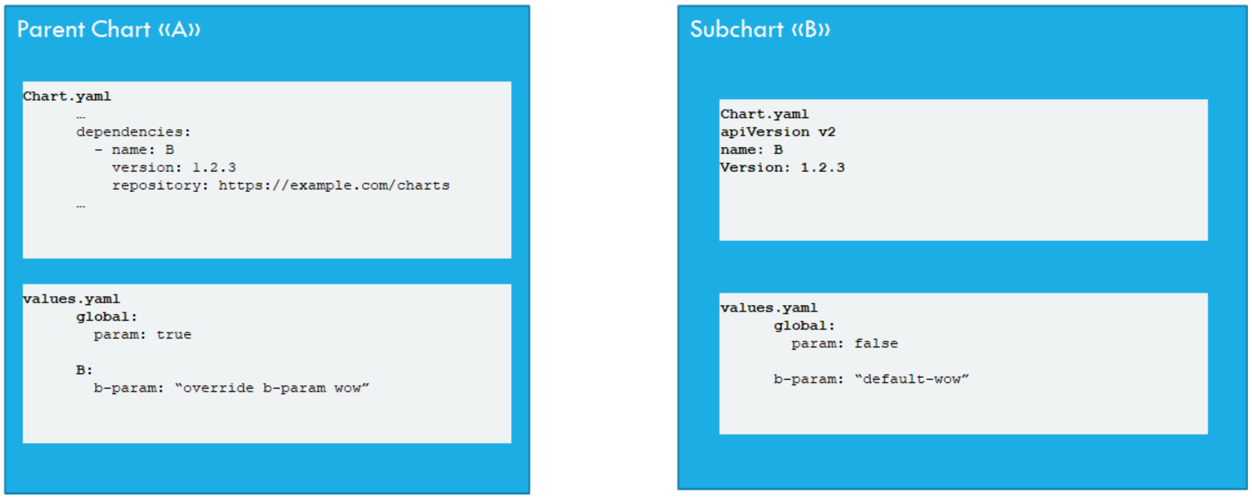
A chart can have one or more dependencies charts, called sub-charts. According to the helm documentation:

- a chart is stand-alone (cannot depend on a parent chart),
- a sub-chart cannot access the values of its parent,
- a parent sub-chart can override values for its sub-charts and
- all charts (parent and sub-chart) can access the global values.

Let's consider two charts, A and B where A depends on B. The file Chart.yaml for the chart A will specify the dependency and in the values file it is possible for chart A to override any value of chart B. The following figure shows how to do it:

It is also important to consider the operational aspects of using dependencies which state that when Helm installs/upgrades a chart, the Kubernetes objects from the chart and all its dependencies are

- aggregated into a single set; then
- sorted by type followed by name; and then



```
Chart.yaml
      …
      dependencies:
        - name: B
          version: 1.2.3
          repository: https://example.com/charts
      …
```

```
values.yaml
      global:
        param: true

      B:
        b-param: "override b-param wow"
```

```
Chart.yaml
apiVersion v2
name: B
Version: 1.2.3
```

```
values.yaml
      global:
        param: false

      b-param: "default-wow"
```

Fig. 12: Chart A parent of chart B

- created/updated in that order.

This means that if chart A defines the following k8s resources:

- namespace "A-Namespace"
- statefulset "A-StatefulSet"
- service "A-Service"
- and chart B defines the following k8s resources:
- namespace "B-Namespace"
- statefulset "B-ReplicaSet"
- service "B-Service"

Then the result of the helm install command for chart A will be:

- A-Namespace
- B-Namespace
- A-Service
- B-Service
- B-ReplicaSet
- A-StatefulSet.

Subcharts architecture

Considering the Module Views for the evolutionary prototype (section "Primary representation: MVP Uses in Kubernetes Deployment"), a partial dependency diagram for the helm charts available within the gitlab.com/ska-telescope group can be represented by the following diagram:

All charts depend on the tango-base and, in general, all charts could need the archiver and the webjive interface. At the moment, this is modelled in skampi repository where there is one parent chart called skampi and all other charts are its subcharts. They are installed with Helm templating instead of normal installation There are a number of disadvantages in this model specifically:

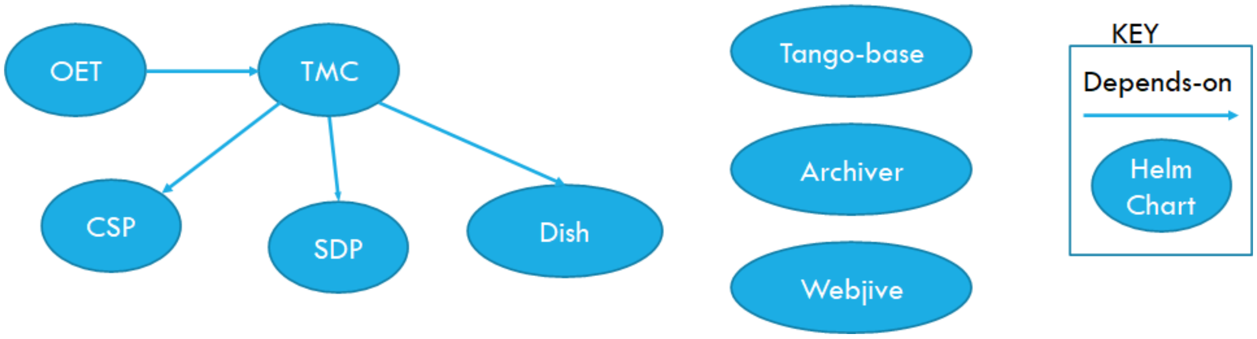


Fig. 13: Simple skampi diagram

- Common testing: one place for all integration testing. No clear distinction between system and integration tests
- Not easy to find logs: many tests on the same namespace
- Same namespace for many deployments
- No versioning: charts are not versioned

Three solutions have been proposed and described in the Supporting model page:

1. One parent chart (umbrella) that contains everything needed and Subcharts with no dependency
2. Charts with dependencies and Subcharts enabled by levels
3. Charts with dependencies and Subcharts enabled with conditions and tags

The chosen solution is an hybrid approach which enables a single level hierarchy for the shared charts and umbrella charts for charts composition (i.e. specific deployment or testing purpose). The rational is:

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

Every dependency must have a common condition on it, so that it will be possible to disable the shared charts if they are included in the parent umbrella. For instance if there is the need (for testing purposes) to have the TMC and the OET charts together the result will be:

The initial model will become:

Gitlab Helm/k8s testing pipeline

In order to enable the GitLab pipeline to deploy and test the specific component each ska-telescope repository must:

- contain at least one helm chart (i.e. starting point is skampi charts): link to example
- have an environment (i.e. test): link to example
- adopt the Makefile for k8s testing: link to example
- use the environment keywords: link to example
- have a common publish chart CI job step: link to example

Also, note that each project/repository in the ska-telescope group has a Kubernetes cluster already enabled.

The test job of the GitLab pipeline needs to be:

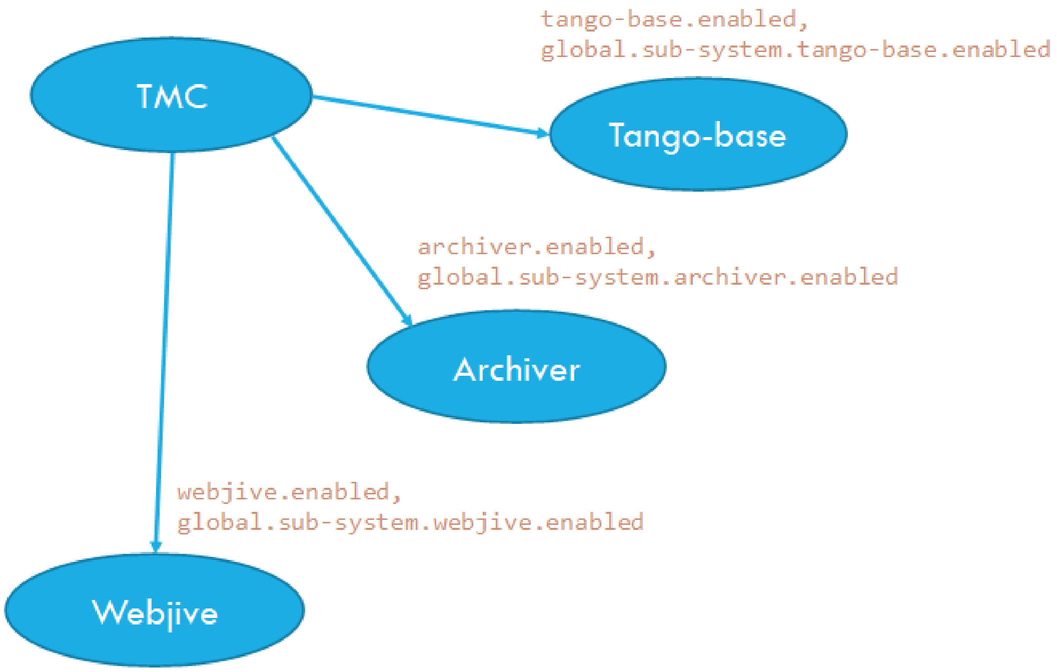
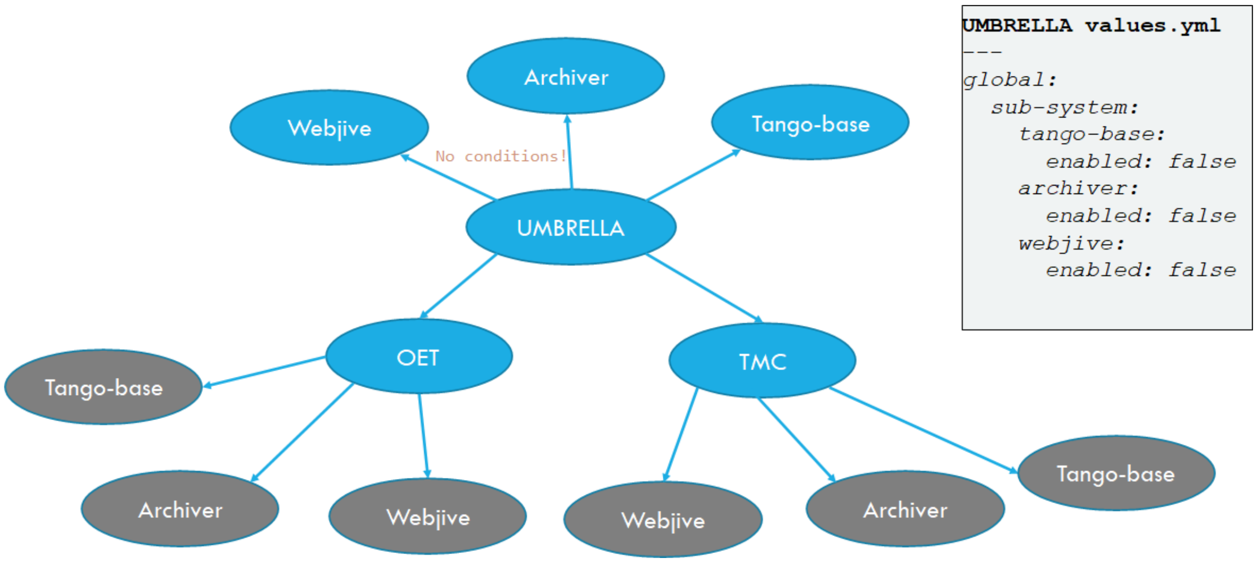
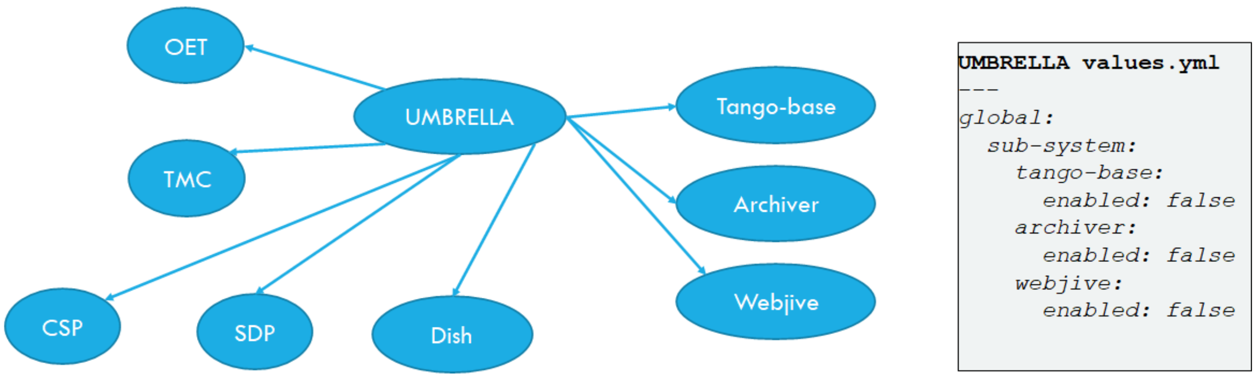

TMC
Tango-base
Archiver
Webjive
tango-base.enabled,
global.sub-system.tango-base.enabled
archiver.enabled,
global.sub-system.archiver.enabled
webjive.enabled,
global.sub-system.webjive.enabled

Fig. 14: Chart TMC with shared charts



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

Fig. 15: Umbrella chart with tmc and oet charts



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

Fig. 16: Umbrella chart for skampi: initial model refactored

```
test:
stage: test
tags:
  - docker-executor
image: nexus.engageska-portugal.pt/ska-docker/deploy:0.4.1
script:
  - kubectl version
  - make install-chart
  - make wait
  - make smoketest
  - make test
after_script:
  - make uninstall-chart
  - make delete_namespace
environment:
  name: test
  kubernetes:
    namespace: ci-$CI_PROJECT_NAME-$CI_COMMIT_SHORT_SHA
artifacts:
  name: "$CI_PROJECT_NAME-$CI_JOB_ID"
  paths:
    - "charts/build"
  reports:
    junit: charts/build/report.xml
```

where:

- make install: installs the chart in the namespace specified in the environment tag
- make wait: wait for all jobs to be completed and all pods to be running
- make smoketest: checks that no containers are waiting
- make test:
    1. Create a pod into the specified namespace
    2. Run pytests
    3. Return the tests results
- after_script: remove everything after tests

The artifacts are the output of the tests and it will have the report both in xml and json but also other information like the pytest output.

Tango-util library chart

A library chart is a type of Helm chart that defines chart primitives or definitions which can be shared by Helm templates in other charts. In SKAMPI, many charts are a collections of device servers so it is possible to harmonize their definition with a library so to keep charts DRY.

The following diagram shows the data model for the harmonized values file:

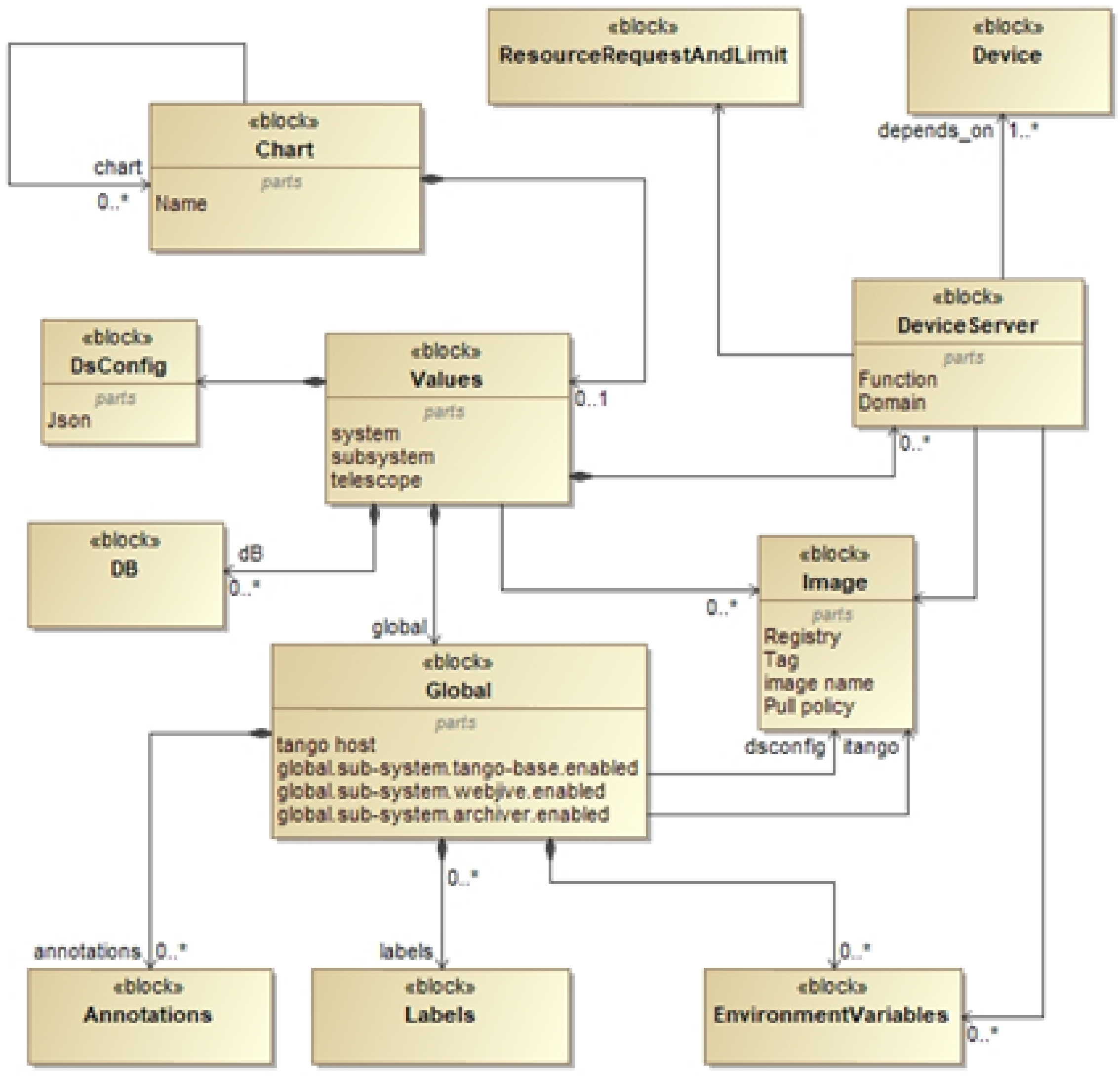


Fig. 17: Data model for the values file

Elements:

Element	Description
Chart	collection of files that describe a related set of Kubernetes resources
Values	built-in objects of helm which provides access to values passed into the chart for templating
DsConfig	dsconfig file configuration
DeviceServer	TANGO Device Server
Device	TANGO device
Global	Global values accessible by all charts
Labels	to be added to all Kubernetes resources
Environment variables	Name/Value pair available in shell
Image	Detail of the docker image to be used
ResourceRequestandLimits	struct for characterise the resource requests and limits for a device server
DB	struct for characterise a DB software application

Rationale:

- Almost all helm charts in the Skampi repository are device server configurations so it appears natural to start the modelling from that concept
- The depends_on relationship has been added so that it is possible to extract the dependency map of the MVP prototype
- Every chart of the ska-telescope can have the shared charts in the dependency list
- The annotations block has been added to enable GitLab's Deploy Boards
- The DeviceServer struct specifies the shell args so that it is possible to start the related container instance of the linked image

Advantages

With this architecture, a number of advantages can be obtained:

- By using a separate deployment (i.e. Namespace) for each test, searching for all the logs of a particular test will be easy: example
- Requires teams to create versions of docker images and charts
- Avoids the use of docker-compose in favour of Kubernetes testing
- Harmonized values yml files (for «common» definitions i.e. TANGO device servers)
- Unit and integration testing within the repositories of teams
- Skampi testing becomes system testing

Helm Best Practices

The Helm community have a well defined set of best practices. The following highlights key aspects of these practices that will help with achieving consistency and reliability.

- charts should be placed in a `charts/` directory within the parent project.
- chart names should be lowercase and hyphenated and must match the directory name eg. `charts/my-app`.
- `name`, `version`, `description`, `home`, `maintainers` and `sources` must be included.

- `version` must follow the Semantic Versioning standards.
- the chart must pass the `helm lint charts/<chart-name>` test.

> **Warning: Helm 2 vs Helm 3**
>
> It should be noted that we have now migrated to using Helm 3. Feel free to upgrade Helm in your development environments using our Ansible Playbook `upgrade_helm.yml` found in the SKA Ansible Playbooks repository.
>
> There are a few changes that may impact specific cases, to read up on them please read up at This blog post, as well as on Helm's own FAQ page.

Example `Chart.yaml` file:

```
name: my-app
version: 1.0.0
description: Very important app
keywords:
- magic
- mpi
home: https://www.skatelescope.org/
icon: http://www.skatelescope.org/wp-content/uploads/2016/07/09545_NEW_LOGO_2014.png
sources:
- https://gitlab.com/ska-telescope/my-app
maintainers:
- name: myaccount
  email: myacount@skatelescope.org
```

Metadata with Helm

All resources should have the following boilerplate metadata to ensure that all resources can be uniquely identified to the chart, application and release:

```
...
metadata:
name: <component>-{{ template "my-app.name" . }}-{{ .Release.Name }}
labels:
    app.kubernetes.io/name: <component>-{{ template "my-app.name" . }}-{{ .Release.
↪Name }}
    helm.sh/chart: "{{ .Chart.Name }}-{{ .Chart.Version }}"
    app.kubernetes.io/instance: "{{ .Release.Name }}"
    app.kubernetes.io/managed-by: "{{ .Release.Service }}"
...
```

Defining resources

The Helm templating language is based on Go template.

All resources go in the `templates/` directory with the general rule is one Kubernetes resource per template file. Files that render resources are suffixed `.yaml` whilst files that contain expressions and macros only go in files suffixed `.tpl`.

Sample resource template for a Service generated by 'helm create mychart'

```
apiVersion: v1
kind: Service
metadata:
name: {{ include "mychart.fullname" . }}
labels:
  app.kubernetes.io/name: {{ include "mychart.name" . }}
  helm.sh/chart: {{ include "mychart.chart" . }}
  app.kubernetes.io/instance: {{ .Release.Name }}
  app.kubernetes.io/managed-by: {{ .Release.Service }}
spec:
  type: {{ .Values.service.type }}
  ports:
  - port: {{ .Values.service.port }}
    targetPort: http
    protocol: TCP
    name: http
  selector:
    app.kubernetes.io/name: {{ include "mychart.name" . }}
    app.kubernetes.io/instance: {{ .Release.Name }}
```

Expression or macro template generated by 'helm create mychart'

```
{{/* vim: set filetype=mustache: */}}
{{/*
Expand the name of the chart.
*/}}
{{- define "mychart.name" -}}
{{- default .Chart.Name .Values.nameOverride | trunc 63 | trimSuffix "-" -}}
{{- end -}}

{{/*
Create a default fully qualified app name.
We truncate at 63 chars because some Kubernetes name fields are limited to this (by␣
↪the DNS naming spec).
If release name contains chart name it will be used as a full name.
*/}}
{{- define "mychart.fullname" -}}
{{- if .Values.fullnameOverride -}}
{{- .Values.fullnameOverride | trunc 63 | trimSuffix "-" -}}
{{- else -}}
{{- $name := default .Chart.Name .Values.nameOverride -}}
{{- if contains $name .Release.Name -}}
{{- .Release.Name | trunc 63 | trimSuffix "-" -}}
{{- else -}}
{{- printf "%s-%s" .Release.Name $name | trunc 63 | trimSuffix "-" -}}
{{- end -}}
{{- end -}}
{{- end -}}

{{/*
Create chart name and version as used by the chart label.
*/}}
{{- define "mychart.chart" -}}
{{- printf "%s-%s" .Chart.Name .Chart.Version | replace "+" "_" | trunc 63 |␣
↪trimSuffix "-" -}}
{{- end -}}
```

Tightly coupled resources may go in the same template file where they are logically linked or there is a form of dependency.

An example of logically linked resources are PersistentVolume and PersistentVolumeClaim definitions. Keeping these together makes debugging and maintenance easier.

PersistentVolume and PersistentVolumeClaim definitions

```
---
kind: PersistentVolume
apiVersion: v1
metadata:
    name: tangodb-{{ template "tango-chart-example.name" . }}-{{ .Release.Name }}
    namespace: {{ .Release.Namespace }}
labels:
    app.kubernetes.io/name: tangodb-{{ template "tango-chart-example.name" . }}
    app.kubernetes.io/instance: "{{ .Release.Name }}"
    app.kubernetes.io/managed-by: "{{ .Release.Service }}"
    helm.sh/chart: "{{ template "tango-chart-example.chart" . }}"
spec:
    storageClassName: standard
    capacity:
        storage: 1Gi
    accessModes:
        - ReadWriteOnce
    hostPath:
        path: /data/tangodb-{{ template "tango-chart-example.name" . }}/

---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: tangodb-{{ template "tango-chart-example.name" . }}-{{ .Release.Name }}
    namespace: {{ .Release.Namespace }}
labels:
    app.kubernetes.io/name: tangodb-{{ template "tango-chart-example.name" . }}
    app.kubernetes.io/instance: "{{ .Release.Name }}"
    app.kubernetes.io/managed-by: "{{ .Release.Service }}"
    helm.sh/chart: "{{ template "tango-chart-example.chart" . }}"
spec:
    storageClassName: standard
    accessModes:
        - ReadWriteOnce
    resources:
        requests:
            storage: 1Gi
    volumeName: tangodb-{{ template "tango-chart-example.name" . }}-{{ .Release.Name }
↪}
```

An example of dependency is the declaration of a Service before the associated Pod/Deployment/StatefulSet/DaemonSet. The Pod will get the environment variables set from the Service as this will be evaluated by the Kubernetes API first as guaranteed by being in the same template file.

Service before the associated Pod/Deployment

```
---
apiVersion: v1
kind: Service
metadata:
name: tango-rest-{{ template "tango-chart-example.name" . }}-{{ .Release.Name }}
namespace: {{ .Release.Namespace }}
labels:
  app.kubernetes.io/name: tango-rest-{{ template "tango-chart-example.name" . }}
  app.kubernetes.io/instance: "{{ .Release.Name }}"
  app.kubernetes.io/managed-by: "{{ .Release.Service }}"
  helm.sh/chart: "{{ template "tango-chart-example.chart" . }}"
spec:
  type: ClusterIP
  ports:
  - name: rest
    port: 80
    targetPort: rest
    protocol: TCP
  selector:
    app.kubernetes.io/name: tango-rest-{{ template "tango-chart-example.name" . }}
    app.kubernetes.io/instance: "{{ .Release.Name }}"

---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: tango-rest-{{ template "tango-chart-example.name" . }}-{{ .Release.Name }}
  namespace: {{ .Release.Namespace }}
labels:
  app.kubernetes.io/name: tango-rest-{{ template "tango-chart-example.name" . }}
  app.kubernetes.io/instance: "{{ .Release.Name }}"
  app.kubernetes.io/managed-by: "{{ .Release.Service }}"
  helm.sh/chart: "{{ template "tango-chart-example.chart" . }}"
spec:
  replicas: {{ .Values.tangorest.replicas }}
  template:
    metadata:
      labels:
        app.kubernetes.io/name: tango-rest-{{ template "tango-chart-example.name" . }}
        app.kubernetes.io/instance: "{{ .Release.Name }}"
        app.kubernetes.io/managed-by: "{{ .Release.Service }}"
        helm.sh/chart: "{{ template "tango-chart-example.chart" . }}"
    spec:
      containers:
      - name: tango-rest
        image: "{{ .Values.tangorest.image.registry }}/{{ .Values.tangorest.image.
↪image }}:{{ .Values.tangorest.image.tag }}"
        imagePullPolicy: {{ .Values.tangorest.image.pullPolicy }}
        command:
        - /usr/local/bin/wait-for-it.sh
        - databaseds-{{ template "tango-chart-example.name" . }}-{{ .Release.Name }}
↪:10000
        - --timeout=30
        - --strict
        - --
        - /usr/bin/supervisord
        - --configuration
```

(continues on next page)

(continued from previous page)

```
        - /etc/supervisor/supervisord.conf
        env:
          - name: TANGO_HOST
            value: databaseds-{{ template "tango-chart-example.name" . }}-{{ .Release.
↪Name }}:10000
        ports:
          - name: rest
            containerPort: 8080
            protocol: TCP
    restartPolicy: Always
{{- with .Values.nodeSelector }}
    nodeSelector:
{{ toYaml . | indent 8 }}
{{- end }}
{{- with .Values.affinity }}
    affinity:
{{ toYaml . | indent 8 }}
{{- end }}
{{- with .Values.tolerations }}
    tolerations:
{{ toYaml . | indent 8 }}
{{- end }}
```

Note: It may also be necessary to consider the alphabetic ordering of template files, if there is a declaration dependency wider than the immediate file, for instance when s `Service` definition and it's environment variables are necessary for multiple Deployment/StatefulSet/DaemonSet definitions. In this case, it maybe necessary to use a numerical file prefix such as 00-service-and-pod.yaml, 01-db-statefulset.yaml …

Use comments liberally in the template files to describe the intended purpose of the resource declarations and any other features of the template markup. # YAML comments get copied through to the rendered template output and are a valuable help when debugging template issues with `helm template charts/chart-name/ ....`

Managing configuration

Helm charts and the Go templating engine enable separation of application management concerns along multiple lines:

- resources are broken out into related and named templates.
- Application specific configuration values are placed in `ConfigMaps`.
- volatile run time configuration values are placed in the `values.yaml` file, and then templated into `ConfigMaps`, container commandline parameters or environment variables as required.
- sensitive configuration is placed in `Secrets`.
- template content is programable (iterators and operators) and this can be parameterised at template rendering time.

Variable names for template substitution should observe the following rules:

- Use camel-case or lowercase variable names - never hyphenated.
- Structure parameter values in shallow nested structures to make it easier to pass on the Helm command line eg: `--set tangodb.db.connection.host=localhost` is convoluted compared to `--set tangodb.host=localhost`.
- Use explicitly typed values eg: `enabled:  false` is not `enabled:  "false"`.

- Be careful of how YAML parsers coerce value types - long integers get coerced into scientific notation so if in doubt use strings and type casting eg: `foo: "12345678"` and `{{ .Values.foo | int }}`.
- use comments in the `values.yaml` liberally to describe the intended purpose of variables.

Config in ConfigMaps

`ConfigMaps` can be used to populate `Pod` configuration files, environment variables and command line parameters where the values are largely stable, and should not be bundled with the container itself. This should include any (small) data artefacts that could be different (hence configured) between different instances of the running containers. Even files that already exist inside a given container image can be overwritten by using the `volumeMounts` example below.

ConfigMap values in Pods

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  SPECIAL_LEVEL: very
  SPECIAL_TYPE: charming
  example.ini: |-
    property.1=value-1
    property.2=value-2
    property.3=value-3
---
apiVersion: v1
kind: Pod
metadata:
 name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      # accessing ConfigMap values in the commandline fron env vars
      command: [ "/bin/sh", "-c", "echo $(SPECIAL_LEVEL_KEY) $(SPECIAL_TYPE_KEY); cat␣
↪/etc/config/example.ini" ]
      env:
        # reference the map and key to assign to env var
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: SPECIAL_LEVEL
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: SPECIAL_TYPE
      volumeMounts:
      # mount a ConfigMap file blob as a configuration file
      - name: config-volume
        mountPath: /etc/config/example.ini
        subPath: example.ini
```

(continues on next page)

(continued from previous page)

```
        readOnly: true
  volumes:
    - name: config-volume
      configMap:
        # Provide the name of the ConfigMap containing the files you want
        # to add to the container
        name: special-config
  restartPolicy: Never
# check the logs with kubectl logs dapi-test-pod
# clean up with kubectl delete pod/dapi-test-pod configmap/special-config
```

Where configuration objects are large or have a sensitive format, then separate these out from the `configmaps.yaml` file, and then include them using the template directive: `tpl (.Files.Glob "configs/*").AsConfig . )` where the `configs/` directory is relative to the `charts/my-chart` directory.

ConfigMap file blobs separated

```
---
apiVersion: v1
kind: ConfigMap
metadata:
name: config-{{ template "tango-chart-example.name" . }}-{{ .Release.Name }}
labels:
    app.kubernetes.io/name: config-{{ template "tango-chart-example.name" . }}-{{ .
↪Release.Name }}
    app.kubernetes.io/instance: "{{ .Release.Name }}"
    app.kubernetes.io/managed-by: "{{ .Release.Service }}"
    helm.sh/chart: "{{ template "tango-chart-example.chart" . }}"
data:
{{ (tpl (.Files.Glob "configs/*").AsConfig . ) | indent 2  }}
```

Secrets

`Secrets` information is treated in almost exactly the same way as `ConfigMaps`. While the default configuration (as at v1.14.x) is for `Secrets` to be stored as Base64 encoded in the etcd database, it is possible and expected that the Kubernetes cluster will be configured with encryption at rest (available from v1.13). All account details, passwords, tokens, keys and certificates should be extracted and managed using `Secrets`.

As was for `ConfigMaps`, separate `Secrets` out into the `secrets.yaml` template.

Secret values in Pods

```
---
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
  username: myuser
  password: mypassword
  config.yaml: |-
    apiUrl: "https://my.api.com/api/v1"
    username: myuser
    password: mypassword
```

(continues on next page)

(continued from previous page)

```
---
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
  - name: mycontainer
    image: k8s.gcr.io/busybox
    # accessing Secret values in the commandline fron env vars
    command: [ "/bin/sh", "-c", "echo $(SECRET_USERNAME) $(SECRET_PASSWORD); cat /etc/
↪config/example.yaml" ]
    env:
    - name: SECRET_USERNAME
      valueFrom:
        secretKeyRef:
          name: mysecret
          key: username
    - name: SECRET_PASSWORD
      valueFrom:
        secretKeyRef:
          name: mysecret
          key: password
    volumeMounts:
    - name: foo
      mountPath: "/etc/config"
  volumes:
  - name: foo
    secret:
      secretName: mysecret
      items:
      - key: config.yaml
        path: example.yaml
        mode: 511
  restartPolicy: Never
# check the logs with kubectl logs secret-env-pod
# clean up with kubectl delete pod/secret-env-pod secret/mysecret
```

Where sensitive data objects are large or have a sensitive format, then separate these out from the `secrets.yaml` file, and then include them using the template directive: `tpl (.Files.Glob "secrets/*").AsSecrets . )` where the `secrets/` directory is relative to the `charts/my-chart` directory.

Secret file blobs separated

```
---
apiVersion: v1
kind: Secret
metadata:
name: secret-{{ template "tango-chart-example.name" . }}-{{ .Release.Name }}
labels:
    app.kubernetes.io/name: secret-{{ template "tango-chart-example.name" . }}-{{ .
↪Release.Name }}
    app.kubernetes.io/instance: "{{ .Release.Name }}"
    app.kubernetes.io/managed-by: "{{ .Release.Service }}"
    helm.sh/chart: "{{ template "tango-chart-example.chart" . }}"
type: Opaque
```

(continues on next page)

(continued from previous page)

```
data:
{{ (tpl (.Files.Glob "secrets/*").AsSecrets . ) | indent 2  }}
```

Storage

`PersistentVolumes` and partner `PersistentVolumeClaims` should be defined by default in a separate template. This template should be bracketed with a switch to enable the storage declaration to be *turned off* (eg: `{{ if .Values.tangodb.createpv }}`), which will most likely be dependent on, and optimised for each environment.

On the `PersistentVolume`:

- All storage should be treated as ephemeral by setting `persistentVolumeReclaimPolicy:  Delete`.
- Explicitly set volume mode eg: `volumeMode:  Filesystem` so that it is clear whether `Filesystem` or `Block` is being requested.
- Explicitly set the access mode eg: `ReadWriteOnce, ReadOnlyMany, or ReadWriteMany` so that it is clear what access rights containers are expected to have.
- always specify the storage class - this should always default to `standard` eg: `storageClassName: standard` given that the default target environment is Minikube.

On the `PersistentVolumeClaim`:

- Always specify the matching storage class eg: `storageClassName:  standard`, so that it will bind to the intended `PersistentVolume` storage class.
- Where possible, always specify an explicit `PersistentVolume` with `volumeName` eg: `volumeName: tangodb-tango-chart-example-test`. This will force the `PersistentVolumeClaim` to bind to a specific `PersistentVolume` and storage class, avoiding the loosely binding issues that volumes can have.

Storage In Kubernetes Clusters Managed by the Systems Team

In any of the existing deployed Kubernetes clusters there are a number of default StorageClasses available, that are backed by Ceph, and integrated using Rook. The `StorageClass` es expose `RDB` block devices and `CephFS` Network File System based storage to Kubernetes.

The StorageClasses are as follows:

Classname	Maps to	Usage
nfss1	CephFS	Shared Network Filesystem - ReadWriteMany
nfs	alias to nfss1	Shared Network Filesystem - ReadWriteMany
bds1	RBD	Single concurrent use ext4 - ReadWriteOnce
block	alias to bds1	Single concurrent use ext4 - ReadWriteOnce

StorageClass naming convention follows the following pattern:

`<xxx type><x class><n version>[-<location>]`

- xxx type - bd=block device, nfs=network filesystem
- x class - s=standard,i=iops optimised (could be ssd/nvme), t=throughput optimised (could be hdd, or cheaper ssd)
- n version - 1=first version,. . .

- location - future tag for denoting location context, rack, dc, etc

Current classes:

- bds1 - block device - single mount (ReadWriteOnce) - standard - version 1
- nfss1 - network filesystem enabled storage (ReadWriteMany) - standard - version 1
- block = shortcut for bds1
- nfs = shortcut for nfss1

Tests

Helm Chart tests live in the `templates/tests` directory, and are essentially one `Pod` per file that must be run-to-completion (i.e. `restartPolicy: Never`). These `Pods` are annotated in one of two ways:

- `"helm.sh/hook": test-success` - `Pod` is expected to exit with return code `0`
- `"helm.sh/hook": test-failure` - `Pod` is expected to exit with return code `not equal 0`

This is a simple solution for test assertions at the `Pod` scale.

As with any other resource definition, tests should have name and metadata correctly scoping them. End the `Pod` name with a string that indicates what the test is suffixed with `-test`.

Helm tests, must be self contained are should be atomic and non-destructive as the intention is that a chart user can use the tests to determine that the chart installed correctly. As with the following example, the test is for checking that `Pods` can reach the DatabaseDS service. Other tests might be checking services are correctly exposed via `Ingress`.

Helm Chart test Pod - metadata and annotations on a simple connection test

```
---
apiVersion: v1
kind: Pod
metadata:
  name: databaseds-{{ template "tango-chart-example.name" . }}-{{ .Release.Name }}-
↪connection-test
  namespace: {{ .Release.Namespace }}
  labels:
    app.kubernetes.io/name: databaseds-{{ template "tango-chart-example.name" . }}
    app.kubernetes.io/instance: "{{ .Release.Name }}"
    app.kubernetes.io/managed-by: "{{ .Release.Service }}"
    helm.sh/chart: "{{ template "tango-chart-example.chart" . }}"
  annotations:
    "helm.sh/hook": test-success
spec:
  {{- if .Values.pullSecrets }}
  imagePullSecrets:
  {{- range .Values.pullSecrets }}
    - name: {{ . }}
  {{- end}}
  {{- end }}
  containers:
  - name: databaseds-{{ template "tango-chart-example.name" . }}-{{ .Release.Name }}-
↪connection-test
    image: "{{ .Values.powersupply.image.registry }}/{{ .Values.powersupply.image.
↪image }}:{{ .Values.powersupply.image.tag }}"
    imagePullPolicy: {{ .Values.powersupply.image.pullPolicy }}
    command:
```

(continues on next page)

(continued from previous page)

```
      - sh
    args:
      - -c
      - "( retry --max=10 -- tango_admin --ping-device test/power_supply/1 ) && echo
↪'test OK'"
    env:
    - name: TANGO_HOST
      value: databaseds-{{ template "tango-chart-example.name" . }}-{{ .Release.Name }
↪}:10000
  restartPolicy: Never
```

Integrating a chart into the SKAMPI repo

Prerequisites

- Verify that Docker, kubectl, Minikube and Helm are installed and working properly - refer to *Incorporate my project into the integration environment*.
- The required docker images have been uploaded to and are available from Nexus, see docker upload instructions

To integrate a helm chart into the *SKAMPI* repo, follow these steps:

Local steps

- Clone the *SKAMPI* repo, available here.
- Add a directory in *charts* with a descriptive name
- Add your helm chart and associated files within that directory
- Check the validity of the chart
  - Verify that the chart is formatted correctly

    ```
    helm lint ./charts/<your_chart_directory>/
    ```

  - Verify that the templates are rendered correctly and the output is as expected

    ```
    helm install --dry-run --debug ./charts/<your_chart_directory>/
    ```

    * For some debugging tips refer to: debugging tips.
  - Check that your chart deploys locally (utilising minikube as per our standards) and behaves as expected

    ```
    make deploy KUBE_NAMESPACE=integration
    make deploy KUBE_NAMESPACE=integration HELM_CHART=<your_chart_directory>
    ```

- Once functionality has been confirmed, go ahead and commit and push the changes

Gitlab

Once the changes had been pushed it will be built in Gitlab. Find the pipeline builds at https://gitlab.com/ska-telescope/skampi/pipelines.

If the pipeline completes successfully, the full integration environment will be available at https://integration.engageska-portugal.pt.

Kubernetes primitives

The following focuses on the core Kubernetes primitives - Pod, Service, and Ingress. These provide the core delivery chain of a networked application to the end consumer.

The Pod

The `Pod` is the basic deployable application unit in Kubernetes, and provides the primary configurable context of an application component. Within this construct, all configuration and resources are plugged in to the application.

This is a complete example that demonstrates container patterns, initContainers and life-cycle hooks discussed in the following sections.

Container patterns and life-cycle hooks

```
---
kind: Service
apiVersion: v1
metadata:
  name: pod-examples
spec:
  type: ClusterIP
  selector:
    app: pod-examples
  ports:
  - name: http
    protocol: TCP
    port: 80
    targetPort: http

---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: pod-examples
  labels:
    app: pod-examples
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: pod-examples
    spec:
      volumes:
      # lifecyle containers as hooks share state using volumes
      - name: shared-data
        emptyDir: {}
      - name: the-end
        hostPath:
          path: /tmp
          type: Directory
```

(continues on next page)

(continued from previous page)

```
    initContainers:
    # initContainers can initialise data, and do pre-flight checks
    - name: init-container
      image: alpine
      command: ['sh', '-c', "echo 'initContainer says: hello!' > /pod-data/status.
↪txt"]
      volumeMounts:
      - name: shared-data
        mountPath: /pod-data

    containers:
    # primary data generator container
    - name: main-app-container
      image: alpine
      command: ["/bin/sh"]
      args: ["-c", "while true; do echo 'Main app says: ' `date` >> /pod-data/
↪status.txt; sleep 5;done"]
      lifecycle:
        # postStart hook is async task called on Pod boot
        # useful for async container warmup tasks that are not hard dependencies
        # definitely not guaranteed to run before main container command
        postStart:
          exec:
            command: ["/bin/sh", "-c", "echo 'Hello from the postStart handler' >> /
↪pod-data/status.txt"]
        # preStop hook is async task called on Pod termination
        # useful for initiating termination cleanup tasks
        # definitely not guaranteed to complete before container termination (sig␣
↪KILL)
        preStop:
          exec:
            command: ["/bin/sh", "-c", "echo 'Hello from the preStop handler' >> /
↪the-end/last.txt"]
      volumeMounts:
      - name: shared-data
        mountPath: /pod-data
      - name: the-end
        mountPath: /the-end

    # Sidecar helper that exposes data over http
    - name: sidecar-nginx-container
      image: nginx
      ports:
        - name: http
          containerPort: 80
          protocol: TCP
      volumeMounts:
      - name: shared-data
        mountPath: /usr/share/nginx/html
      livenessProbe:
        httpGet:
          path: /index.html
          port: http
      readinessProbe:
        httpGet:
          path: /index.html
          port: http
```

(continues on next page)

(continued from previous page)

```
      # Ambassador pattern used as a proxy or shim to access external inputs
      # gets date from Google and adds it to input
      - name: ambassador-container
        image: alpine
        command: ["/bin/sh"]
        args: ["-c", "while true; do echo 'Ambassador says: '`wget -S -q 'https://
↪google.com/' 2>&1 | grep -i '^  Date:' | head -1 | sed 's/^  [Dd]ate: //g'` > /pod-
↪data/input.txt; sleep 60; done"]
        volumeMounts:
        - name: shared-data
          mountPath: /pod-data

      # Adapter pattern used as a proxy or shim to generate/render outputs
      # fit for external consumption (similar to Sidecar)
      # reformats input data from sidecar and ambassador ready for output
      - name: adapter-container
        image: alpine
        command: ["/bin/sh"]
        args: ["-c", "while true; do cat /pod-data/status.txt | head -3 > /pod-data/
↪index.html; cat /pod-data/input.txt | head -1 >> /pod-data/index.html; cat /pod-
↪data/status.txt | tail -1 >> /pod-data/index.html;  echo 'All from your friendly␣
↪Adapter' >> /pod-data/index.html; sleep 5; done"]
        volumeMounts:
        - name: shared-data
          mountPath: /pod-data
```

This will produce output that demonstrates each of the containers fulfilling their role:

```
$ curl http://`kubectl get service/pod-examples -o jsonpath="{.spec.
↪clusterIP}"`
initContainer says: hello!
Main app says:  Thu May 2 03:45:42 UTC 2019
Hello from the postStart handler
Ambassador says: Thu, 02 May 2019 03:45:55 GMT
Main app says:  Thu May 2 03:46:12 UTC 2019
All from your friendly Adapter

$ kubectl delete deployment/pod-examples service/pod-examples
deployment.extensions "pod-examples" deleted
service "pod-examples" deleted
piers@wattle:~$ cat /tmp/last.txt
Hello from the preStop handler
```

Container patterns

The `Pod` is a cluster of one or more containers that share the same resource namespaces. This enables the Pod cluster to communicate as though they are on the same host which is ideal for preserving the one-process-per-container ideal, but be able to deliver orchestrated processes as a single application that can be separately maintained.

All `Pod` deployments should be designed around having a core or leading container. All other containers in the `Pod` provide auxillary or secondary services. There are three main patterns for multi-container `Pods`:

- Sidecar - extend the primary container functionality eg: adds logging, metrics, health checks (as input to livenessProbe/readinessProbe).

- Ambasador - container that acts as an out-bound proxy for the primary container by handling translations to external services.
- Adapter - container that acts as an in-bound proxy for the primary container aligning interfaces with alternative standards.

initContainers

Any serial container action that does not neatly fit into the one-process-per-container pattern, should be placed in an `initContainer`. These are typically actions like initialising databases, checking for upgrade processes, executing migrations. `initContainer` are executed in order, and if any one of them fails, the `Pod` will be restarted inline with the `restartPolicy`. With this behaviour, it is important to ensure that the `initContainer` actions are idempotent, or there will be harmful side effects on restarts.

postStart/preStop

Life-cycle hooks have very few effective usecases as there is no guarantee that a `postStart` task will run before the main container command does (this is demonstrated above), and there is no guarantee that a `preStop` task (which is only issued when a Pod is terminated - not completed) will complete before the `KILL` signal is issued to the parent container after the cluster wide configured grace period (30s).

The value of the lifecycle hooks are generally reserved for:

- `postStart` - running an asynchronous non-critical task in the parent container that would otherwise slow down the boot time for the `Pod` and impact service availability.
- `preStop` - initiating asynchronous clean up tasks via an external service - essentially an opportunity to send a quick message out before the `Pod` is fully terminated.

readinessProbe/livenessProbe

Readiness probes are used by the scheduler to determine whether the container is in a state ready to serve requests. Liveness probes are used by the scheduler to determine whether the container continues to be in a healthy state for serving requests. Where possible, `livenessProbe` and `readinessProbe` should be specified. This is automatically used to calculate whether a `Pod` is available and healthy and whether it should be added and load balanced in a `Service`. These features can play an important role in the continuity of service when clusters are auto-healed, workloads are shifted from node to node, or during rolling updates to deployments.

The following shows the registered probes and their status for the *sidecar container in the examples above*:

```
$ kubectl describe deployment.apps/pod-examples
...
sidecar-nginx-container:
    Image:         nginx
    Port:          80/TCP
    Host Port:     0/TCP
    Liveness:      http-get http://:http/index.html delay=0s timeout=1s␣
↪period=10s #success=1 #failure=3
    Readiness:     http-get http://:http/index.html delay=0s timeout=1s␣
↪period=10s #success=1 #failure=3
    Environment:  <none>
    Mounts:
    /usr/share/nginx/html from shared-data (rw)
...
```

While probes can be a command, it is better to make health checks an http service that is combined with an application metrics handler so that external applications can use the same feature to do health checking (eg: Prometheus, or Icinga).

Sharing, Networking, Devices, Host Resource Access

Sharing resources is often the bottle neck in High Performance Computing, and where the greatest attention to detail is required with containerised applications in order to gain acceptable performance and efficency.

Containers within a `Pod` can share resources with each other directly using shared volumes, network, and memory. These are the preferred methods because they are cross-platform portable for containers in general, Kubernetes and OS/hardware.

The following example demonstrates how to share memory as a volume between containers:

Pod containers sharing memory

```yaml
---
kind: Service
apiVersion: v1
metadata:
  name: pod-sharing-memory-examples
  labels:
    app: pod-sharing-memory-examples
spec:
  type: ClusterIP
  selector:
    app: pod-sharing-memory-examples
  ports:
  - name: ncat
    protocol: TCP
    port: 5678
    targetPort: ncat

---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: pod-sharing-memory-examples
  labels:
    app: pod-sharing-memory-examples
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: pod-sharing-memory-examples
    spec:
      containers:
      # Producer - write to shared memory
      - name: producer-container
        image: python:3.7
        command: ["/bin/sh"]
        args: ["-c", "python3 /src/mmapexample.py -p; sleep infinity"]
        volumeMounts:
        - name: src
          mountPath: /src/mmapexample.py
```

(continues on next page)

(continued from previous page)

```
          subPath: mmapexample.py
          readOnly: true
        - mountPath: /dev/shm
          name: dshm

      # Consumer - read from shared memory and publish on 5678
      - name: consumer-container
        image: python:3.7
        command: ["/bin/sh"]
        # mutating container - this is bad practice but we need netcat for this␣
↪example
        args: ["-c", "apt-get update; apt-get -y install netcat-openbsd; python3 -u /
↪src/mmapexample.py | nc -l -k -p 5678; sleep infinity"]
        ports:
        - name: ncat
          containerPort: 5678
          protocol: TCP
        volumeMounts:
        - name: src
          mountPath: /src/mmapexample.py
          subPath: mmapexample.py
          readOnly: true
        - mountPath: /dev/shm
          name: dshm

      volumes:
        - name: src
          configMap:
            name: pod-sharing-memory-examples
        - name: dshm
          emptyDir:
            medium: Memory

    # test with:
    # $ nc `kubectl get service/pod-sharing-memory-examples -o jsonpath="{.spec.
↪clusterIP}"` 5678
    # Producers says: 2019-05-05 19:21:10
    # Producers says: 2019-05-05 19:21:11
    # Producers says: 2019-05-05 19:21:12
    # $ kubectl delete deployment,svc,configmap -l app=pod-sharing-memory-examples
    # deployment.extensions "pod-sharing-memory-examples" deleted
    # service "pod-sharing-memory-examples" deleted
    # configmap "pod-sharing-memory-examples" deleted
    # debug with: kubectl logs -l app=pod-sharing-memory-examples -c producer-
↪container

---
apiVersion: v1
kind: ConfigMap
metadata:
  name: pod-sharing-memory-examples
  labels:
    app: pod-sharing-memory-examples
data:
  mmapexample.py: |-
    #!/usr/bin/env python3
    # -*- coding: utf-8 -*-
```

(continues on next page)

(continued from previous page)

```
""" example mmap python client
"""

import datetime
import time
import getopt
import os
import os.path
import sys
import logging
from collections import namedtuple
import mmap
import signal


def parse_opts():
    """ Parse out the command line options
    """
    options = {
        'mqueue': "/example_shared_memory_queue",
        'debug': False,
        'producer': False
    }

    try:
        (opts, _) = getopt.getopt(sys.argv[1:],
                                  'dpm:',
                                  ["debug",
                                  "producer"
                                  "mqueue="])
    except getopt.GetoptError:
        print('mmapexample.py [-d -p -m <message_queue_name>]')
        sys.exit(2)

    dopts = {}
    for (key, value) in opts:
        dopts[key] = value
    if '-p' in dopts:
        options['producer'] = True
    if '-m' in dopts:
        options['mqueue'] = dopts['-m']
    if '-d' in dopts:
        options['debug'] = True

    # container class for options parameters
    option = namedtuple('option', options.keys())
    return option(**options)


# main
def main():
    """ Main
    """
    options = parse_opts()

    # setup logging
    logging.basicConfig(level=(logging.DEBUG if options.debug
```

(continues on next page)

(continued from previous page)

```
                                else logging.INFO),
                            format=('%(asctime)s [%(name)s] ' +
                                    '%(levelname)s: %(message)s'))
    logging.info('mqueue: %s mode: %s', options.mqueue,
                 ('Producer' if options.producer else 'Consumer'))

    # trap the keyboard interrupt
    def signal_handler(signal_caught, frame):
        """ Catch the keyboard interrupt and gracefully exit
        """
        logging.info('You pressed Ctrl+C!: %s/%s', signal_caught, frame)
        sys.exit(0)

    signal.signal(signal.SIGINT, signal_handler)

    mqueue_fd = os.open("/dev/shm/" + options.mqueue,
                        os.O_RDWR | os.O_SYNC | os.O_CREAT)

    last = ""
    while True:
        try:
            if options.producer:
                now = datetime.datetime.now()
                data = "Producers says: %s\n" % \
                    (now.strftime("%Y-%m-%d %H:%M:%S"))
                logging.debug('sending out to mqueue: %s', data)
                os.ftruncate(mqueue_fd, 512)
                with mmap.mmap(mqueue_fd, 0) as mqueue:
                    mqueue.seek(0)
                    mqueue[0:len(data)] = data.encode('utf-8')
                    mqueue.flush()
            else:
                with mmap.mmap(mqueue_fd, 0,
                               access=mmap.ACCESS_READ) as mqueue:
                    mqueue.seek(0)
                    data = mqueue.readline().rstrip().decode('utf-8')
                    logging.debug('from mqueue: %s', data)
                    if data == last:
                        logging.debug('same as last time - skipping')
                    else:
                        last = data
                        sys.stdout.write(data+"\n")
                        sys.stdout.flush()
        except Exception as ex:                 # pylint: disable=broad-except
            logging.debug('error: %s', repr(ex))

        time.sleep(1)

    logging.info('Finished')
    sys.exit(0)


# main
if __name__ == "__main__":

    main()
```

The following example demonstrates how to share memory over POSIX IPC between containers:

Pod containers sharing memory over POSIX IPC

```
---
kind: Service
apiVersion: v1
metadata:
  name: pod-ipc-sharing-examples
  labels:
    app: pod-ipc-sharing-examples
spec:
  type: ClusterIP
  selector:
    app: pod-ipc-sharing-examples
  ports:
  - name: ncat
    protocol: TCP
    port: 1234
    targetPort: ncat

---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: pod-ipc-sharing-examples
  labels:
    app: pod-ipc-sharing-examples
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: pod-ipc-sharing-examples
    spec:
      volumes:
      - name: shared-data
        emptyDir: {}

      initContainers:
      # get and build the ipc shmem tool
      - name: builder-container
        image: golang:1.11
        command: ['sh', '-c', "export GOPATH=/src; go get gitlab.com/ghetzel/shmtool"]
        volumeMounts:
        - name: shared-data
          mountPath: /src

      containers:
      # Producer
      - name: producer-container
        image: alpine
        command: ["/bin/sh"]

        args:
        - "-c"
        - >
          apk add -U util-linux;
```

(continues on next page)

(continued from previous page)

```
          mkdir /lib64 && ln -s /lib/libc.musl-x86_64.so.1 /lib64/ld-linux-x86-64.so.
↪2;
          ipcmk --shmem 1KiB;
          echo "ipcmk again as chmtool cant handle 0 SHMID";
          ipcmk --shmem 1KiB; > /pod-data/memaddr.txt;
          while true;
           do echo 'Main app (pod-ipc-sharing-examples) says: ' `date` | /pod-data/
↪bin/shmtool open -s 1024 `ipcs -m | cut -d' ' -f 2 | sed  '/^$/d' | tail -1`;
              sleep 1;
           done
        volumeMounts:
        - name: shared-data
          mountPath: /pod-data

      # Consumer - read from the pipe and publish on 1234
      - name: consumer-container
        image: alpine
        command: ["/bin/sh"]
        args:
        - "-c"
        - >
          apk add --update coreutils util-linux;
          mkdir /lib64 && ln -s /lib/libc.musl-x86_64.so.1 /lib64/ld-linux-x86-64.so.
↪2;
          sleep 3;
          (while true;
             do /pod-data/bin/shmtool read `ipcs -m | cut -d' ' -f 2 | sed  '/^$/d' |␣
↪tail -1`;
               sleep 1;
             done) | stdbuf -i0 nc -l -k -p 1234
        ports:
        - name: ncat
          containerPort: 1234
          protocol: TCP
        volumeMounts:
        - name: shared-data
          mountPath: /pod-data

# test with:
#  $ nc `kubectl get service/pod-ipc-sharing-examples -o jsonpath="{.spec.clusterIP}
↪"` 1234
#  Main app (pod-ipc-sharing-examples) says:  Tue May 7 20:46:03 UTC 2019
#  Main app (pod-ipc-sharing-examples) says:  Tue May 7 20:46:04 UTC 2019
#  Main app (pod-ipc-sharing-examples) says:  Tue May 7 20:46:05 UTC 2019
# $ kubectl delete deployment,svc -l app=pod-ipc-sharing-examples
# deployment.extensions "pod-ipc-sharing-examples" deleted
# service "pod-ipc-sharing-examples" deleted
```

The following example demonstrates how to share over a named pipe between containers:

Pod containers sharing over named pipe

```
---
kind: Service
apiVersion: v1
metadata:
  name: pod-sharing-examples
```

(continues on next page)

(continued from previous page)

```
  labels:
    app: pod-sharing-examples
spec:
  type: ClusterIP
  selector:
    app: pod-sharing-examples
  ports:
  - name: ncat
    protocol: TCP
    port: 1234
    targetPort: ncat

---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: pod-sharing-examples
  labels:
    app: pod-sharing-examples
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: pod-sharing-examples
    spec:
      volumes:
      # lifecyle containers as hooks share state using volumes
      - name: shared-data
        emptyDir: {}

      initContainers:
      # Setup the named pipe for inter-container communication
      - name: init-container
        image: alpine
        command: ['sh', '-c', "mkfifo /pod-data/piper"]
        volumeMounts:
        - name: shared-data
          mountPath: /pod-data

      containers:
      # Producer
      - name: producer-container
        image: alpine
        command: ["/bin/sh"]
        args: ["-c", "while true; do echo 'Main app (pod-sharing-examples) says: '␣
↪`date` >> /pod-data/piper; sleep 1;done"]
        volumeMounts:
        - name: shared-data
          mountPath: /pod-data

      # Consumer - read from the pipe and publish on 1234
      - name: consumer-container
        image: alpine
        command: ["/bin/sh"]
        args: ["-c", "apk add --update coreutils; tail -f /pod-data/piper | stdbuf -
↪i0 nc -l -k -p 1234"]
```

(continues on next page)

(continued from previous page)

```
        ports:
        - name: ncat
          containerPort: 1234
          protocol: TCP
        volumeMounts:
        - name: shared-data
          mountPath: /pod-data

# test with:
#  $ nc `kubectl get service/pod-sharing-examples -o jsonpath="{.spec.clusterIP}"`␣
↪1234
#  Main app says:  Thu May 2 20:48:56 UTC 2019
#  Main app says:  Thu May 2 20:49:53 UTC 2019
#  Main app says:  Thu May 2 20:49:56 UTC 2019
# $ kubectl delete deployment,svc -l app=pod-sharing-examples
# deployment.extensions "pod-sharing-examples" deleted
# service "pod-sharing-examples" deleted
```

The following example demonstrates how to share over the localhost network between containers:

Pod containers sharing over localhost network

```
---
kind: Service
apiVersion: v1
metadata:
  name: pod-sharing-network-examples
  labels:
    app: pod-sharing-network-examples
spec:
  type: ClusterIP
  selector:
    app: pod-sharing-network-examples
  ports:
  - name: ncat
    protocol: TCP
    port: 5678
    targetPort: ncat

---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: pod-sharing-network-examples
  labels:
    app: pod-sharing-network-examples
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: pod-sharing-network-examples
    spec:
      containers:
      # Producer
      - name: producer-container
        image: alpine
```

(continues on next page)

(continued from previous page)

```
        command: ["/bin/sh"]
        args: ["-c", "apk add --update coreutils; (while true; do echo 'Main app (pod-
↪sharing-network-examples) says: ' `date`; sleep 1; done) | stdbuf -i0 nc -lk -p 1234
↪"]

      # Consumer - read from the local port and publish on 5678
      - name: consumer-container
        image: alpine
        command: ["/bin/sh"]
        args: ["-c", "apk add --update coreutils; nc localhost 1234 | stdbuf -i0 nc -
↪l -k -p 5678"]
        ports:
        - name: ncat
          containerPort: 5678
          protocol: TCP

    # test with:
    #  $ nc `kubectl get service/pod-sharing-network-examples -o jsonpath="{.spec.
↪clusterIP}"` 5678
    #  Main app says:  Thu May 2 20:48:56 UTC 2019
    #  Main app says:  Thu May 2 20:49:53 UTC 2019
    #  Main app says:  Thu May 2 20:49:56 UTC 2019
    # $ kubectl delete deployment,svc -l app=pod-sharing-network-examples
    # deployment.extensions "pod-sharing-network-examples" deleted
    # service "pod-sharing-network-examples" deleted
```

Performance driven networking requirements are a concern with HPC. Often the solution is to bind an application directly to a specific host network adapter. Historically, the solution for this in containers has been to escalate the privileges of the container so that it is running in the host namespace, and this is achieved in in Kubernetes using the following approach:

```
...
spec:
  containers:
    - name: my-privileged-container
      securityContext:
        privileged: true
...
```

This **SHOULD** be avoided at all costs. This pushes the container into the host namespace for processes, network and storage. A critical side effect of this is that any port that the container consumes can conflict with host services, and will mean that **ONLY** a single instance of this container can run on any given host. Outside of these functional concerns, it is a serious source of security breach as the privileged container has full (root) access to the node including any applications (and containers) running there.

To date, the only valid exceptions discovered have been:

- Core daemon services running for the Kubernetes and OpenStack control plane that are deployed as containers but are node level services.
- Storage, Network, or Device Kubernetes plugins that need to deploy OS kernel drivers.

As a first step to resolving a networking issue, the Kubernetes and Platform management team should always be approached to help resolve architectural issues to avoid this approach. In the event of not being able to reconcile the requirement, then the following `hostNetwork` solution should be attempted first:

```
...
spec:
  containers:
    - name: my-hostnetwork-container
      securityContext:
        hostNetwork: true
```

Use of Services

Service resources should be defined in the same template file as the associated application deployment and ordered at the top. This will ensure that service related environment variables will be passed into the deployment at scheduling time. It is good practice to only have a single Service resource per deployment that covers the port mapping/exposure for each application port. It is also important to only have one deployment per Service as it will make debugging considerably harder mapping a Service to more than one application. As part of this, ensure that the selector definition is specific to the fully qualified deployment including release and version to prevent leakage across multiple deployment versions. Fully qualify port definitions with name, port, protocol and targetPort so that the interface is self documenting. Using names for targetPort the same as name is encouraged as this can give useful hints as to the function of the container interface.

Service resource with fully qualified port description and specific selector

```
---
apiVersion: v1
kind: Service
metadata:
name: tango-rest-{{ template "tango-chart-example.name" . }}-{{ .Release.Name }}
namespace: {{ .Release.Namespace }}
labels:
  app.kubernetes.io/name: tango-rest-{{ template "tango-chart-example.name" . }}
  app.kubernetes.io/instance: "{{ .Release.Name }}"
  app.kubernetes.io/managed-by: "{{ .Release.Service }}"
  helm.sh/chart: "{{ template "tango-chart-example.chart" . }}"
spec:
  type: ClusterIP
  ports:
  - name: rest
    protocol: TCP
    port: 80
    targetPort: rest
  selector:
    app.kubernetes.io/name: tango-rest-{{ template "tango-chart-example.name" . }}
    app.kubernetes.io/instance: "{{ .Release.Name }}"
```

type: ClusterIP is the default and should almost always be used and declared. NodePort should only be used under exceptional circumstances as it will reserve a fixed port on the underlying node using up the limited node port address range resource.

Only expose ports that are actually needed external to the deployment. This will help reduce clutter and reduce the surface area for attack on an application.

Use of Ingress

A Helm chart represents an application to be deployed, so it follows that it is best practice to have a single Ingress resource per chart. This represents the single frontend for an application that exposes it to the outside world (relative to the Kubernetes cluster). If a chart seemingly requires multiple hostnames and/or has services that want to inhabit the same port or URI space, then consider splitting this into multiple charts so that the component application can be published independently.

It is useful to parameterise the control of SSL/TLS configuration so that this can be opted in to in various deployment strategies (as below).

One Ingress per chart with TLS parameterised

```
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
name: rest-api-{{ template "tango-chart-example.name" . }}-{{ .Release.Name }}
labels:
  app.kubernetes.io/name: rest-{{ template "tango-chart-example.name" . }}
  app.kubernetes.io/instance: "{{ .Release.Name }}"
  app.kubernetes.io/managed-by: "{{ .Release.Service }}"
  helm.sh/chart: "{{ template "tango-chart-example.chart" . }}"
annotations:
  {{- range $key, $value := .Values.ingress.annotations }}
  {{ $key }}: {{ $value | quote }}
  {{- end }}
spec:
  rules:
    - host: {{ .Values.ingress.hostname }}
      http:
        paths:
          - path: /
            backend:
              serviceName:  tango-rest-{{ template "tango-chart-example.name" . }}-{{␣
↪.Release.Name }}
              servicePort: 80
{{- if .Values.ingress.tls.enabled }}
  tls:
    - secretName: {{ tpl .Values.ingress.tls.secretname . }}
      hosts:
        - {{ tpl .Values.ingress.hostname . }}
{{- end -}}
```

Scheduling and running cloud native application suites

Security

Security covers many things, but this section will focus on RBAC and network Policies.

Roles

Kubernetes will implement role based access control which will be used to control external and internal user access to scheduling and consuming resources.

While it is possible to create `serviceAccounts` to modify the privileges for a deployment, this should generally be avoided so that the access control profile of the deploying user can be inherited at launch time.

Do not create `ClusterRole` and `ClusterRoleBinding` resources and/or allocate these to `ServiceAccounts` used in a deployment as these have extended system wide access rights. `Role` and

`RoleBinding` are scoped to the deployment `Namespace` so limit the scope for damage.

Pod Security Policies

Pod Security Policies will affect what can be requested in the securityContext section.

It should be assumed that Kubernetes clusters will run restrictive Pod security policies, so it should be expected that:

- `Pods` do not need to access resources outside the current `Namespace`.
- `Pods` do not run as `privileged: true` and will not have privilege escalation.
- `hostNetwork` activation will require discussion with operations.
- `hostIPC` will be unavailable.
- `hostPID` will be unavailable.
- Containers should run as a non-root user.
- host ports will be restricted.
- host paths will be restricted (`hostPath` mounts).
- it maybe required to have read only root filesystem (layer in container).
- Capabilities maybe dropped and a restricted list put in place to determine what can be added.
- it should be expected that the `default` service account credentials will **NOT** be mounted into the running containers by default - applications should rarely need to query the Kubernetes API, so access will be removed by default.

In general, only system level deployments such as Kubernetes control plane components (eg: adminsion controllers, device drivers, Operators, etc.) are the only deployments that should have cluster level rights.

Network Policies

Explicit Network Policies are encouraged to restrict unintended access across deployments, and to secure applications from some forms of intrusion.

The following restricts access to the deployed TangoDB to only the DatabaseDS application.

One Ingress per chart with TLS parameterised

```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: tangodb-{{ template "tango-chart-example.name" . }}-{{ .Release.Name }}-
↪network-policy
spec:
  podSelector:
    matchLabels:
      app.kubernetes.io/name: tangodb-{{ template "tango-chart-example.name" . }}
      app.kubernetes.io/instance: "{{ .Release.Name }}"
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
```

(continues on next page)

(continued from previous page)

```
    - podSelector:
    # enable the DatabaseDS interface
        matchLabels:
          app.kubernetes.io/name: databaseds-{{ template "tango-chart-example.name" .␣
↪}}
          app.kubernetes.io/instance: "{{ .Release.Name }}"
    ports:
    - name: ds
      protocol: TCP
      port: 10000
  egress:
  - to:
    # anywhere in the standard Pod Network address range to all ports
    - ipBlock:
        cidr: 10.0.0.0/16
```

Images, Tags, and pullPolicy

Only use images from trusted sources. In most cases this should be only from the official SKA repository, with a few exceptions such as the core vender supported images for key services such as MySQL. It is anticipated that in the future the SKA will host mirrors and/or pull-through caches for key external software components, and will then firewall off access to external repositories that are not explicitly trusted.

As a general rule, stable image tags should be used for images that at least include the Major and Minor version number of Semantic Versioning eg: `mysql:5.27`. As curated images come from trusted sources, this ensures that the deployment process gets a functionally stable starting point that will still accrue bug fixing and security patching over time. Do **NOT** use the `latest` tag as it is likely that this will break your application in future as it gives no way of guaranteeing feature parity and stability.

In Helm Charts, it is good practice to parameterise the registry, image and tag of each container so that these can be varied in different environment deployments by changing `values`. Also parameterise the `pullPolicy` so that communication with the registry at container boot time can be easily turned on and off.

```
...
containers:
- name: tangodb
  image: "{{ .Values.tangodb.image.registry }}/{{ .Values.tangodb.image.
↪image }}:{{ .Values.tangodb.image.tag }}"
  imagePullPolicy: {{ .Values.tangodb.image.pullPolicy }}
```

Resource reservations and constraints

Compute platform level resources encompass:

- Memory.
- CPU.
- Plugin based devices.
- Extended resources - configured node level logical resources.

Resources can be either specified in terms of:

- Limits - the maximum amount of resource a container is allowed to consume before it maybe restarted or evicted.

- Requests - the amount of resource a container requires to be available before it will be scheduled.

Limits and requests are specified at the individual container level:

```
...
containers:
- name: tango-device-thing
  resources:
    requests:
      cpu: 4000m    # 4 cores
      memory: 512M  # 0.5GB
      skatelescope.org/widget: 3
    limits:
      cpu: 8000m    # 8 cores
      memory: 1024M  # 1GB
```

Resource requirements should be explicitly set both in terms of requests and limits (not normally applicable to extended resources) as this can be used by the scheduler to determine load balancing policy, and to determine when an application is misbehaving. These parameters should be set as configured `values.yaml` parameters.

Restarts

Containers should be designed to cleanly crash - the main process should exit on a fatal error (no internal restart). This then will ensure that the configured `livenessProbe` and `readinessProbe` function correctly and where necessary, remove the affected `Pod` from `Services` ensuring that there are no dead service connections.

Logging

The SKA has adopted *SKA Log Message Format* as the logging standard to be used by all SKA software. This should be considered a base line standard and will be decorated with additional data by an infrastructure wide integrated logging solution (eg: ElasticStack). To ensure compliance with this, all containers must log to `stdout/stderr` and/or be configured to log to `syslog`. Connection to `syslog` should be configurable using *standard container mechanisms* such as mounted files (handled by `ConfigMaps`) or environment variables. This will ensure that any deployed application can be automatically plugged into the infrastructure wide logging and monitoring solution. A simple way to achieve this is to use a logging client library that is dynamically configurable for output destination such as `import logging` for `Python`.

Metrics

Each `Pod` should have an application metrics handler that emits the adopted container standard format. For efficency purposes this should be amalgamated with the `livenessProbe` and `readinessProbe`.

Scheduling

Scheduling in Kubernetes enables the resources of the entire cluster to be allocated using a fine grained model. These resources can be partitioned according to user policies, namespaces, and quotas. The default scheduler is a comprehensive rules processing engine that should be able to satisfy most needs.

The primary mechanism for routing incoming tasks to execution is by having a labelling system throughout the cluster that reflects the distribution profile of workloads and types of resources required, coupled with Node and Pod affinity/anti-affinity rules. These are applied like a sieve to the available resources that the Scheduler keeps track of to determine if resources are available and where the next Pod can be placed.

Scheduling on Kubernetes behaves similarly to a force directed graph, in that the tensions between the interdependent rules form the pressures of the spring bars that influence relative placement across the cluster.

When creating scheduling constraints, attempt to keep them as generic as possible. Concentrate on declaring rules related to the individual Helm chart and the current chart in relation to any dependent charts (subcharts). Avoid coding in node specific requirements. Often it is more efficient to outsource the rules to the `values.yaml` file as they are almost guaranteed to change between environments.

```
---
...
{{- with .Values.nodeSelector }}
      nodeSelector:
{{ toYaml . | indent 8 }}
{{- end }}
{{- with .Values.affinity }}
      affinity:
{{ toYaml . | indent 8 }}
{{- end }}
{{- with .Values.tolerations }}
      tolerations:
{{ toYaml . | indent 8 }}
{{- end }}
...
```

Always remember that the Kubernetes API is declarative and expect that deployments will use the `apply` semantics of kubectl, with the scheduler constantly trying to move the system towards the desired state as and when resources become available as well as in response to failures. This means that scheduling is not guaranteed, so any downstream depedencies must be able to cope with that (also a tenent of micro-services architecture).

Examples of scheduling control patterns

The below scheduling scenarios are run using the following conditions:

- container replicas launched using a sleep command in busybox, defined in a StatefulSet.
- Specific node.
- Type of node.
- Density - 1 per node, n per node.
- Position next another Pod - specific Pod, or Pod type.
- Soft and hard rules.
- A four node cluster - master and three minions.
- The nodes have been split into two groups: rack01 - k8s-master-0 and k8s-minion-0, and rack02 - k8s-minion-1, and k8s-minion-2.
- The master node has the labels: node-role.kubernetes.io/headnode, and node-role.kubernetes.io/master.

The aim is to demonstrate how the scheduler works, and how to configure for the common use cases.

obs1 and obs2 - nodeAffinity

Use nodeSelector to force all 3 replicas onto `rack: rack01` for obs1-rack01 and `rack02` for obs2-rack02:

node select rack01 for obs1-rack01 and rack02 for obs2-rack02

```
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: obs1-rack01
  labels:
    group: scheduling-examples
    app: obs1
spec:
  replicas: 3
  selector:
    matchLabels:
      app: obs1
  serviceName: obs1
  template:
    metadata:
      labels:
        group: scheduling-examples
        app: obs1
      annotations:
        description: node select rack01
    spec:
      containers:
      - image: busybox:1.28.3
        name: obs1-rack01
        command: ["sleep", "365d"]
      nodeSelector:
        rack: rack01

---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: obs2-rack02
  labels:
    group: scheduling-examples
    app: obs2
spec:
  replicas: 3
  selector:
    matchLabels:
      app: obs2
  serviceName: obs2
  template:
    metadata:
      labels:
        group: scheduling-examples
        app: obs2
      annotations:
        description: node select rack02
    spec:
      containers:
      - image: busybox:1.28.3
        name: obs2-rack02
        command: ["sleep", "365d"]
      nodeSelector:
        rack: rack02
```

Scenario obs1 - run 3 Pods on hosts allocated to rack01. Only nodes master-0, and minion-0 are used reflecting rack01.

```
NAME           DESC                STATUS  NODE
obs1-rack01-0 node select rack01 Running k8s-master-0
obs1-rack01-1 node select rack01 Running k8s-minion-0
obs1-rack01-2 node select rack01 Running k8s-master-0
```

and for Scenario obs2 - run 3 Pods on hosts allocated to rack02. Only minion-1 and minion-2 are used reflecting rack02.

```
NAME           DESC                STATUS  NODE
obs2-rack02-0 node select rack02 Running k8s-minion-2
obs2-rack02-1 node select rack02 Running k8s-minion-1
obs2-rack02-2 node select rack02 Running k8s-minion-2
```

obs3 - nodeAffinity exclussion

Use nodeAffinity `operator:  NotIn` rules to exclude the master node from scheduling:

nodeAffinity NotIn master

```
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: obs3-node-affinity-not-master
  labels:
    group: scheduling-examples
    app: obs3
spec:
  replicas: 4
  selector:
    matchLabels:
      app: obs3
  serviceName: obs3
  template:
    metadata:
      labels:
        group: scheduling-examples
        app: obs3
      annotations:
        description: nodeAffinity NotIn master
    spec:
      containers:
      - image: busybox:1.28.3
        name: obs3-node-affinity-not-master
        command: ["sleep", "365d"]
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: node-role.kubernetes.io/master
                operator: NotIn
                values:
                - ""
```

Scenario obs3 - run 4 Pods on any host so long as they are not labelled node-role.kubernetes.io/master. In this case minion-0 and minion-1 have been selected minion-2 could also have been used.

```
NAME                            DESC                      STATUS  NODE
obs3-node-affinity-not-master-0 nodeAffinity NotIn master Running k8s-minion-1
obs3-node-affinity-not-master-1 nodeAffinity NotIn master Running k8s-minion-0
obs3-node-affinity-not-master-2 nodeAffinity NotIn master Running k8s-minion-1
obs3-node-affinity-not-master-3 nodeAffinity NotIn master Running k8s-minion-0
```

obs4 - nodeAntiAffinity

Use podAffinity (hard requiredDuringSchedulingIgnoredDuringExecution) to position on the same node as obs1-rack01, and nodeAntiAffinity to (soft preferredDuringSchedulingIgnoredDuringExecution) exclude the node labelled 'node-role.kubernetes.io/headnode' from scheduling:

podAffinity require obs1-rack01, nodeAntiAffinity prefer headnode

```
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: obs4-pod-affinity-obs1-pref-not-headnode
  labels:
    group: scheduling-examples
    app: obs4
spec:
  replicas: 5
  selector:
    matchLabels:
      app: obs4
  serviceName: obs4
  template:
    metadata:
      labels:
        group: scheduling-examples
        app: obs4
      annotations:
        description: podAffinity req obs1, nodeAntiAffinity pref headnode
    spec:
      containers:
      - image: busybox:1.28.3
        name: obs4-pod-affinity-obs1-pref-not-headnode
        command: ["sleep", "365d"]
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - obs1
            topologyKey: kubernetes.io/hostname
        nodeAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 100
```

(continues on next page)

(continued from previous page)

```
            preference:
              matchExpressions:
              - key: node-role.kubernetes.io/headnode
                operator: NotIn
                values:
                - ""
```

Scenario obs4 - run 5 Pods using required Pod Affinity with obs1 and preferred Node Anti Affinity with headnode (master label). Pods have been scheduled on minion-0 and master-0 as this is where obs1 is. This is further compounded by the anti affinity rule with headnode where only one replica is on master-0.

```
NAME                                       DESC                                         ␣
↪ STATUS  NODE
obs4-pod-affinity-obs1-pref-not-headnode-0 podAffinity req obs1, nodeAntiAffinity␣
↪pref headnode Running k8s-minion-0
obs4-pod-affinity-obs1-pref-not-headnode-1 podAffinity req obs1, nodeAntiAffinity␣
↪pref headnode Running k8s-minion-0
obs4-pod-affinity-obs1-pref-not-headnode-2 podAffinity req obs1, nodeAntiAffinity␣
↪pref headnode Running k8s-minion-0
obs4-pod-affinity-obs1-pref-not-headnode-3 podAffinity req obs1, nodeAntiAffinity␣
↪pref headnode Running k8s-master-0
obs4-pod-affinity-obs1-pref-not-headnode-4 podAffinity req obs1, nodeAntiAffinity␣
↪pref headnode Running k8s-minion-0
```

obs5 - podAntiAffinity

Use podAntiAffinity (hard requiredDuringSchedulingIgnoredDuringExecution) to ensure only one instance of self per node (topologyKey: “kubernetes.io/hostname”), and podAffinity to require a position on the same node as obs3:

podAntiAffinity require self and podAffinity require obs3

```
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: obs5-pod-one-per-node-and-obs3
  labels:
    group: scheduling-examples
    app: obs5
spec:
  replicas: 5
  selector:
    matchLabels:
      app: obs5
  serviceName: obs5
  template:
    metadata:
      labels:
        group: scheduling-examples
        app: obs5
      annotations:
        description: podAntiAffinity req self, podAffinity req obs3
    spec:
      containers:
      - image: busybox:1.28.3
```

(continues on next page)

(continued from previous page)

```
        name: obs5-pod-one-per-node-and-obs3
        command: ["sleep", "365d"]
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - obs5
            topologyKey: "kubernetes.io/hostname"
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - obs3
            topologyKey: "kubernetes.io/hostname"
```

Scenario obs5 - run 3 Pods using required Pod Anti Affinity with self (force schedule one per node) and require Pod Affinity with obs3. This has forced scheduling of one per node, and because obs3 is only running on two different nodes the 3rd replica is in a constant state of Pending. Pod Affinity is described with a topology key that is

obs6 - Taint NoSchedule

kubernetes.io/hostname ie. the node identifier. The topology key sets the scope for implementing the rule, so could be a node, a group of nodes, an OS or device classificaton etc.

```
NAME                                DESC                                           ␣
↪STATUS  NODE
obs5-pod-one-per-node-and-obs3-0 podAntiAffinity req self, podAffinity req obs3␣
↪Running k8s-minion-0
obs5-pod-one-per-node-and-obs3-1 podAntiAffinity req self, podAffinity req obs3␣
↪Running k8s-minion-1
obs5-pod-one-per-node-and-obs3-2 podAntiAffinity req self, podAffinity req obs3␣
↪Pending <none>
```

First, the master node is tainted to disallow scheduling with `kubectl cordon <master node>`.

Use nodeSelector to force all 3 replicas onto `rack:  rack01`, but this will fail to schedule as the taint will not allow it so subsequently forced onto minion-0:

node select rack01, but trapped by Taint NoSchedule

```
---
# kubectl taint nodes k8s-master-0 key1=value1:NoSchedule, or kubectl cordon k8s-
↪master-0
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: obs6-rack01-taint
  labels:
```

(continues on next page)

(continued from previous page)

```
    group: scheduling-examples
    app: obs6
spec:
  replicas: 3
  selector:
    matchLabels:
      app: obs6
  serviceName: obs6
  template:
    metadata:
      labels:
        group: scheduling-examples
        app: obs6
      annotations:
        description: node select rack01, but trapped by Taint NoSchedule
    spec:
      containers:
      - image: busybox:1.28.3
        name: obs6-rack01-taint
        command: ["sleep", "365d"]
      nodeSelector:
        rack: rack01
```

The resulting schedule is:

```
NAME                READY STATUS  RESTARTS AGE IP              NODE NOMINATED NODE
obs6-rack01-taint-0 1/1   Running 0        32s 192.168.105.180 k8s-minion-0 <none>
obs6-rack01-taint-1 1/1   Running 0        31s 192.168.105.177 k8s-minion-0 <none>
obs6-rack01-taint-2 1/1   Running 0        29s 192.168.105.181 k8s-minion-0 <none>
```

For obs6, a StatefulSet that has nodeSelector:

```
nodeSelector:
rack: rack01
```

The result shows that of the two nodes (ks-master-0, and k8s-minion-0) in rack01, only k8s-minion-0 is available for these Pods.

obs7 - add toleration

Repeat obs6 as obs7 but add a toleration to the NoSchedule taint:

node select rack01, with Toleration to Taint NoSchedule

```
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: obs7-rack01-taint-and-toleration
  labels:
    group: scheduling-examples
    app: obs7
spec:
  replicas: 3
  selector:
```

(continues on next page)

(continued from previous page)

```
    matchLabels:
      app: obs7
  serviceName: obs7
  template:
    metadata:
      labels:
        group: scheduling-examples
        app: obs7
      annotations:
        description: node select rack01, with Tolleration to Taint NoSchedule
    spec:
      containers:
      - image: busybox:1.28.3
        name: obs7-rack01-taint-and-toleration
        command: ["sleep", "365d"]
      nodeSelector:
        rack: rack01
      tolerations:
      - key: "key1"
        operator: "Equal"
        value: "value1"
        effect: "NoSchedule"
```

Now with the added a Toleration to the Taint, we have the following:

```
NAME                                 READY STATUS RESTARTS AGE IP              NODE␣
↪NOMINATED NODE
obs7-rack01-taint-and-toleration-0 1/1   Running 0       33s 192.168.105.184 k8s-
↪minion-0 <none>
obs7-rack01-taint-and-toleration-1 1/1   Running 0       32s 192.168.72.27   k8s-
↪master-0 <none>
obs7-rack01-taint-and-toleration-2 1/1   Running 0       31s 192.168.105.182 k8s-
↪minion-0 <none>
```

For a StatefulSet that has nodeSelector and Tolerations:

```
nodeSelector:
  rack: rack01
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoSchedule"
```

The result shows that the two nodes k8s-master-0, and k8s-minion-0 in rack01, are available for these Pods.

2.11.7 Hosting a docker image on the *Central Artefact Repository*

As part of our goal to align all developmental efforts to one standard, we have documented a procedure of how we would like all the *SKA* developers to version tag their docker images and what process to follow in ensuring that they are able to make use of the existing Gitlab CI/CD pipeline to automate the building of docker images, for now, and have them published on the *SKA* docker registry which is hosted on *Central Artefact Repository*.

Tagging the docker image

To explicitly tag a docker image run the following command:

```
$ docker tag <source_image> artefact.skatelescope.org/<repository_name>/<image_name>:
↪<tag_name>
```

This command will create an alias by the name of the `<image_name>` that refers to the `<source_image>`.

Note that naming and tagging conventions are outlined in the containerisation standards. And those should follow the same semantic versioning used for the repository.

Uploading the docker image to the *Central Artefact Repository*

Once the docker image has been built and tagged, it can then be uploaded to the *Central Artefact Repository* registry by executing the following command:

```
$ docker push artefact.skatelescope.org/<repository_name>/<image_name>:<tag_name>
```

2.12 Documenting a project

Documenting a project is the key to make it usable, understandable and maintainable, making the difference between a technically excellent software and a successful software. SKA has defined a set of standards and practices that shall be implemented when developing software documentation. A more comprehensive set of resources can be found online at:

- A beginner guide to writing documentation

2.12.1 What to document

Based on the documentation provided everyone should be able to:

- understand the problem the project is trying to solve
- understand how the project is implemented
- know exactly what this projects depends on
- download and build the project
- execute the project tests
- run the project

In order to achieve this goal any structure can be used allowing free text as well as code-extracted documentation.

Documenting the Public API

When it comes to software systems, such as services or libraries, it is of paramount importance that the public API exposed by the software component is clearly captured and documented.

2.12.2 How to document

Documentation on git

Each software repository shall contain its documentation as part of the code included in a git repository. In this way the documentation will be released with the same cadence as the code and it will always be possible to trace a particular version of the documentation to a particular version of the code documented.

Free text documentation must be placed in a `docs` folder in the upper level of the repository structure. Sphinx generates automatically extracted files under this folder as well, wherever there are docstrings in the code.

Using sphinx

Documentation must be realised using the sphinx package and Restructured Text . SKA provides a predefined sphinx template for this purpose in the SKA Python skeleton project. Every project shall use the same `docs` folder as a starting point for assembling their own documentation.

Sphinx can be used to generate text documents such as this portal, but it also provides capabilities to automatically extract and parse code documentation, or docstrings. Refer to the *Python Coding Guidelines* for more information.

Extracting documentation from code

Todo:

- add hello world class with parameters to the SKA Python Skeleton Project
- add code snippet here as an example of additional documentation which is decoupled from code, and describe the pitfalls of separating documentation from the code.

2.12.3 Integration into the Developer Portal

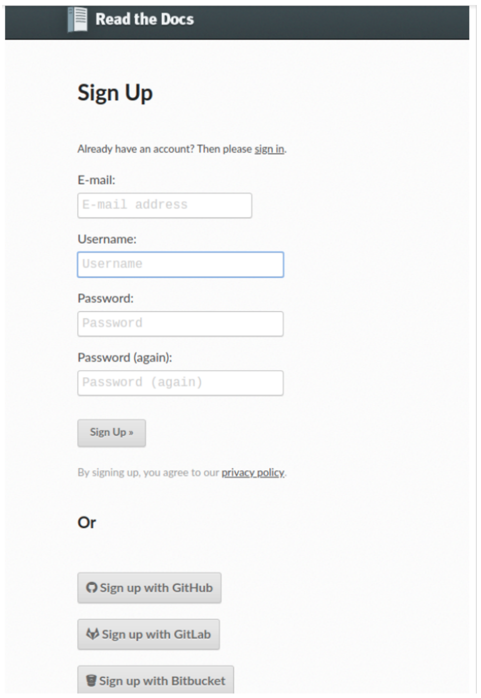
The developer portal is hosted on ReadTheDocs. On the *Alpahbetical list of projects and subgroups* page a list of all the projects that are hosted on GitLab is available, with badges to show the build status of the project's documentation. Each badge is also a hyperlink to the project's documentation that you can click on.

Every SKA project's documentation is hosted on Readthedocs as a *subproject* of the developer portal, so that all projects have a common URL for easier search-ability. For example: whereas the developer portal's URL is https://developer.skatelescope.org, the ska_python_skeleton project is at https://developer.skatelescope.org/projects/ska-python-skeleton.

In order to add the project's documentation as a subproject on Readthedocs, a project must first be imported into Readthedocs.

Register on ReadTheDocs

Developers working on the SKA are members of the ska-telescope organisation on GitLab. Registering an account using the OAuth credentials on ReadTheDocs is recommended, because then the integration between the SKA GitLab and SKA ReadTheDocs services is done automatically. The integrations can also be set up manually later, and is not difficult.



Sign up / sign in with GitLab account

Import project to ReadTheDocs

After signing in, one lands on the Dashboard, and the steps for importing a project are pretty self-explanatory from here. While importing the project **name** should be the *ska-telescope-* and project's gitlab slug (part in the url after https://gitlab.com/ska-telescope/), i.e. *ska-telescope-ska-python-skeleton*. After the project is imported successfully, name should be changed to the name of the project as listed in Gitlab project site. Project name could be changed in the *Admin* page of Read the Docs project site. As a last step, *kurtcobain-19* account should be added to the project as a maintainer for the system team to manage the documentation later on.

Add project as a sub-project on ReadTheDocs

A sub-project must be added by a user with Maintainer privileges on the main project.

Currently only the System Team members have these permissions. Please ask on the Slack channel #team-system-support to have your project added.

For more information on how to add a subproject, go to Read The Docs.

2.13 Software Package Release Procedure

Whilst source code related to software artefacts are hosted on GitLab, the delivered runtime artefacts for the SKAO are maintained and curated in the Central Artefact Repository. It is here that they will navigate through the process of verification and testing with the aim of finally being promoted to a state ready for production release.

Artefacts that are candidates for promotion to the Central Artefact Repository will need to follow the ADR-25 - Software naming conventions, and must conform to the *Definition of Done*.

These conventions are designed to integrate with the leading packaging and deployment tools commonly available for each artefact type.

For intermediate artefacts, it is recommended that the builtin packages (repository) features available in GitLab are used. These can be accessed directly in the GitLab CI/CD pipeline. Examples of these are given below for OCI Images, PyPi packages, and Raw (Generic) artefact artefacts.

Table of Contents

2.13.1 Central Artefact Repository

The Central Artefact Repository provides the storage and library management facilities for artefacts throughout the Software Development Life Cycle. Being central to the SDLC means that it is highly desirable that the Repository is stable, long lived, and can evolve with the SKAO requirements through the different stages of DevSecOps maturity, and the life time of the project.

An Artefact Repository is essentially a content management system for software artefacts delivered in their published form. The Artefact Repository makes it possible to publish and retrieve the artefacts using the native tools, processes and protocols commonly associated with the related programming language and/or language based framework. The Artefacts are versioned, and decorated with extensible metadata that is used to help manage the life cycle of the Artefacts. The Central Artefact Repository provides APIs and reporting interfaces to provide integration into extended DevSecOps processes such as CI/CD, BoM and dependency management, license management, provenance, vulnerability scanning and release management. It also provides controlled access to Artefacts through IAM, and offers notary features for provenance through TLS/SSL and signatures.

The purpose of the Central Artefact Repository within the context of the SKAO, is to provide a controlled single source of truth for all artefacts that enter the software delivery and verification processes through to the curation and maintenance of approved software artefacts available for production deployment and historical reference for the life time of the Observatory. This means that the Central Artefact Repository not only holds the canonical reference versions of all artefacts within the SKAO landscape, but it also holds the stateful context of these artefacts as they progress through their continuous life-cycle from development to production deployment, to decommissioning.

2.13.2 Deployment

The Central Artefact Repository plays a key role in regulating the flow of artefacts throughout the Software Development Life Cycle. It is highly integrated into all the phases of software development, build, test, and publish. In this position, it can ensure that only approved software artefacts are included in composite products, and subsequently delivered to the production environments.

Whilst it is important for the Repository to be highly available and performant in the context of the SDLC, it is not responsible for directly servicing the delivery of artefacts into the operational environments. This will be manged by high speed delivery services and caches.

The repository is based on *Nexus* Repository Manager 3 deployed on an independent UK based hosting service. The core deployment is nexus-oss-edition with the nexus-core-feature, nexus-cma-feature, nexus-oss-feature features enabled.

LDAP authentication has been integrated for SKAO administration purposes, with an additional minimal set of accounts managed for publishing artefacts. All repositories are enable read-only for anonymous access. Additionally, email has been integrated for handling task notifications.

Configured Repositories

The SKAO aims to maintain repositories with native interface support for the core languages and frameworks used for software development within the project. This includes:

- Docker (OCI Images)
- Helm (Charts)
- PyPi (Wheels/Source Distributions)
- Conan (C/C++)
- NPM (Node)
- Maven (Java)
- GitLFS (Large File Support)
- Apt (Debian)

- Yum (Fedora)

Additionally, there are also upsteam proxy/caching facilities available for:

- Docker (OCI Images - only available inside AWS VPC)
- Helm - https://artefact.skatelescope.org/repository/helm-proxy/ (from https://charts.helm.sh/stable)
- PyPi - https://artefact.skatelescope.org/repository/pypi-all/ (from pypi.python.org and include pypi-internal)
- Conda - https://artefact.skatelescope.org/repository/conda-proxy/ (from https://repo.continuum.io/pkgs/)
- Conan - https://artefact.skatelescope.org/repository/conan-proxy/ (from https://conan.bintray.com)
- NPM - https://artefact.skatelescope.org/repository/npm-all/ (from https://registry.npmjs.org and include npm-internal which is not active yet)
- Maven - https://artefact.skatelescope.org/repository/maven-public/ (from maven-release, maven-snapshots, https://repo1.maven.org/maven2/)
- Apt - https://artefact.skatelescope.org/repository/ubuntu-archive/, https://artefact.skatelescope.org/repository/ubuntu18.04-proxy/, and https://artefact.skatelescope.org/repository/ubuntu20.04-proxy/
- Yum - CentOS7 https://artefact.skatelescope.org/repository/yum_centos_7-internal/ (from http://download.fedoraproject.org/pub/epel/7/x86_64 and yum_centos_7-internal), CentOS8 https://artefact.skatelescope.org/repository/yum_centos_8-internal/ (from http://download.fedoraproject.org/pub/epel/8/Everything/x86_64 and yum_centos_8-internal)
- Go Lang - https://artefact.skatelescope.org/repository/go-proxy/ (from https://golang.org/pkg/)

Finally, there are repositories that utilise the Nexus Raw format to provide library space for the following:

- Ansible
- Raw objects (binary, text etc.)

Versioning

As part of the goal to align all developmental efforts to one standard, we have documented a procedure of how we would like all the *SKAO* developers to version their releases and what process to follow in ensuring that they are able to make use of the existing Gitlab CI/CD pipeline to automate the building of artefacts. This standard is defined in detail for each artefact type in ADR-25 - Software naming conventions. These convetions are fundamentally derived from the Sematic Versioning standard 2.0.0. In a nutshell, this follows a dotted numeric notation for *Major*.'Minor'.'Patch' eg: *1.2.3*, but please check the above guidance for the details, and language specifics.

Artefact Naming

In additon to the semantic versioning scheme, when publishing artefacts to the repositories, the naming conventions for the artefact must be adhered to (also detailed in ADR-25 - Software naming conventions). The general rules are:

- Prefix the artefact with the namespace'd name of the GitLab repository that holds the source code
- Name the artefact after it's core function
- Observe the Semantic Versioning standard for this kind of artefact
- Do not use generic versions such as 'latest' tags for container images
- Published artefacts are immutable - do not re-release an artefact version
- Filters and cleanup policies are implemented to purge artefacts that do not adhere to standards, and where possible validation hooks will deny publishing of incorrectly named/versioned artefacts. For instance images with the tag 'latest' will be trapped by a cleanup policy.

Deploying Artefacts

While the Central Artefact Repository is available for anonymous browsing and pulling of artefacts, all artefacts must be published via the SKAO GitLab CI/CD infrastructure. The GitLab Runner environment provides the credentials. These are specified in the *full list of environment variables*, with examples given below.

OCI Image

The OCI Image repository is locate at https://artefact.skatelescope.org/#browse/browse:docker-internal .

Example: publish an OCI Image for the tango-cpp base image from ska-tango-images

```
# checkout https://gitlab.com/ska-telescope/ska-tango-images
# Build and tag the image for a fictitious version 9.3.4 repo-prefix=ska-tango-images␣
↪core-function=tango-cpp
docker build -t ${CAR_OCI_REGISTRY_HOST}/ska-tango-images/tango-cpp:9.3.4 .
# login to the registry
echo ${CAR_OCI_REGISTRY_PASSWORD} | docker login --username ${CAR_OCI_REGISTRY_
↪USERNAME} --password-stdin ${CAR_OCI_REGISTRY_HOST}
# Push the image
docker push ${CAR_OCI_REGISTRY_HOST}/ska-tango-images/tango-cpp:9.3.4
This image has been published at https://artefact.skatelescope.org/#browse/
↪browse:docker-internal:v2%2Fska-tango-images%2Ftango-cpp%2Ftags%2F9.3.4
```

Using the GitLab OCI Registry

The GitLab OCI Registry is a useful service for storing intermediate images, that are required between job steps within a Pipeline or between piplines (eg: where base images are used and subsequent pipeline triggers). The following is an example of interacting with a project specific repository:

```
build oci image:
  image: docker:19.03.12
  stage: build
  services:
    - docker:19.03.12-dind
  variables:
    IMAGE_TAG: $CI_REGISTRY_IMAGE:$SEMANTIC_VERSION
  script:
    - echo "$CI_JOB_TOKEN $CI" | docker login -u $CI_JOB_USER --password-stdin $CI_
↪REGISTRY
    - docker build -t $IMAGE_TAG .
    - docker push $IMAGE_TAG
```

Helm Chart

Helm Charts are published to the Central Artefact Repository in a native repository, however (at the time of writing) there is a move in the Cloud Native community to extend the storage of Charts to OCI compliant repositories. This support has been made available in `` `helm` `` and is supported by both Nexus and the GitLab Container Registry.

Package and publish Helm Charts to the SKAO Helm Chart Repository

The process of packaging and publishing Helm charts to the SKAO repository is very simple. A few lines are needed in the `.gitlab-ci.yml` file, and the project needs to have a `charts` directory under the root of the project, that contains all your project's charts. If the `charts` folder is not under the project root, a line can be added in the CI job to first change to the directory containing this `charts` directory, however this is discouraged. For further information on best practices with regards to specifically the folder structure of charts, refer to The Chart Best Practices Guide, and also to our own set of *Helm Best Practices*.

As an example, let's take the following project structure:

```
.
├── my-project
│   ├── charts
│   |   └── my-first-chart
│   |   └── my-second-chart
│   ├── .gitlab-ci.yml
│   └── README.md
```

Refer to the Helm repository guide to understand how to package a chart, but to package and publish the two charts in the above example, simply add the following code to your `.gitlab-ci.yml` file and also ensure that your pipeline has a *publish* stage:

```
# uncomment and specify specific charts to publish
# variables:
#   CHARTS_TO_PUBLISH: my-first-chart my-second-chart

# Ensure your .gitlab-ci.yml has "publish" stage defined!
include:
  - project: 'ska-telescope/templates-repository'
    file: 'gitlab-ci/includes/helm_publish.yml'
```

In case you only want to publish a sub-set of the charts in your project, you can uncomment the variable declaration lines (above) in the job specifying the `CHARTS_TO_PUBLISH` variable. Note that the list in the above example is redundant, since the default behaviour is to publish all the charts found in the `charts/` folder, and in this case there are only those two charts.

The CI job that is included using the above lines of code takes care of packaging the chart in a temporary directory and pushes it to the SKAO repository. The job runs manually, which means that you need to trigger it on the Gitlab web UI in the CI/CD pipeline view. Note, triggering the job, you can specify the `CHARTS_TO_PUBLISH` variable before the job executes again, however, re-running this job in turn will not use the manual variable specification again and will result in an attempt to publish all the charts under the `charts/` folder.

If no new versions of charts are found (i.e. if the version of the chart that you are trying to publish is already listed in the SKAO Helm repository), none will be uploaded. All the changes will be listed at the end of the CI Pipeline job.

Note: A chart has a `version` number and an `appVersion`. Updating only the appVersion number will *not* result in an update to the chart repository - if you want a new version of the application to be uploaded, you *must* update the chart version as well. Read more on the Helm documentation.

Working with a Helm Repository

Working with a Helm chart repository is well-documented on The Official Helm Chart Repository Guide.

Using the GitLab Registry for Helm Charts

Helm now has experimental (February, 2021) support for using OCI Registries as a Helm Chart Repository. This makes it possible to use GitLab as an intermediate store within CI/CD pipelines. The basic steps are:

- enable OCI Registry
- activate GPG support
- login to registry
- save chart (package)
- push chart to registry

Example:

```
helm publish to gitlb registry:
  stage: build
  variables:
    - HELM_EXPERIMENTAL_OCI: 1
  tags:
    - docker-executor
  script:
    - curl https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 |␣
↪bash
    - echo "$CI_JOB_TOKEN $CI" | helm registry login -u $CI_JOB_USER $CI_REGISTRY
    - helm chart save charts/<chart>/ $CI_REGISTRY/<chart>:<semantic_version>
    - helm chart push $CI_REGISTRY/<chart>:<semantic_version>
```

Adding the SKAO repository

The Helm Chart index is here https://artefact.skatelescope.org/#browse/search/helm . This consists of the hosted repository *helm-internal* and the upstream proxy of https://charts.helm.sh/stable.

In order to add the Helm chart repo to your local list of repositories, run

```
$ helm repo add skao https://nexus.engageska-portugal.pt/repository/helm-chart
```

Search available charts in a repo

To browse through the repo to find the available charts, you can then say (if, for example, you decided to name the repo `skatelescope`), to see output similar to this:

```
$ helm search skatelescope
NAME                          CHART VERSION   APP VERSION     DESCRIPTION
skatelescope/sdp-prototype    0.2.1           1.0             helm chart to deploy␣
↪the SDP Prototype on Kubernetes
skatelescope/test-app         0.1.0           1.0             A Helm chart for␣
↪Kubernetes
skatelescope/webjive          0.1.0           1.0             A Helm chart for␣
↪deploying the WebJive on Kubernetes
```

To install the test-app, you call **helm install the-app-i-want-to-test skatelescope/test-app** to install it in the default namespace. Test this with **kubectl get pods -n default**.

Update the repo

Almost like a **git fetch** command, you can update your local repositories' indexes by running

```
$ helm repo update
```

Note: this will update *ALL* your local repositories' index files.

PyPi and Python Packaging

Creating a Version

A developer should make use of the git annotated tags to indicate that this current commit is to serve as a release. For example:

```
$ git tag -a "1.0.0" -m "Release 1.0.0. This is a patch release that resolves
  issue <JIRA issue>."
```

After that is complete, then the tag needs to be published to the origin:

```
$ git push origin <tag_name>
```

Caution: The format of the tag must observe semantic versioning eg: N.N.N

Minimum Metadata requirements

For proper Python packaging, the following metadata must be present in the repository:

- Package name
- Package version
- Gitlab repo url
- Description of the package
- Classifiers

All of this should be specified in the *setup.py* module that lives in the project root directory, or the *project.toml* file if *poetry* is used for the build.

Additional metadata files that should be included in the root directory, are:

- README.{md|rst} - A description of the package including installation steps
- CHANGELOG.{md|rst} - A log of release versions and the changes in each version
- LICENSE - A text file with the relevant license

Building Python Packages

The following command will be executed in order to build a wheel for a Python package:

```
$ python setup.py sdist bdist_wheel
```

This will form part of the CI pipeline job for the repository so that it can be build automatically. The developer should add this build step in their *.gitlab-ci.yml* file, for example:

```
build_wheel for publication: # Executed on a tag:
  stage: build
  tags:
    - docker-executor
  script:
    - pip install setuptools
    - python setup.py egg_info -b+$CI_COMMIT_SHORT_SHA sdist bdist_wheel # --
↪universal option to be used for pure python packages
```

This will build a *Python* wheel that can then be published to the Central Artefact Repository. For developmental purposes one can replace the `-b+$CI_COMMIT_SHORT_SHA` command line option with `-b+dev.$CI_COMMIT_SHORT_SHA` to have the wheel built on each commit.

Publishing packages to *Nexus*

Provided that the release branch has been tagged precisely as described in the above sections then the CI job will be triggered by the availability of the tag to publish the *Python* wheel to the *SKAO* pypi registry on *Nexus*.

Publishing using `setup.py`:

```
# with setup.py
publish-python:
  stage: publish
  tags:
    - k8srunner
  variables:
    TWINE_USERNAME: $CAR_PYPI_USERNAME
    TWINE_PASSWORD: $CAR_PYPI_PASSWORD
  script:
    - pip3 install twine
    - twine upload --repository-url $CAR_PYPI_REPOSITORY_URL dist/*
  only:
    variables:
       - $CI_COMMIT_MESSAGE =~ /^.+$/ # Confirm tag message exists
       - $CI_COMMIT_TAG =~ /^((([0-9]+)\.([0-9]+)\.([0-9]+)(?:-([0-9a-zA-Z-]+(?:\.[0-
↪9a-zA-Z-]+)*))?)(?:\+([0-9a-zA-Z-]+(?:\.[0-9a-zA-Z-]+)*))?)$/ # Confirm semantic␣
↪versioning of tag
```

Publishing using `poetry`:

```
# with poetry and project.toml
publish-python:
  stage: publish
  tags:
    - k8srunner
  variables:
    POETRY_HTTP_BASIC_PYPI_USERNAME: $CAR_PYPI_USERNAME
    POETRY_HTTP_BASIC_PYPI_PASSWORD: $CAR_PYPI_PASSWORD
  before_script:
    - pip install poetry
    - poetry config virtualenvs.create false
```

(continues on next page)

(continued from previous page)

```
    - poetry install --no-root
    - poetry config repositories.skao $CAR_PYPI_REPOSITORY_URL
  script:
    - poetry build
    - poetry publish -r skao
  when: on_success
  only:
    refs:
      - tags
    variables:
      # Confirm tag message exists
      - $CI_COMMIT_MESSAGE =~ /^.+$/
      # Confirm semantic versioning of tag
      - $CI_COMMIT_TAG =~ /^((([0-9]+)\.([0-9]+)\.([0-9]+)(?:-([0-9a-zA-Z-]+(?:\.[0-
↪9a-zA-Z-]+)*))?)(?:\+([0-9a-zA-Z-]+(?:\.[0-9a-zA-Z-]+)*))?)$/
```

Publishing to the GitLab Project PyPi package repository:

```
# with poetry and project.toml
publish-python-gitlab:
  stage: build
  tags:
    - k8srunner
  variables:
    POETRY_HTTP_BASIC_PYPI_USERNAME: gitlab-ci-token
    POETRY_HTTP_BASIC_PYPI_PASSWORD: $CI_JOB_TOKEN
  before_script:
    - pip install poetry
    - poetry config virtualenvs.create false
    - poetry install --no-root
    - poetry config repositories.gitlab https://gitlab.com/api/v4/projects/${CI_
↪PROJECT_ID}/packages/pypi
  script:
    - poetry build
    - poetry publish -r gitlab
```

Installing a package from *Nexus*

The Python Package Index is located at https://artefact.skatelescope.org/#browse/search/pypi . A combined PyPi index of pypi-internal and pypi.python.org is available from https://artefact.skatelescope.org/repository/pypi-all/ .

Packages for upload must follow the SKAO naming convention starting with ska- (ADR-25) and incorporating the semantic version number. The following example shows the Python ska_logging class.

For developers who want to install a python package from the *SKAO* pypi registry hosted on *Nexus*, they should edit the project's Pipfile to have the following section(s), for example:

```
[[source]]
url = 'https://artefact.skatelescope.org/#browse/search/pypi'
verify_ssl = true
name = 'skao'

[packages]
'skaskeleton' = {version='*', index='skao'}
```

Installing a package from *GitLab*

The Python Package Index is located at `` `https://__token__:${CI_JOB_TOKEN}@gitlab.com/api/v4/projects/${CI_PROJECT_ID}/packages/pypi/simple` ``. This can be configured in the `` `~/.pypirc` `` files as follows within the CI/CD pipeline:

```
[distutils]
index-servers = gitlab

[gitlab]
repository = https://gitlab.example.com/api/v4/projects/${env.CI_PROJECT_ID}/packages/
↪pypi
username = gitlab-ci-token
password = ${env.CI_JOB_TOKEN}
...
```

Ansible Roles and Collections

Ansible roles and collections are held in a Raw format repository *helm-internal* . These are uploaded as individual files following the ADR-25 conventions of *<repository>/<role/collection name>* .

The following example is for common systems role collections:

```
curl -u ${CAR_ANSIBLE_USERNAME}:${CAR_ANSIBLE_PASSWORD} \
  --upload-file ska_cicd_docker_base--0.4.0.tar.gz \
  ${CAR_ANSIBLE_REPOSITORY_URL}/ska-cicd-roles/ska_cicd_docker_base--0.4.0.tar.gz
```

Raw

Raw artefacts are typically *tar.gz* files, images, reports, data files, and specific repositories that do not have direct functional support in Nexus (same as for Ansible roles and collections). These are hosted here raw-internal . Note that the artefact directory structure must be prefixed by the related repository, but can be flexible but meaningful after that.

The following example shows the publishing of an external dependency library for the Tango base image builds:

```
curl -u ${CAR_RAW_USERNAME}:${CAR_RAW_PASSWORD} \
  --upload-file tango-9.3.4.tar.gz \
  ${CAR_RAW_REPOSITORY_URL}/ska-tango-images/libraries/tango-9.3.4.tar.gz
```

GitLab Generic Package Repository

The GitLab Generic Repository can be used to store arbitrary artefacts between job steps (as an alternative to the gitlab runner cache) or between pipelines.

Upload artefacts in CI/CD with:

```
upload:
  stage: upload
  script:
    - 'curl --header "JOB-TOKEN: $CI_JOB_TOKEN" --upload-file path/to/really_
↪important.tar.gz "${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/packages/generic/$
↪{YOUR_PACKAGE_NAME}/${SEMANTIC_VERSION}/really_important.tar.gz"'
```

These can later be retrieved with:

```
download:
  stage: download
  script:
    - 'wget --header="JOB-TOKEN: $CI_JOB_TOKEN" ${CI_API_V4_URL}/projects/${CI_
↪PROJECT_ID}/packages/generic/${YOUR_PACKAGE_NAME}/${SEMANTIC_VERSION}/really_
↪important.tar.gz'
```

2.14 SKA Log Message Format

2.14.1 Logging Levels

The log format should correspond to the default Python logging levels.

Typically, a Python log message consists of two bits of information from the perspective of calling *log* function in code:

1. Log level - corresponds to the Python standard logging levels
2. Message - a UTF-8 encoded string

The logging levels used can be any one of the five standard Python logging levels. For other language runtimes, the appropriate level that corresponds to the RFC5424 standard should be used.

Note: The Syslog RFC is used as a reference format for mapping log levels only. The *Log Message Standard* does not conform to RFC5424 and is not an extension of syslog. For reasoning behind this, see *Design Motivations* section.

Mapping of Logging Levels

The table below maps Python logging levels to that of RFC5424 (syslog).

```
(This table is inside a block quote because when HTML is viewed in browser,
the table content gets replaced by a list of repositories.)

======== ============= ====================== ======================
Python   RFC5424       RFC5424 Numerical Code Your language runtime
======== ============= ====================== ======================
DEBUG    Debug         7                      ?
INFO     Informational 6                      ?
WARNING  Warning       4                      ?
ERROR    Error         3                      ?
CRITICAL Critical      2                      ?
======== ============= ====================== ======================
```

For guidelines on when to use a particular log level, please refer to the official Python logging HOWTO.

2.14.2 Log Message Standard

All processes that execute inside containers **must** log to *stdout*.

In order for log messages to be ingested successfully into the logging system once deployed, the log message should conform to the following format (in ABNF):

```
SKA-LOGMSG = VERSION "|" TIMESTAMP "|" SEVERITY "|" [THREAD-ID] "|" [FUNCTION] "|"␣
↪[LINE-LOC] "|" [TAGS] "|" MESSAGE LF
VERSION    = 1*2DIGIT                                                   ;␣
↪(compulsory) version of SKA log standard this log message implements - starts at 1
TIMESTAMP  = FULL-DATE "T" FULL-TIME                                    ;␣
↪(compulsory) ISO8601 compliant timestamp normalised to UTC
THREAD-ID  = *32("-" / ALPHA / DIGIT)                                   ; (optional)␣
↪thread id, e.g. "MainThread" or "Thread-1"
FUNCTION   = *(NAMESPACE ".") *ALPHA                                    ; (optional)␣
↪full namespace of function, e.g. package.module.TangoDevice.method
LINE-LOC   = FILENAME "#" LINENO *" "                                   ; (optional)␣
↪file and line where log was called
SEVERITY   = ("DEBUG" / "INFO" / "WARNING" / "ERROR" / "CRITICAL") *" " ;␣
↪(compulsory) log level/severity
TAGS       = TAG *[ "," TAG ]                                           ; (optional)␣
↪comma-separated list of tags e.g. facility:MID,receptor:m043
MESSAGE    = *OCTET                                                     ; message␣
↪content (UTF-8 string) (should we think about constraining length?)
FILENAME   = 1*64 (ALPHA / "." / "_" / "-" / DIGIT)                     ; from 1 up␣
↪to 64 characters
LINENO     = 1*5DIGIT                                                   ; up to 5␣
↪digits (hopefully no file has more than 99,999 loc)
TAG        = *(ALPHA / "-") ":" *VCHAR                                  ; name-value␣
↪pairs
FULL-DATE  = 4DIGIT "-" 2DIGIT "-" 2DIGIT                               ; e.g. 2019-
↪12-31
FULL-TIME  = 2DIGIT ":" 2DIGIT ":" 2DIGIT "." 3*6DIGIT "Z"              ; 23:42:50.
↪523Z = 42 minutes and 50.523 seconds after the 23rd hour in UTC. Minimum subsecond␣
↪precision should be 3 decimal points.
OCTET      = %d00-255                                                   ; any byte
DIGIT      = %d48-57                                                    ; 0 - 9
```

Examples:

```
1|2019-12-31T23:12:37.526Z|INFO||testpackage.testmodule.TestDevice.test_fn|test.py
↪#1|tango-device:my/dev/name| Regular information should be logged like this FYI
1|2019-12-31T23:45:42.328Z|DEBUG||testpackage.testmodule.TestDevice.test_fn|test.py
↪#150|| x = 67, y = 24
1|2019-12-31T23:49:53.543Z|WARNING||testpackage.testmodule.TestDevice.test_fn|test.py
↪#16|| z is unspecified, defaulting to 0!
1|2019-12-31T23:50:17.124Z|ERROR||testpackage.testmodule.TestDevice.test_fn|test.py
↪#165|site:Element| Could not connect to database!
1|2019-12-31T23:51:23.036Z|CRITICAL||testpackage.testmodule.TestDevice.test_fn|test.py
↪#16|| Invalid operation. Cannot continue.
```

Versioning

The log standard is versioned so that it can be modified or extended. Theoretically anything after the first "|" delimiter can be changed as long as you specify a different version number up to 99.

For example, if it's decided that LINE-LOC is no longer needed as a first class field in the log standard, we can publish a new version omitting it.

Version 1:

```
1|2019-12-31T23:49:13.543Z|WARNING||testpackage.testmodule.TestDevice.test_fn|test.py
↪#16|| z is unspecified, defaulting to 0!
```

Version 2:

```
2|2019-12-31T23:49:13.543Z|WARNING||test.py#16|| z is unspecified, defaulting to 0!
```

Parsing

The format is simple enough and the only fixed points is the choice of delimiter ("|") and number of first class fields (8, as of version 1). This allows for two basic parsing strategies.

Procedural

Splitting by delimiter and then refering to the index is a common operation in any programming environment.

Python (str.split)

```
log_line = "1|2019-12-31T23:50:17.124Z|ERROR||testpackage.testmodule.TestDevice.test_
↪fn|test.py#165|site:Element| Could not connect to database!"
structured_log = log_line.split('|')
log_level = structured_log[5]
```

Regex

The following regular expression can match all fields between the "|" delimiters:

[^|]+(?=|[^|]*$) A more specific regex that leverages named capture to extract matches:

```
^(?<version>\d+)[|](?<timestamp>[0-9TZ\-:.]+)[|](?<level>[\w\s]+)[|](?<thread>[\w-
↪]*)[|](?<function>[\w\-.]*)[|](?<lineloc>[\w\s.#]*)[|](?<tags>[\w\:,-]*)[|](?
↪<message>.*)$
```

For a demonstration see: https://rubular.com/r/e0njVOGCN59mtA

2.14.3 Design Motivations

The design of the log format above is a work in progress and a first attempt to introduce standardised logging practices. Some preliminary investigations were made to survey the current logging practices employed in different teams/components (see a report on this, Investigation of Logging Practices).

Assumption 1: First-party components to be integrated on a system level will be containerised.

Implication: Containerisation best practices with regards to logging should apply. This means logging to *stdout* or console so that the routing and handling of log messages can be handled by the container runtime (*dockerd*, *containerd*) or dynamic infrastructure platform (k8s).

Assumption 2: A log ingestor component will be deployed as part of logging architecture.

Implication: A log ingestor is responsible for:

- fetching log data from a source, e.g. journald, file , socket, etc.
- processing it, e.g. parsing based on standardised format to extract key information and transform to other formats such as JSON to be sent to a log datastore.
- shipping it to a log datastore (Elasticsearch) or another log ingestor (Logstash)

Syslog (RFC5424)

We question the need for conforming to syslog standard in container level logs that print to *stdout*. From prior investigations, the existing log practices in the SKA codebase do not necessarily conform to syslog either, nor is there a consistent pattern. We used this opportunity to propose a log format the meets the folllowing goals:

As such we believe the most important features of a standard log message are:

1. to prescribe minimum supported bits of useful information, this includes
   a. timestamp
   b. log level
   c. extensible tags - a mechanism to specify arbitrary tags [1]
   d. fully qualified name of call context (the function in source code that log comes from) [1]
   e. filename where log call is situated [1]
   f. line number in file [1]
2. should be easy to parse
3. readability for local development

Log messages that conform to a standard can always be transformed into syslog compliant loglines before being shipped to a log aggregator.

Time stamps

Timestamps are included as part of the standard log message so that we can troubleshoot a class of issues that might occur between processes and the ingestion of logs, .e.g. reconcile order of log messages between ingestor and process.

Tags

To avoid upfront assumptions about what identifiers are universally required, we specify a section for adding arbitrary tags. We can standardise on some tag names later on, e.g. `TangoDeviceName:powersupply,Tango`

2.14.4 Further work

Log Ingestor Transformations

Implementation details of how log transformations ought to work, will be architecture specific but we still need to understand how to achieve it in the chosen technology (whether fluentd or filebeat+logstash).

This implies deploying a log ingestor close as possible to the target container/process and have it transform log messages according to the above spec before shipping it to log storage (elasticsearch).

Field size limits

Decide on reasonable size limits for each field, e.g. SEVERITY will always be between 4-8 characters: INFO(4), CRITICAL(8)

Should MESSAGE have a size limit? What if we want to add an arbitrary data structure inside the MESSAGE such as a JSON object? Should it support that or be disallowed upfront?

2.14.5 Standard Tags (LogViewer)

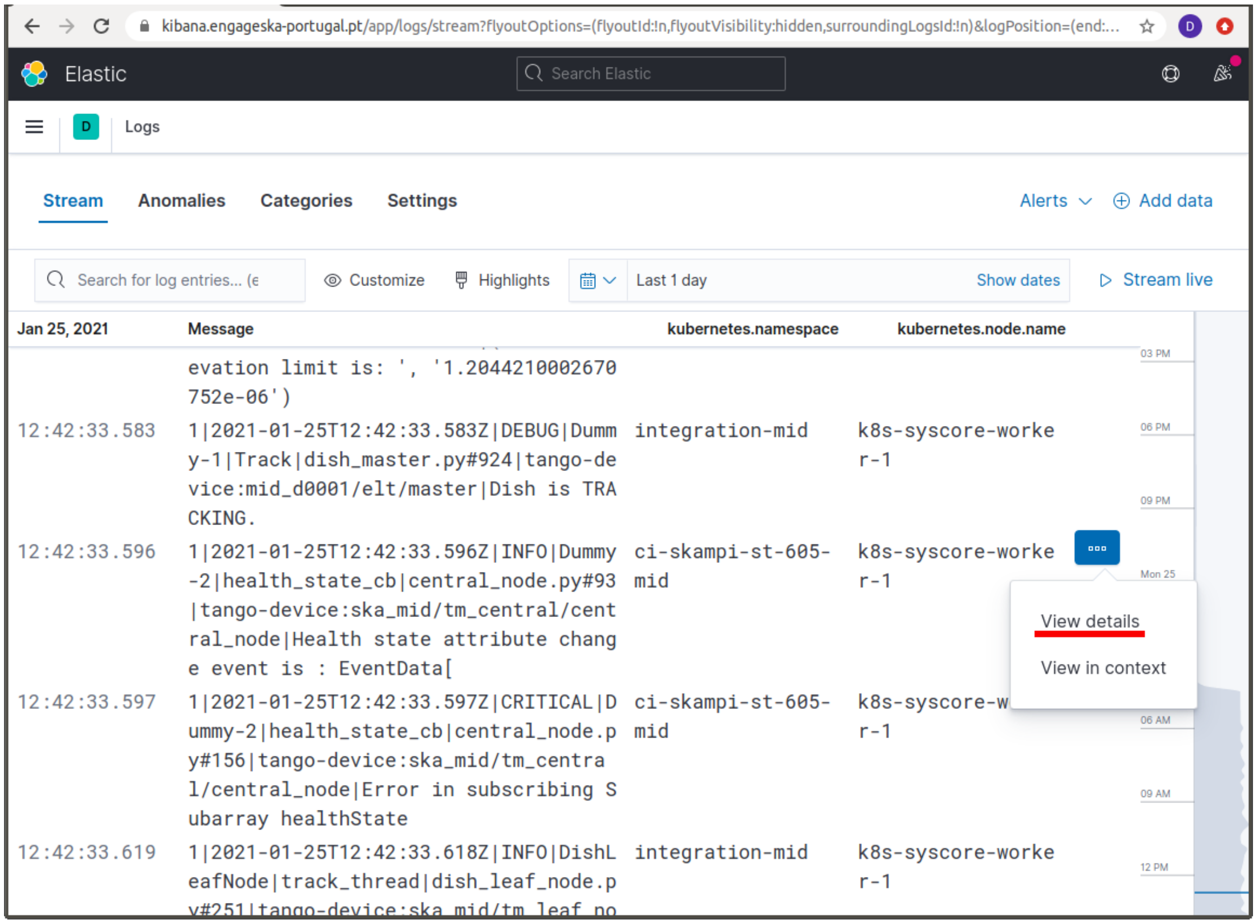
A list of tags (identifiers) we want to add to log messages for easy filtering and semantic clarity:

- Tag: deviceName
  - Description: Identifier that corresponds to the TANGO device name, a string in the form: "<facility>/<family>/<device>".
    - facility : The TANGO facility encodes the telescope (LOW/MID) and its sub-system [2] (see [3]),
    - family : Family within facility (see [3]),
    - device : TANGO device name (see [3]).
  - Example: `MID-D0125/rx/controller`, where
    - `MID-D0125` : Dish serial number,
    - `rx` : Dish Single Pixel Feed Receiver (SPFRx),
    - `controller` : Dish SPFRx controller.
- Tag: subSystem
  - Description: For software that are not TANGO devices, the name of the telescope sub-system [2].
  - Example: `SDP`

2.14.6 Filtering the Logs on Kibana

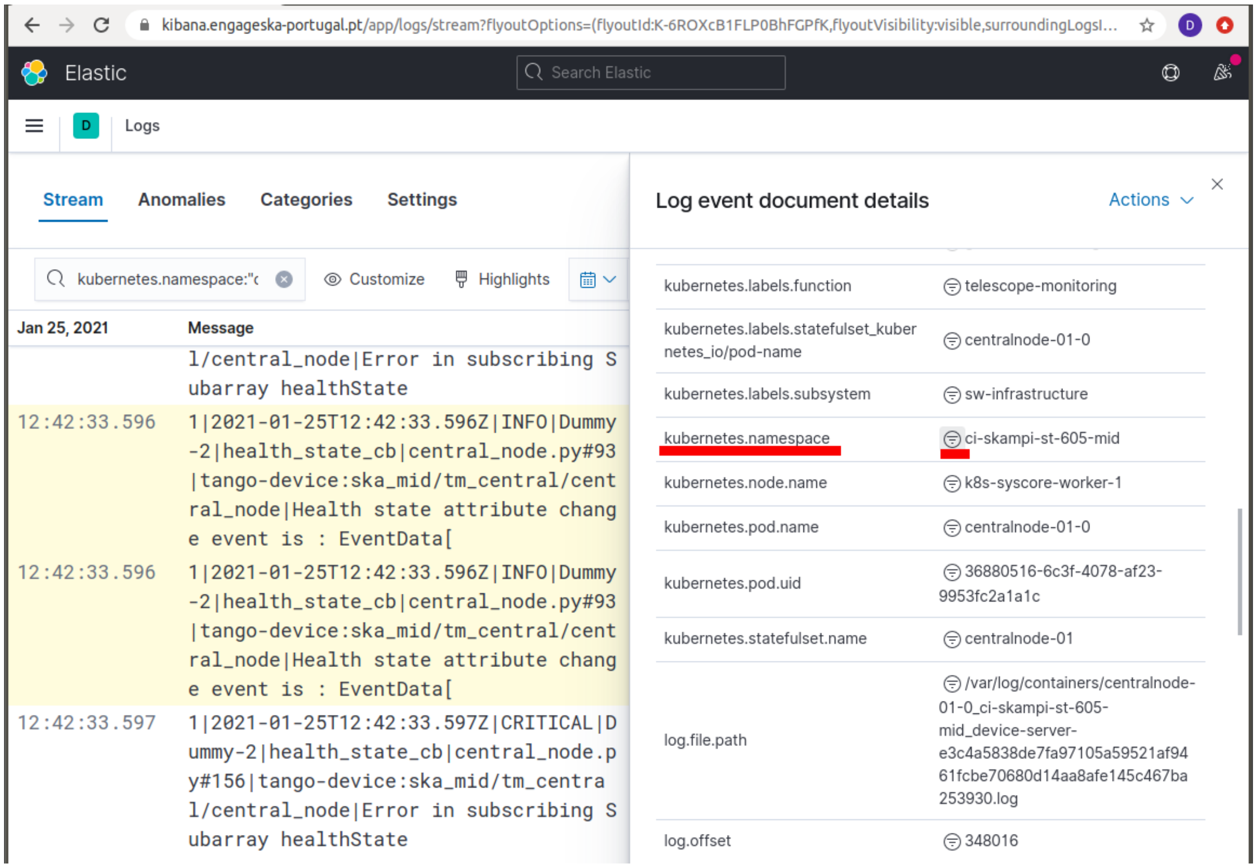
Log messages from the core syscore cluster can be checked in our monitoring platform at https://kibana.engageska-portugal.pt/app/logs

Kibana allows for filtering on log messages on the basis of a series of fields. These fields can be added as columns to display information using the **Settings** option, and filtering based on the values of those fields can be done directly on the **Search** box or by selecting the **View details** menu:


kibana.engageska-portugal.pt/app/logs/stream?flyoutOptions=(flyoutId:!n,flyoutVisibility:hidden,surroundingLogsId:!n)&logPosition=(end:...
Elastic
Search Elastic
Logs
Stream
Anomalies
Categories
Settings
Alerts
Add data
Search for log entries... (e
Customize
Highlights
Last 1 day
Show dates
Stream live
View details
View in context

```
Jan 25, 2021  Message                              kubernetes.namespace   kubernetes.node.name
              evation limit is: ', '1.2044210002670
              752e-06')
12:42:33.583  1|2021-01-25T12:42:33.583Z|DEBUG|Dumm  integration-mid   k8s-syscore-worke
              y-1|Track|dish_master.py#924|tango-de                    r-1
              vice:mid_d0001/elt/master|Dish is TRA
              CKING.
12:42:33.596  1|2021-01-25T12:42:33.596Z|INFO|Dummy  ci-skampi-st-605-  k8s-syscore-worke
              -2|health_state_cb|central_node.py#93  mid               r-1
              |tango-device:ska_mid/tm_central/cent
              ral_node|Health state attribute chang
              e event is : EventData[
12:42:33.597  1|2021-01-25T12:42:33.597Z|CRITICAL|D  ci-skampi-st-605-  k8s-syscore-w
              ummy-2|health_state_cb|central_node.p  mid               r-1
              y#156|tango-device:ska_mid/tm_centra
              l/central_node|Error in subscribing S
              ubarray healthState
12:42:33.619  1|2021-01-25T12:42:33.618Z|INFO|DishL  integration-mid   k8s-syscore-worke
              eafNode|track_thread|dish_leaf_node.p                    r-1
              y#251|tango-device:ska_mid/tm_leaf_no
```

In the example above in order to retrieve only the log messages relevant for the skampi development pipeline `ci-skampi-st-605-mid` one should then select the corresponding `kubernetes.namespace` field value.

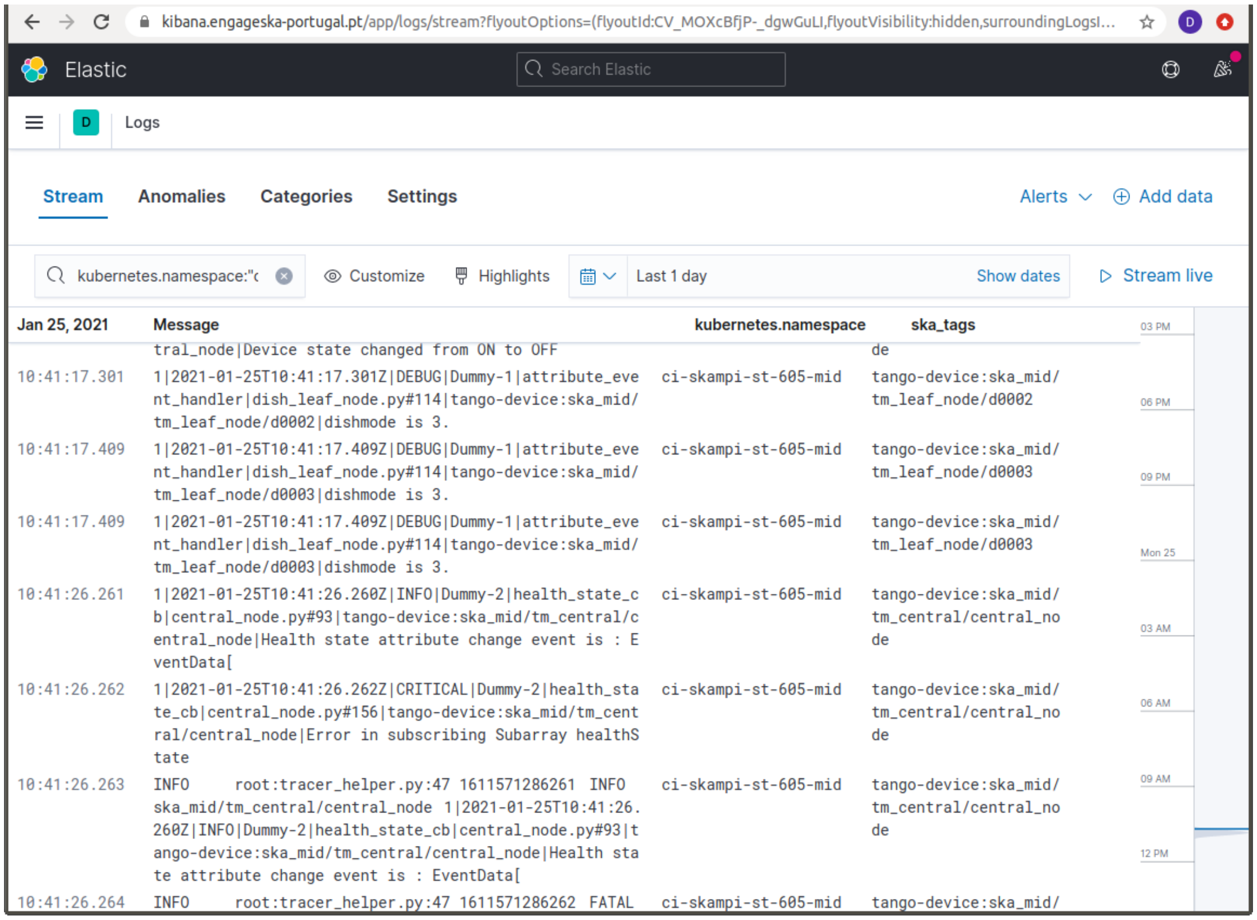

kibana.engageska-portugal.pt/app/logs/stream?flyoutOptions=(flyoutId:K-6ROXcB1FLP0BhFGPfK,flyoutVisibility:visible,surroundingLogsI...
Elastic
Search Elastic
Logs
Stream
Anomalies
Categories
Settings
kubernetes.namespace:"c
Customize
Highlights
Log event document details
Actions

```
Jan 25, 2021    Message
                l/central_node|Error in subscribing S
                ubarray healthState
12:42:33.596    1|2021-01-25T12:42:33.596Z|INFO|Dummy
                -2|health_state_cb|central_node.py#93
                |tango-device:ska_mid/tm_central/cent
                ral_node|Health state attribute chang
                e event is : EventData[
12:42:33.596    1|2021-01-25T12:42:33.596Z|INFO|Dummy
                -2|health_state_cb|central_node.py#93
                |tango-device:ska_mid/tm_central/cent
                ral_node|Health state attribute chang
                e event is : EventData[
12:42:33.597    1|2021-01-25T12:42:33.597Z|CRITICAL|D
                ummy-2|health_state_cb|central_node.p
                y#156|tango-device:ska_mid/tm_centra
                l/central_node|Error in subscribing S
                ubarray healthState
```

kubernetes.labels.function	telescope-monitoring
kubernetes.labels.statefulset_kubernetes_io/pod-name	centralnode-01-0
kubernetes.labels.subsystem	sw-infrastructure
kubernetes.namespace	ci-skampi-st-605-mid
kubernetes.node.name	k8s-syscore-worker-1
kubernetes.pod.name	centralnode-01-0
kubernetes.pod.uid	36880516-6c3f-4078-af23-9953fc2a1a1c
kubernetes.statefulset.name	centralnode-01
log.file.path	/var/log/containers/centralnode-01-0_ci-skampi-st-605-mid_device-server-e3c4a5838de7fa97105a59521af9461fcbe70680d14aa8afe145c467ba253930.log
log.offset	348016

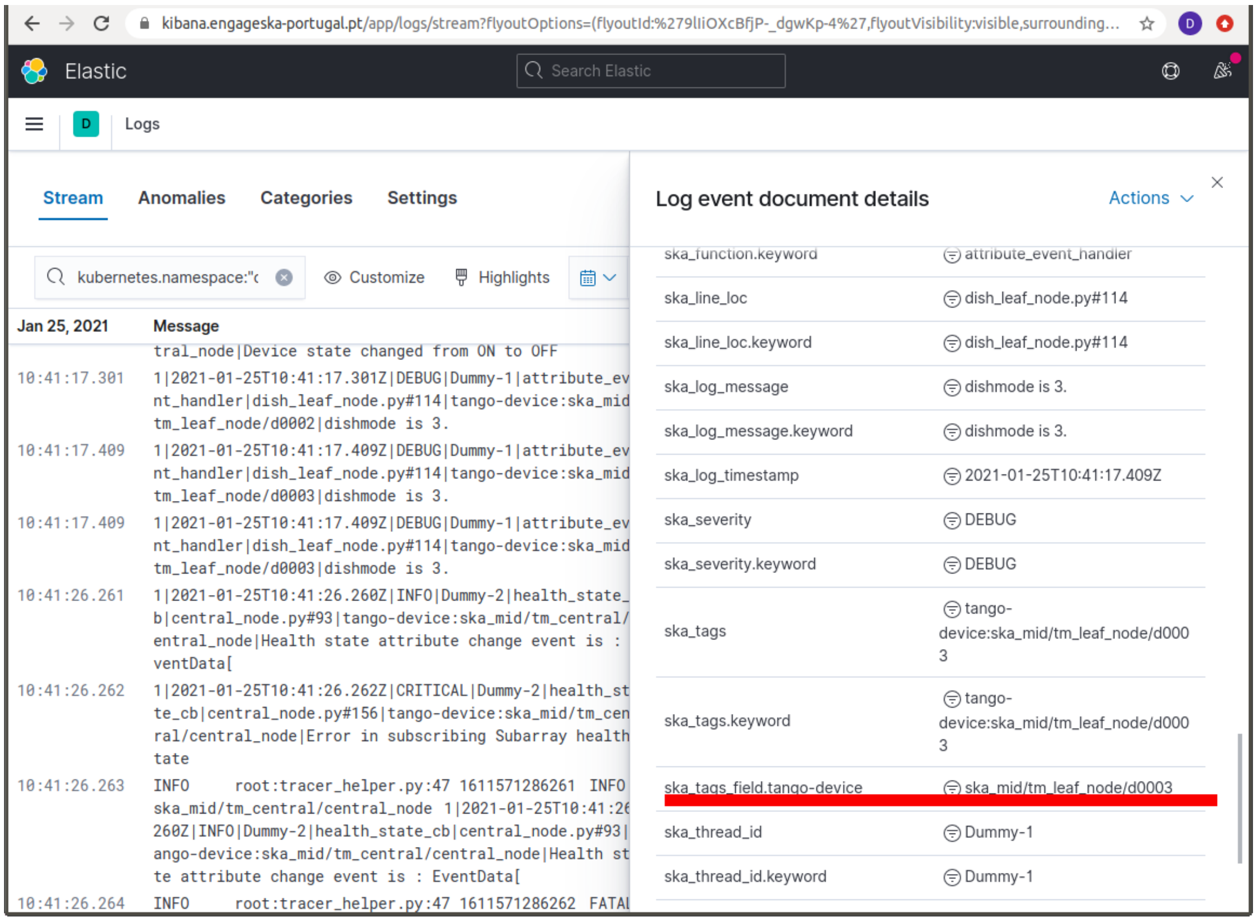
There are many other field options using kubernetes information, for example `kubernetes.node.name` and `kubernetes.pod.name` that can be used for efficient filtering.

The fact the SKA logging format allows for simple key-value pairs (SKA Tags) to be included in log messages let us refine the filtering. Tags are parsed to a field named `ska_tags` and on this field there can be one or more device properties separated by commas.



```
Jan 25, 2021  Message                                            kubernetes.namespace  ska_tags
              tral_node|Device state changed from ON to OFF                            de
10:41:17.301  1|2021-01-25T10:41:17.301Z|DEBUG|Dummy-1|attribute_eve  ci-skampi-st-605-mid  tango-device:ska_mid/
              nt_handler|dish_leaf_node.py#114|tango-device:ska_mid/                           tm_leaf_node/d0002
              tm_leaf_node/d0002|dishmode is 3.
10:41:17.409  1|2021-01-25T10:41:17.409Z|DEBUG|Dummy-1|attribute_eve  ci-skampi-st-605-mid  tango-device:ska_mid/
              nt_handler|dish_leaf_node.py#114|tango-device:ska_mid/                           tm_leaf_node/d0003
              tm_leaf_node/d0003|dishmode is 3.
10:41:17.409  1|2021-01-25T10:41:17.409Z|DEBUG|Dummy-1|attribute_eve  ci-skampi-st-605-mid  tango-device:ska_mid/
              nt_handler|dish_leaf_node.py#114|tango-device:ska_mid/                           tm_leaf_node/d0003
              tm_leaf_node/d0003|dishmode is 3.
10:41:26.261  1|2021-01-25T10:41:26.260Z|INFO|Dummy-2|health_state_c  ci-skampi-st-605-mid  tango-device:ska_mid/
              b|central_node.py#93|tango-device:ska_mid/tm_central/c                           tm_central/central_no
              entral_node|Health state attribute change event is : E                           de
              ventData[
10:41:26.262  1|2021-01-25T10:41:26.262Z|CRITICAL|Dummy-2|health_sta  ci-skampi-st-605-mid  tango-device:ska_mid/
              te_cb|central_node.py#156|tango-device:ska_mid/tm_cent                           tm_central/central_no
              ral/central_node|Error in subscribing Subarray healthS                           de
              tate
10:41:26.263  INFO     root:tracer_helper.py:47 1611571286261  INFO   ci-skampi-st-605-mid  tango-device:ska_mid/
              ska_mid/tm_central/central_node 1|2021-01-25T10:41:26.                           tm_central/central_no
              260Z|INFO|Dummy-2|health_state_cb|central_node.py#93|t                           de
              ango-device:ska_mid/tm_central/central_node|Health sta
              te attribute change event is : EventData[
10:41:26.264  INFO     root:tracer_helper.py:47 1611571286262  FATAL  ci-skampi-st-605-mid  tango-device:ska_mid/
```

The field `ska_tags` is also parsed so that the key is added to a `ska_tags_field` prefix that will store the value. For the example above, this means filtering the messages using the value of the `ska_tags_field.tango-device` field.


kibana.engageska-portugal.pt/app/logs/stream?flyoutOptions=(flyoutId:%279lIiOXcBfjP-_dgwKp-4%27,flyoutVisibility:visible,surrounding...
Elastic
Search Elastic
Logs
Stream
Anomalies
Categories
Settings
kubernetes.namespace:"c
Customize
Highlights
Log event document details
Actions

```
Jan 25, 2021  Message
              tral_node|Device state changed from ON to OFF
10:41:17.301  1|2021-01-25T10:41:17.301Z|DEBUG|Dummy-1|attribute_ev
              nt_handler|dish_leaf_node.py#114|tango-device:ska_mid
              tm_leaf_node/d0002|dishmode is 3.
10:41:17.409  1|2021-01-25T10:41:17.409Z|DEBUG|Dummy-1|attribute_ev
              nt_handler|dish_leaf_node.py#114|tango-device:ska_mid
              tm_leaf_node/d0003|dishmode is 3.
10:41:17.409  1|2021-01-25T10:41:17.409Z|DEBUG|Dummy-1|attribute_ev
              nt_handler|dish_leaf_node.py#114|tango-device:ska_mid
              tm_leaf_node/d0003|dishmode is 3.
10:41:26.261  1|2021-01-25T10:41:26.260Z|INFO|Dummy-2|health_state_
              b|central_node.py#93|tango-device:ska_mid/tm_central/
              entral_node|Health state attribute change event is :
              ventData[
10:41:26.262  1|2021-01-25T10:41:26.262Z|CRITICAL|Dummy-2|health_st
              te_cb|central_node.py#156|tango-device:ska_mid/tm_cen
              ral/central_node|Error in subscribing Subarray health
              tate
10:41:26.263  INFO     root:tracer_helper.py:47 1611571286261 INFO
              ska_mid/tm_central/central_node 1|2021-01-25T10:41:26
              260Z|INFO|Dummy-2|health_state_cb|central_node.py#93|
              ango-device:ska_mid/tm_central/central_node|Health st
              te attribute change event is : EventData[
10:41:26.264  INFO     root:tracer_helper.py:47 1611571286262 FATAL
```

ska_function.keyword	attribute_event_handler
ska_line_loc	dish_leaf_node.py#114
ska_line_loc.keyword	dish_leaf_node.py#114
ska_log_message	dishmode is 3.
ska_log_message.keyword	dishmode is 3.
ska_log_timestamp	2021-01-25T10:41:17.409Z
ska_severity	DEBUG
ska_severity.keyword	DEBUG
ska_tags	tango-device:ska_mid/tm_leaf_node/d0003
ska_tags.keyword	tango-device:ska_mid/tm_leaf_node/d0003
ska_tags_field.tango-device	ska_mid/tm_leaf_node/d0003
ska_thread_id	Dummy-1
ska_thread_id.keyword	Dummy-1

Making the selection illustrated above means that only messages with the value `ska_mid/tm_leaf_node/d0003` for the `ska_tags_field.tango-device` field would be displayed.

[1] Optional, since it won't apply to all contexts, e.g. third-party applications.

[2] CSP, Dish, INAU, INSA, LFAA, SDP, SaDT, TM.

[3] 000-000000-012, SKA1 TANGO Naming Convention (CS_GUIDELINES Volume2), Rev 01

2.15 Reporting Bugs

Coming soon.

2.16 Coding guidelines

2.16.1 C++ Coding Guidelines

This section describes requirements and guidelines for development and testing of a new C++ project on GitLab. The Continuous Integration tools are GitLab specific - but most of the processes could be easily moved to a Jenkins Pipeline if required.

Table of Contents

An Example C++ Project

We have created a skeleton C++ project. Which should provide a full introduction to the various recommendations and requirements for the development of C++. The philosophy behind the development of this template was to demonstrate one way to meet the project guidelines. There are probably as many ways to organise C++ projects as there are developers there are sure to be some controversial design decisions made.

This projects demonstrates a recommended project layout. It also demonstrates how to implement the following recommended C++ project development features.

- Continuous integration (CI) setup using Gitlab
- CMake as a build tool.
- GoogleTest framework and example unit testing.
- C++ linting using clang for stylistic errors.
- Also test running under valgrind for memory errors.
- gcov to measure test coverage

All building and testing is done within a docker container.

Project Layout

Top Level

We have a top level of the project containing:

CMakeLists.txt

We have built this template using CMake. The structure of this file will be discussed in *Building the Project*.

LICENSE

All projects *must* have a license. We have include the recommended BSD 3 clause license template *please fill it in*.

README.md

Should at the very least provide a brief description, installation instructions, pre-requisites

src

This is the top of *the source tree*. It does not have to be called src. But we recommend that the source tree has its head here. The first namespace being the level below this one.

version.txt

In this template the version file is parsed by the CMakeLists file during the build process. This specific functionality is not required, but clear versioning is. The SKA project mandates Semantic Versioning.

The Source Tree

Our recommended source tree structure follows:

```
.
├── CMakeLists.txt
├── LICENSE
├── README.md
├── cmake
│   └── HelloWorldConfig.cmake.in
├── dependencies.txt
├── examples
│   └── helloworld-user
├── external
│   ├── cmake-modules
│   └── gtest-1.8.1
├── src
│   ├── apps
│   └── ska
└── version.txt

9 directories, 6 files
```

A number of design decisions have been made here:

What To Do With Namespaces

The directory structure follows the namespaces. We have defined an uppermost *ska* namespace and require all projects providing specfic ska functionality do the same. We have also defined a *nested* namespace and the directory structure follows in kind. This allows code to be grouped together easily by namespace. Also the installation assumes this structure. Therefore headers are included using their full namespaces. This avoids pollution of the install tree.

Headers

Header (.h) files are included with and template definitions (.tcc) and implementation files (.cpp). We have done this to make it easier to navigate the source tree. Having a separate include tree adds unnecessary complexity.

Design Patterns

We recommend that project follow design patterns where applicable. The example class structure follows the *Pimpl* construction design pattern but there are many others. This has an abstract base class *hello* and a derived class *wave* which is a type of hello, but we are also including an implementation of a *wave*. We have done this as this scheme allows multiple implementations of a wave to be created without forcing a recompilation of every source file that includes *wave.h*. Some example patterns for different use cases can be found in this article.

Unit Test Locations

Tests are co-located at the namespace level of the classes that they test. Another scheme would be to have a separate branch from the top level that maintains the same directory structure. We prefer this scheme for the same reasons as we prefer to colocate the headers and the implementation. Plus anything that promotes the writing of unit tests at the same time as the classes are developed is a good thing

Todo: What about functional tests

Continuous Integration

GitLab manages builds via a YAML file held inside your repository. Similar to a Jenkinsfile in philosophy. This is the file in the skeleton HelloWorld project.

```
# This file is a template, and might need editing before it works on your project.
# use the official gcc image, based on debian
# can use verions as well, like gcc:5.2
# see https://hub.docker.com/_/gcc/
#
# This base image is based on debian:buster-slim and contains:
#  * gcc 8.3.0
#  * clang 7.0.1
#  * cmake 3.13.4
#  * and more
#
# For details see https://github.com/ska-telescope/cpp_build_base
#
image: nexus.engageska-portugal.pt/ska-docker/cpp_build_base

variables:
  # Needed if you want automatic submodule checkout
  # For details see https://docs.gitlab.com/ee/ci/yaml/README.html#git-submodule-
↪strategy
  GIT_SUBMODULE_STRATEGY: normal

.common: {tags: [engageska, docker]}

stages:
  - build
  - linting
  - test
  - pages

build_debug:
  extends: .common
  stage: build
  # instead of calling g++ directly you can also use some build toolkit like make
  # install the necessary build tools when needed
  script:
    - mkdir build
    - cd build
    - cmake .. -DCMAKE_BUILD_TYPE=Debug -DCMAKE_CXX_FLAGS="-coverage" -DCMAKE_EXE_
↪LINKER_FLAGS="-coverage"
    - make
  artifacts:
    paths:
      - build

build_release:
  extends: .common
  stage: build
```

(continues on next page)

(continued from previous page)

```
  script:
    - mkdir build
    - cd build
    - cmake .. -DCMAKE_BUILD_TYPE=Release
    - make
  artifacts:
    paths:
      - build

build_export_compile_commands:
 extends: .common
 stage: build
 script:
    - rm -rf build && mkdir build
    - cd build
    - cmake .. -DCMAKE_BUILD_TYPE=Debug -DCMAKE_EXPORT_COMPILE_COMMANDS=ON -DCMAKE_
↪CXX_COMPILER=clang++
 artifacts:
    paths:
      - build

lint_clang_tidy:
  extends: .common
  stage: linting
  dependencies:
    - build_export_compile_commands
  script:
    - cd build
    - run-clang-tidy -checks='cppcoreguidelines-*,performance-*,readibility-*,
↪modernize-*,misc-*,clang-analyzer-*,google-*'

lint_iwyu:
  extends: .common
  stage: linting
  dependencies:
    - build_export_compile_commands
  script:
    - cd build
    - iwyu_tool -p .

lint_cppcheck:
  extends: .common
  stage: linting
  dependencies:
    - build_export_compile_commands
  script:
    - cd build
    - cppcheck --error-exitcode=1 --project=compile_commands.json -q --std=c++14 -i
↪$PWD/../external -i $PWD/../src/ska/helloworld/test

test:
  extends: .common
  stage: test
  dependencies:
    - build_debug
  script:
    - cd build
```

(continues on next page)

(continued from previous page)

```
    - ctest -T test --no-compress-output
  after_script:
    - cd build
    - ctest2junit > ctest.xml
  artifacts:
    paths:
      - build/
    reports:
      junit: build/ctest.xml

test_installation:
  extends: .common
  stage: test
  dependencies:
    - build_release
  script:
    - cd build
    - cmake . -DCMAKE_INSTALL_PREFIX=/opt
    - make install
    - cd ../examples/helloworld-user
    - mkdir build
    - cd build
    - cmake .. -DCMAKE_PREFIX_PATH=/opt
    - make all

# A job that runs the tests under valgrid
# It might take a while, so not always run by default
test_memcheck:
  extends: .common
  stage: test
  dependencies:
    - build_debug
  before_script:
    - apt update && apt install -y valgrind
  script:
    - cd build
    - ctest -T memcheck
  only:
    - tags
    - schedules

pages:
  extends: .common
  stage: pages
  dependencies:
    - test
  script:
    - mkdir .public
    - cd .public
    - gcovr -r ../ -e '.*/external/.*' -e '.*/CompilerIdCXX/.*' -e '.*/tests/.*' --
↪html --html-details -o index.html
    - cd ..
    - mv .public public
  artifacts:
    paths:
      - public
```

We have four stages of the CI

- build
- lint
- test
- pages

These stages are automatically run by the GitLab runners when you push to the repository. The pipeline halts and you are informed if any of the steps fail. There are some subtleties in the way the test results and test coverage are reported and we deal with them below as we go through the steps in more detail.

Building The Project

Todo: Need to add some CMake Coding Conventions

The Image

We recommend building using the cpp_base image that we have developed and stored in the image repository. This, or an image derived from it contains all the tools required to operate this CI pipeline and you should avoid the pain of installing and configuring system tools.

CMake for Beginners

We recommend using CMake as a build tool. It is widely used, includes the ability to generate buildfiles for different development environments. This allows developers the freedom to use whatever IDE they would like (e.g. Eclipse and Xcode) from the same CMake files.

The CMake application parses the CMakeLists.txt file and generates a set of build scripts. An important element of the CMake philosophy is that you can build the application out-of-place, thereby supporting multiple build configurations from the same source tree.

This is the top level CMakeLists.txt file

```
cmake_minimum_required(VERSION 3.5)

file(STRINGS version.txt HelloWorld_VERSION)
message(STATUS "Building HelloWorld version ${HelloWorld_VERSION}")

# Project configuration, specifying version, languages,
# and the C++ standard to use for the whole project
project(HelloWorld LANGUAGES CXX VERSION ${HelloWorld_VERSION})
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

include(CTest)

# External projects
if (BUILD_TESTING)
        add_subdirectory(external/gtest-1.8.1 googletest)
# installed as git submodule - if this is your first clone you need to
```

(continues on next page)

(continued from previous page)

```
# git submodule init
# git submodule update
# This is a cmake module and needs no further input from you
endif()

add_subdirectory(src)
```

This file defines the project, but the actual applications and libraries are found futher down the tree. An example CMake file in the template that is used to build the executeable is

```
# add the executable
add_executable (HelloWorld hello_world.cc)
target_link_libraries (HelloWorld HelloClasses)

install (TARGETS HelloWorld DESTINATION bin)

add_test(NAME HelloWorldExec COMMAND HelloWorld)
```

There are a number of beginner CMake tutorials on the web. The following commands should be all you need to build the HelloWorld project on your system.

```
>cd cpp_template
```

For an out-of-place build you make your build directory. This can be anywhere

```
> mkdir build
> cd build
```

Then you run *cmake* - you control where you want the installation to go by setting the install prefix. But you also need to point *cmake* to the directory containing your top level CMakeLists.txt file

```
> cmake -DCMAKE_INSTALL_PREFIX=my_install_dir my_source_code
```

CMake then runs, first attempting to resolve all the compile time dependencies that the project defines. Note you do not have to generate Makefiles, CMake alos supports XCode and Eclipse projects. Checkout the CMake manpage for the current list.

After the build files have been generated you are free to build thr project. With the default Makefiles, you just need:

```
>make install
```

If you are using Eclipse or XCode you will have to start the build within your environment.

Withing the CI/CD control file above you can see that we invoke CMake with different *BUILD_TYPES* these add specific compiler flags. Please see the cmake documentation for more information.

CMake Coding Conventions

Unfortunately there is even less consensus in the community as to the most effective way to write CMake files, thought there are a lot of opinions. We have found a good distillation of some effective ideas can be found here. And GitLab has an introduction to Modern CMake.

In general however we recommend Modern CMake (minimum version of 3.0). A lot of legacy projects will have used 2.8 - but we cannot recommend that new projects use this. It is no longer maintained and does not contain many features of modern (v3+ CMake). As far as we are aware CMake v3.0 is backward compatible with 2.8, so if you are onboarding code that uses 2.8 we strongly suggest you upgrade.

Todo: Maybe add some more advice? use targets not global settings etc . . .

The SKA CMake Module Repository

We recommend you include the CMake Module Repository as a git submodule.

```
> cd external
> git submodule add https://gitlab.com/ska-telescope/cmake-modules.git
> cd <project_root>
> git add .gitmodules
```

There are two issues with submodules that should be made clear.

1) When you clone your repository you do not get your submodules - in order to populate them you have to:

```
>git submodule sync
>git submodule update --init
```

2) The commit of the submodule that was originally added to the repository is the one that you get when you clone your parent. You can work on the submodule and push if you want. But the main project will only pick up the changes via the .gitmodule directory. Note that GitLab automates this process for CI/CD builds using the GIT_SUBMODULE_STRATEGY: normal.

This directory is linked into the CMake search path by adding:

```
> set(CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/external/cmake-modules")
```

to the top-level CMakeLists.txt.

This directory will be searched ahead of the CMake system installation for Find<Package>.cmake files. This will allow the sharing for MODULE and CONFIG files for common packages

Including the version number

There is a template configuration file that demonstrates how to pull in the version number into the code base. We do this in a few steps. First the Semantic Version number is held in plain text in version.txt. This is done so that it is easy to "bump" the version number on code changes. The second stage is the the CMakeLists.txt file reads this file and creates an internal version number using its contents. Finally the configuration file is used to generate a header file containing the version number that can be included by the code.

Including External Projects

We have including the GoogleTest framework as an external project, instead of a dependency to demonstrate one way to use external projects that you want, or need to build from source. Most dependencies will not need to be built or included this way. The typical method in CMake is to use its find_package() functionality to track down the location of a dependency installed locally. Note that there is no common or standard way of installing dependencies automatically.

Dependencies

As there is no standard we propose that for clarity any external packages that you require be listed in a *dependencies.txt* file the format of entries is this file is defined here to follow the format of the CMake find_package() method i.e.

```
[install | find | both] <package> [version] [EXACT] [QUIET] [MODULE] [REQUIRED]␣
↪[[COMPONENTS] [components...]] [OPTIONAL_COMPONENTS components...] [NO_POLICY_
↪SCOPE])
```

for example

```
find cfitsio REQURIED
```

Except for the initial prefix (install, find or both) the other [] are optional, and are passed directly to the find_package() cmake function.

We have added a *dependencies.cmake* function that will attempt to find all the dependencies listed using the find_package() functionality of CMake. Of course if the build depends upon an external dependency that is not present in the build image, automated CI *will* fail.

Todo: To avoid this we will add other macros to install the external dependencies using a package management tool. When one is downselected.

Therefore package-name is prefixed by one of *install*, *find*, or *both*.

- *install* is used as a keyword so you can parse the file for packages to install (using apt for example),
- *find* is used as a keyword for the dependencies function to use the find_package() functionality. It is possible, indeed likely that the package name for apt and for the cmake module will not be the same. Hence the separate keywords
- *both* for the case where the package has the same name for both the install tool and the cmake module.

Prior to defining a specific dependencies tool, and as apt does not take a list file as an argument, you could use something like this to parse the contents of the file and install the requirements.

```
> cat dependencies.txt | grep -E '^install|^both' | awk '{print $2}' | xargs sudo apt-
↪get
```

Todo: Should such a line to the CI pipeline for the installation of cfitsio as an example, together with a Findcfitsio.cmake module in the ska cmake-modules repository.

Coding Style & Conventions

We are not advocating that software be restructured and rewritten before on-boarding - However we recommend that new software follow The cplusplus Core Guidelines.

The clang/llvm compiler tools have an extension which can provide some direct criticism of your code for stylistic errors (and even automatically fix them). For example in our lint step we suggest you run:

```
run-clang-tidy -checks='cppcoreguidelines-*,performance-*,readibility-*,modernize-*,
↪misc-*,clang-analyzer-*,google-*'
```

Note: The GoogleTest macros generate a lot of warnings . . . Google have their own code guidelines . . .

In the linting stage we also run cppcheck as a separate step.

Warning: The linting stage as presented here is spotting an error in the GTest macros. So we have explicitly removed the test directory from the cppcheck path.

Unit testing

Setting Up The Tests

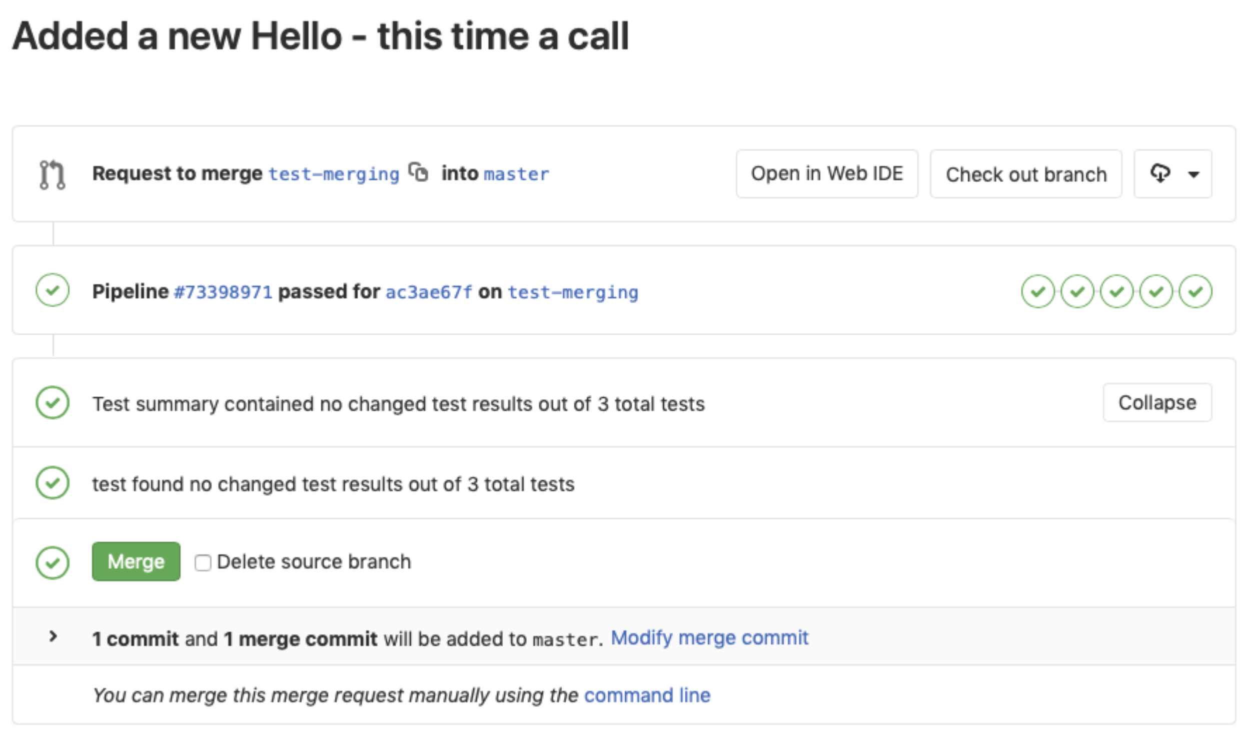
Within the template we give examples of how to write a Unit Test in The Google Test framework.

You will also see from the CI script that we publish the test results in the following manner:

```
stage: test
dependencies:
    - build_debug
script:
    - cd build
    - ctest -T test --no-compress-output
after_script:
    - cd build
    - ctest2junit > ctest.xml
artifacts:
    paths:
    - build/
reports:
  junit: build/ctest.xml
```

How GitLab Can Use The Results

The above directives publish the results to the GitLab JUnit test plugin, but the system is very minimal. Primarily it is used in examining merge requests. When you push to the GitLab repo your branch is built. On completion the test reqults are compared if tests fail you are warned, if they pass you are notified. For example. I have created a test branch for the repo that contains a new derived instance of an hello world. I have included a test and I checked locally that all the existing code built - after the merge request is started GitLab gives the following report:


Added a new Hello - this time a call

Request to merge test-merging into master

Pipeline #73398971 passed for ac3ae67f on test-merging

Test summary contained no changed test results out of 3 total tests

test found no changed test results out of 3 total tests

1 commit and 1 merge commit will be added to master. Modify merge commit

You can merge this merge request manually using the command line

You can see that there is not much detail - but the tests results are parsed and any differences are noted.

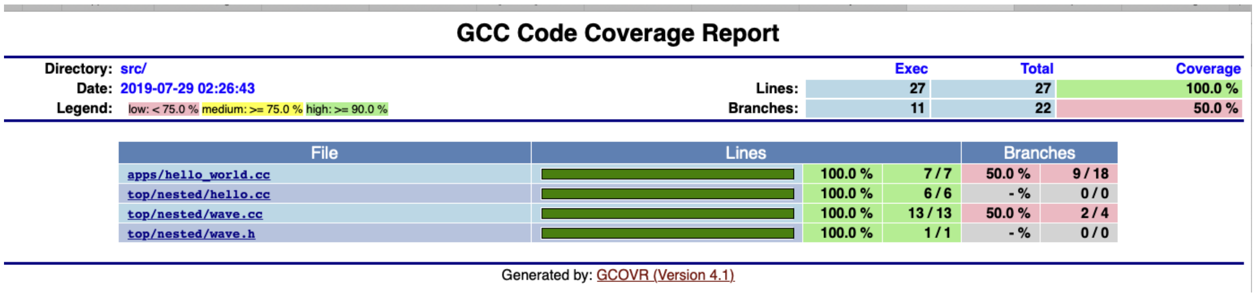
Pages (or Publishing Artifacts)

We use this step to run the test coverage tool gcov, and to publish the results:

```
stage: pages
dependencies:
  - test
script:
  - mkdir .public
  - cd .public
  - gcovr -r ../ -e '.*/external/.*' -e '.*/CompilerIdCXX/.*' -e '.*/tests/.*' --html␣
↪--html-details -o index.html
  - cd ..
  - mv .public public
artifacts:
  paths:
    - public
```

Note that the atifacts step this allows the results to be accessed via the pipelines pages. Every build stores its artifacts from the test step and the pages step.

Note: As far as we know there is no plugin for the coverage artifacts generated by gcov



GCC Code Coverage Report

			Exec	Total	Coverage
Directory:	src/				
Date:	2019-07-29 02:26:43	Lines:	27	27	100.0 %
Legend:	low: < 75.0 % medium: >= 75.0 % high: >= 90.0 %	Branches:	11	22	50.0 %

File	Lines			Branches	
apps/hello_world.cc	🟩	100.0 %	7 / 7	50.0 %	9 / 18
top/nested/hello.cc	🟩	100.0 %	6 / 6	- %	0 / 0
top/nested/wave.cc	🟩	100.0 %	13 / 13	50.0 %	2 / 4
top/nested/wave.h	🟩	100.0 %	1 / 1	- %	0 / 0

Generated by: GCOVR (Version 4.1)

But you can access the artifact from the pipeline.

Summary

This basic template project is available on GitLab. And demonstrates the following:

1) Provides a base image on which to run C++ builds
2) Describes example basic dependency management is possible at least based on CMake but way of CMake External projects and git submodules
3) Presents a convention for defininig third party/external to project libs such that they are independent of the dependency management layer that will be supported by the systems team.
4) Proposes a C++ 14 language standard and related static code checking tools
5) Outlines header naming conventions that follow namespace definitions
6) Proposal of GoogleTest for a common C++ unit testing library

2.16.2 SKA Javascript Coding Guidelines

Prerequisites:

As a javascript developer you will need the following:

- An account on gitlab
- Access to the SKA gitlab account (https://gitlab.com/ska-telescope)
- Access to git (e.g. Git Bash if on Windows environment (https://gitforwindows.org/)
- Access to node and the node package manager (npm)

Setting up a new web project

Each new web project should start by creating a new SKA gitlab project as a fork of the ska-telescope/ska-react-webapp-skeleton project.

- The readme file will need updated to reflect the nature of the new web project

Conventions

JavaScript has many code style guides. For the SKA Projects we have settled on the AirBnB JavaScript Style Guide. Any differences to what is presented by AirBnB should be documented here:

As a developer it is worth familiarizing yourself with the AirBnB guide above, and some of the background reading that they suggest.

File Suffixes

- Use .jsx for any file which contains JSX, otherwise use .js
- Test files should be prefixed .test.{js|jsx}

Code structure

Within each project code should be grouped by features or routes. This is based on the React guidance at https://reactjs.org/docs/faq-structure.html

Locate CSS, JS, and tests together inside folders grouped by feature or route.

```
components/
    ├── App/
    │ ├── App.css
    │ ├── App.jsx
    │ └── App.test.jsx
    └── NavBar/
          ├── NavBar.css
          ├── NavBar.jsx
          └── NavBar.test.jsx
```

The definition of a "feature" is not universal, and it is up to you to choose the granularity. If you can't come up with a list of top-level folders, you can ask the users of your product what major parts it consists of, and use their mental model as a blueprint.

Linting

The react web skeleton project comes with ESLint and Prettier support already configured to support the AirBnB style guide rules. We suggest that whenever possible you verify your code style and patterns in your editor as you code.

Instructions for how to install plugins to support this do this for Visual Studio (VS Code) and JetBrains (WebStorm, IntelliJ IDEA etc.) are include in the ska-react-webapp-skeleton readme file.

Dependencies

- The use of node and npm is assumed for package management. All packages and dependencies required to build and run the code should be defined in the `package.json.` for the project.
- Avoid relying on any the installation of any global packages to run or build the code.
- Take care to differentiate between dependencies and devDependencies when installing packages. The 'dependencies' section should only include the dependencies required to run the code. The devDependencies should be used for any packages required to build, test or to deploy the code.

- Only install packages for a reason. Adding any new third party dependency should be a team decision, and preferably discussed with the system team.
- Remove unused packages
- Prefer popular packages. Sites such as https://www.npmjs.com/search and https://npms.io can be used to get an analysed ranking, as well as checking out the number of stars on gitlab.
- Prefer packages with a good coverage of working tests provided
- Any 3rd party packages used should be compatible with the SKA BSD-3 Licence terms.

When developing packages and modules either for your own project or for sharing with other SKA javascript teams.

- Prefer smaller modules with a coherent closely related purpose
- Prefer files with a single named export where possible
- Ensure all imports use the same names for exports as used in the file where the export is defined.
- Avoid default exports or imports.

Named exports (and using the same names consistently throughout the code) make life easier for anyone using your code.

Having a name makes it possible for IDEs to find and import dependencies for you automatically. For a good perspective on this read: https://humanwhocodes.com/blog/2019/01/stop-using-default-exports-javascript-module/

Documentation and Testing

- All external interfaces should be fully documented.
- All code should be well commented. As a minimum any method used outside the file in which it is written should be documented following standard JSDoc conventions.
- Working unit tests should exist for as much code as possible. At a minimum all code developed or modified within the SKA Project should have working tests (see the definition of done above. We are currently aiming at +75% code coverage for all web projects.

Data and Configuration Files

- Use proxies and relative paths where possible. Avoid hard coded URLs. Any explicit paths should be derived from a consistent configuration source. (See for example https://facebook.github.io/create-react-app/docs/proxying-api-requests-in-development#configuring-the-proxy-manually

Console output, warnings and errors

- Use the debugger rather than console statements.
- Remove all console statements when done. Production code should not contain any console statements.

Todo:

- Testing Guidelines
- Writing Command-Line Scripts
- C or Cython Extensions

2.16.3 Python Coding Guidelines

This section describes requirements and guidelines.

An Example Python Project

We have created a skeleton Python project which should provide a full introduction to the various recommendations and requirements for the development of Python. The philosophy behind the development of this template was to demonstrate one way to meet the project guidelines and demonstrate a recommended project layout.

The recommended project layout is as follows:

```
.
├── CHANGELOG.rst
├── docker-requirements.txt
├── docs
│   ├── Makefile
│   └── src
│       ├── conf.py
│       ├── index.rst
│       ├── package
│       │   └── guide.rst
│       ├── README.md -> ../../README.md
│       ├── _static
│       │   ├── css
│       │   │   └── custom.css
│       │   ├── img
│       │   │   ├── favicon.ico
│       │   │   ├── logo.jpg
│       │   │   └── logo.svg
│       │   └── js
│       │       └── github.js
│       └── _templates
│           ├── footer.html
│           └── layout.html
├── LICENSE
├── Makefile
├── Pipfile
├── README.md
├── setup.cfg
├── setup.py
├── src
│   └── ska
│       └── skeleton
│           ├── example.py
│           └── __init__.py
└── tests
    ├── __init__.py
    └── test_example.py
```

Interface and Dependencies

- All code must be compatible with Python 3.7 and later.
- The new Python 3 formatting style should be used (i.e. `"{0:s}".format("spam")` instead of `"%s" % "spam"`), or use format strings `f"{spam}"` if variable `spam="spam"` is in scope.

Documentation and Testing

- Docstrings must be present for all public classes/methods/functions, and must follow the form outlined by PEP8 Docstring Conventions.
- Unit tests should be provided for as many public methods and functions as possible, and should adhere to Pytest best practices.

Data and Configuration

- All persistent configuration should use python-dotenv. Such configuration `.env` files should be placed at the top of the `ska_python_skeleton` module and provide a description that is sufficient for users to understand the settings changes.

Standard output, warnings, and errors

The built-in `print(...)` function should only be used for output that is explicitly requested by the user, for example `print_header(...)` or `list_catalogs(...)`. Any other standard output, warnings, and errors should follow these rules:

- For errors/exceptions, one should always use `raise` with one of the built-in exception classes, or a custom exception class. The nondescript `Exception` class should be avoided as much as possible, in favor of more specific exceptions (*IOError*, *ValueError*, etc.).
- For warnings, one should always use `warnings.warn(message, warning_class)`. These get redirected to `log.warning()` by default.
- For informational and debugging messages, one should always use `log.info(message)` and `log.debug(message)`.

Logging implementation

There is a standard Python logging module for logging in SKA projects. This module ensures that messages are formatted correctly according to our formatting standards.

For details on how to use the logging module with detailed examples, please refer to: https://gitlab.com/ska-telescope/ska-logging/tree/master#ska-logging-configuration-library

Coding Style/Conventions

- The code will follow the standard PEP8 Style Guide for Python Code. In particular, this includes using only 4 spaces for indentation, and never tabs.
- The `import numpy as np`, `import matplotlib as mpl`, and `import matplotlib.pyplot as plt` naming conventions should be used wherever relevant. `from packagename import *` should never be used, except as a tool to flatten the namespace of a module. An example of the allowed usage is given in *Acceptable use of from module import **.
- Classes should either use direct variable access, or Python's property mechanism for setting object instance variables. `get_value`/`set_value` style methods should be used only when getting and setting the values requires a computationally-expensive operation. *Properties vs. get_/set_* below illustrates this guideline.
- Classes should use the builtin `super()` function when making calls to methods in their super-class(es) unless there are specific reasons not to. `super()` should be used consistently in all subclasses since it does not work otherwise. *super() vs. Direct Calling* illustrates why this is important.

- Multiple inheritance should be avoided in general without good reason.
- `__init__.py` files for modules should not contain any significant implementation code. `__init__.py` can contain docstrings and code for organizing the module layout, however (e.g. `from submodule import *` in accord with the guideline above). If a module is small enough that it fits in one file, it should simply be a single file, rather than a directory with an `__init__.py` file.

Unicode guidelines

For maximum compatibility, we need to assume that writing non-ASCII characters to the console or to files will not work.

Including C Code

- When C extensions are used, the Python interface for those extensions must meet the aforementioned Python interface guidelines.
- The use of Cython is strongly recommended for C extensions. Cython extensions should store `.pyx` files in the source code repository, but they should be compiled to `.c` files that are updated in the repository when important changes are made to the `.pyx` file.
- In cases where C extensions are needed but Cython cannot be used, the PEP 7 Style Guide for C Code is recommended.

Examples

This section shows examples in order to illustrate points from the guidelines.

Properties vs. get_/set_

This example shows a sample class illustrating the guideline regarding the use of properties as opposed to getter/setter methods.

Let's assuming you've defined a `':class:`Star`'` class and create an instance like this:

```
>>> s = Star(B=5.48, V=4.83)
```

You should always use attribute syntax like this:

```
>>> s.color = 0.4
>>> print(s.color)
0.4
```

Using Python properties, attribute syntax can still do anything possible with a get/set method. For lengthy or complex calculations, however, use a method:

```
>>> print(s.compute_color(5800, age=5e9))
0.4
```

super() vs. Direct Calling

By calling `super()` the entire method resolution order for `D` is precomputed, enabling each superclass to cooperatively determine which class should be handed control in the next `super()` call:

```
# This is safe

class A(object):
    def method(self):
        print('Doing A')

class B(A):
    def method(self):
        print('Doing B')
        super().method()


class C(A):
    def method(self):
        print('Doing C')
        super().method()

class D(C, B):
    def method(self):
        print('Doing D')
        super().method()
```

```
>>> d = D()
>>> d.method()
Doing D
Doing C
Doing B
Doing A
```

As you can see, each superclass's method is entered only once. For this to work it is very important that each method in a class that calls its superclass's version of that method use `super()` instead of calling the method directly. In the most common case of single-inheritance, using `super()` is functionally equivalent to calling the superclass's method directly. But as soon as a class is used in a multiple-inheritance hierarchy it must use `super()` in order to cooperate with other classes in the hierarchy.

Note: For more information on the the benefits of `super()`, see https://rhettinger.wordpress.com/2011/05/26/super-considered-super/

Acceptable use of `from module import *`

`from module import *` is discouraged in a module that contains implementation code, as it impedes clarity and often imports unused variables. It can, however, be used for a package that is laid out in the following manner:

```
packagename
packagename/__init__.py
packagename/submodule1.py
packagename/submodule2.py
```

In this case, `packagename/__init__.py` may be:

```
"""
A docstring describing the package goes here
"""
```

(continues on next page)

(continued from previous page)

```
from submodule1 import *
from submodule2 import *
```

This allows functions or classes in the submodules to be used directly as `packagename.foo` rather than `packagename.submodule1.foo`. If this is used, it is strongly recommended that the submodules make use of the `__all__` variable to specify which modules should be imported. Thus, `submodule2.py` might read:

```
from numpy import array, linspace

__all__ = ['foo', 'AClass']

def foo(bar):
    # the function would be defined here
    pass

class AClass(object):
    # the class is defined here
    pass
```

This ensures that `from submodule import *` only imports `':func:`foo'` and `':class:`AClass'`, but not `':class:`numpy.array'` or `':func:`numpy.linspace'`.

. _python-packaging:

Packaging

SKA python packages use setuptools to assemble the packages

Apart from the standard arguments to `setup()`, several extra enhancements are used.

Running tests out of the top-level directory can lead to conflicts when the package is a direct child. It is best to put the package code in a subdirectory e.g. `src`

`package_dir={"": "src"}`

Package layout:

```
setup.py
setup.cfg
requirements.txt
requirements-test.txt
src/ska/foo/__init__.py
src/ska/foo/bardevice.py
tests/__init__.py
tests/test_bardevice.py
docs/requirements.txt
docs/source/conf.py
docs/source/index.rst
```

Namespace

It is recommended to use the `ska` namespace package for modules developed in Python which are directly related to SKA. Namespace packages in python3 are native and distinguish themselves as directories without `__init__.py` files. They have to be declared by using the output of `setuptools.find_namespace_packages()` to supply to the `packages` keyword in `setup()`

Requirements

There are many ways to handle the installation of dependencies in `python`. *pip* Best practice though is to put direct dependencies into *install_requires* usually with a lower compatibility bound, but not explicit, e.g.

```
install_requires=[
     "pytango >= 9.3.2",
     "lmcbaseclasses >= 0.5.4"
 ]
```

For testing a package there is better to use and explicit *requirements.txt* file rather than `setup(test_requires=)`. A typical development scenario could look like:

```
pip install -e . # pulls in dependencies via install_requires
pip install -r requirements-test.txt
pytest tests
```

Entry points

If your code contains scripts or *main* functions in your module, these can automatically wrapped as executables in the deployed package. For example Tango `Device Classes` can be exposed without adding wrapper scripts

`entry_points=`

Sample `setup.py`

Here is a sample for the *foo* module in the `ska` namespace package:

```
setuptools.setup(
    name="foo.bar",
    description="Foo stuff",
    version=0.0.1,
    author="Prof Dr Dr Foo",
    author_email="foo AT bar DOT org",
    license="IP here",
    url="https://foo.bar.org",
    classifiers=[
        "Development Status :: 3 - Alpha",
        "Intended Audience :: Developers",
        "License :: Other/Proprietary License",
        "Operating System :: OS Independent",
        "Programming Language :: Python",
        "Topic :: Software Development :: Libraries :: Python Modules",
        "Topic :: Scientific/Engineering :: Astronomy"],
    platforms=["OS Independent"],
    package_dir={"": "src"},
    packages=setuptools.find_namespace_packages(where="src"),
    entry_points={
        "console_scripts": [
            "FooDevice=ska.foo.bar_device:main",
        ]
    },
    include_package_data=True,
    install_requires=[
```

(continues on next page)

(continued from previous page)

```
        # should be pulled in by lmcbaseclasses but isn't
        "pytango >= 9.3.2",
        "lmcbaseclasses >= 0.5.4"
    ],
    keywords="foo tango ska",
    zip_safe=False
)
```

Reproducible workflow

Todo: This section is still evolving

While testing in a local environment is quick and easy it doesn't fully guarantee independence of the system. A widely used way to abstract environment handling is tox, which can control your testing workflow through several stages similar to what various continuous integration pipelines do.

Simply install via `pip install tox`

Here is a SKA pytango package example tox.ini:

```
[tox]
envlist = py37

[testenv]
setenv = PIP_DISABLE_VERSION_CHECK = 1
install_command = python -m pip install --extra-index-url https://nexus.engageska-
↪portugal.pt/repository/pypi/simple {opts} {packages}
deps =
    -rrequirements.txt  # runtime requirements
    -rrequirements-test.txt   # test/development requirements
commands =
    # this ugly hack is here because:
    # https://github.com/tox-dev/tox/issues/149
    python -m pip install -U --extra-index-url https://nexus.engageska-portugal.pt/
↪repository/pypi/simple -r{toxinidir}/requirements.txt
    #
    python -m pytest {posargs}
# use system site-packages for pytango (and c++ library dependencies)
sitepackages = true

[testenv:docs]
description = build documentation
basepython = python3
sitepackages = false # we want to run docs without pytango, as that isn't available␣
↪on RTD
skip_install = true
install_command = python -m pip install -U {opts} {packages}
deps = -rdocs/requirements.txt
commands =
    sphinx-build -E -W -c docs/source/ -b html docs/source/ docs/build/html

[testenv:lint]
basepython = python3
```

(continues on next page)

(continued from previous page)

```
skip_install = true
description = report linting
whitelist_externals = mkdir
deps = -rrequirements-tst.txt
commands =
    - mkdir -p build/reports
    - python -m flake8 --max-line-length=88 --format=junit-xml --output-file=build/
↪reports/linting.xml
    python -m flake8 --max-line-length=88 --statistics --show-source
```

To use *tox* simple invoke as follows:

```
tox -e py37
tox -e lint
...
```

Acknowledgements

The present document's coding guidelines are derived from project Astropy's coding guidelines.

Note: Futher discussion on firmware development process, continuous integration, automated testing and recommendations for best practices can be found in the following google doc. This document will be gradully implemented and the resulting decision will for part of this developer portal.

https://docs.google.com/document/d/1Kfc_4vLUy-0pSbi9HVeEkAmhuvEIEnt4voFnXxsc0zM/edit?usp=sharing

2.16.4 VHDL Coding Style Guidelines

Following a coding style is an integral part of robust development. It should not be a burden, and something done afterwards to pass code review – it should be done continuously at all stages of code development. A clean coding style is desired. This allows other team members (and yourself in a month or a year) to read and digest your code quickly, and to use the code with a high confidence in its correctness.

The VHDL coding guidelines developed and published by ALSE are excellent, and will form the basis of this coding guideline. They are available from their website:

http://www.alse-fr.com/VHDL-Coding-Guide.html

and as a pdf:

http://www.alse-fr.com/sites/alse-fr.com/IMG/pdf/vhdl_coding_v4_eng.pdf.

The following numbered additions/modifications/clarifications are used:

The VHDL-2008 standard will be used, so far as its features are supported by synthesis and simulation tools. Synthesis tools' support for VHDL-2008 features has improved to a level where its usage can result in simpler, less verbose, and more descriptive code.

P_11) VHDL keywords shall be in lower case. Use an editor with syntax highlighting.

N_8) Active low signals should be avoided. In preference signals should be named to make them active high (positive logic). If no such obvious name can be found then active low signal names should have the suffix _n. For example the following are equivalent:

```
tx_disable == tx_enable_n.
```

Note that in VHDL 2008, operators can be applied in port maps. So conversion can be applied without having to create an intermediate signal (and hence specify a name for it). For example:

```
E_EXT_MOD : entity extern_lib.module
port map ( ...
  i_tx_enable_n <= not tx_enable,
  ...);
```

N_9) When ports of mode 'out' need to be accessed internally, the derived internal signals shall be named using <out port name>_i (for example o_outbus_i being the internally accessible signal from which the output port o_outbus is directly derived. However, in VHDL-2008 the output port can be accessed directly making this irrelevant.

N_10) Use instance names derived from the entity names. Prefix E_ should be used to identify the direct instantiation of the entity, and C_ for instantiation from a component declaration. The label shall be in all upper case. For example:

```
E_FIR16X8 : entity work.fir16x8 port map ( etc...
C_FIR16X8 : fir16x8 port map ( etc...
```

If there are multiple instances then instance names should have a descriptor appended that adds information. Avoid simply appending a number (consider using a generate loop instead). For example:

```
E_EMIF_BOTTOM_RIGHT : entity work.external_memory_interface port map ( ...
E_EMIF_TOP_LEFT     : entity work.external_memory_interface port map ( ...
E_EMIF_TOP_RIGHT    : entity work.external_memory_interface port map ( ...
```

N_14) Names for clock and reset signals shall conform to the following convention:

- Clocks: `<name>_clk`
- Reset: `<name>_clk_<reset_name>_rst` (active high reset, synchronised to `<name>_clk`)
- Reset: `<name>_clk_<reset_name>_rst_n` (active low reset, synchronised to `<name>_clk`)

`<name>` should be shared with the signals in the clock's domain.

An entity with a single clock, should have the input clock `i_clk`, and with a single active high synchronous reset `i_clk_rst`.

N_15) Constants shall use the c_ prefix, and the name be capitalised, for example:

```
constant c_BYTE_WIDTH : natural := 8;
```

N_11) Generics shall use the g_ prefix, and the name be capitalised, for example:

```
generic (g_BLOCK_LENGTH : natural := 256);
```

N_12) Variables shall use the v_ prefix, for example:

```
variable v_sum : unsigned(7 downto 0);
```

N_13) Process labels shall use the P_ prefix, and be capitalised for example:

```
P_DO_READ : process(i_clk)
Begin
    if rising_edge(i_clk) then
        ...
    end if;
end process P_DO_READ;
```

N_12) Generate labels should use the G_ prefix, and be capitalised, for example:

```
G_USE_BUFFER : if not g_SIM generate
    -- instantiate buffer module
else generate
    -- default signal assignments
end generate G_USE_BUFFER;
G_EACH_DATA : for idx in 0 to g_DATA_WIDTH-1 generate
    -- assignments/instantiations for each bit in the data.
end generate G_EACH_DATA;
```

C_6a) Allow numeric_std version of unsigned and signed types in ports. This increases the information in port description, giving meaning to the bit vector that is not available when declared as a std_logic_vector.

2.17 Frequently Asked Questions

2.17.1 I want to..

Add a new project to SKA organisation

- *Create a new project*

Develop a Tango device

- A sample PyTango device project that can be forked can be found at https://gitlab.com/ska-telescope/tango-example/
- Documentation for it can be found at https://developer.skatelescope.org/projects/tango-example/en/latest/?badge=latest

Containerise my solution

A summary of our containerisation standards can be found in the *Container Standards CheatSheet*.

- **Verify Docker installation**
    - Docker installation instructions:

```
$ docker -v

  Docker version 1.7.0, build 0baf609
```

Incorporate my project into the integration environment

We use Kubernetes as an orchestration layer - refer to our *Orchestration Guidelines*.

Once a project is ready to form part of the integrated solution, we need to verify that all prerequisites are installed and working properly.

- **Verify kubectl installation**
    - kubectl installation instructions.

```
$ kubectl version

  Client Version: version.Info{Major:"1", Minor:"15", GitVersion:"v1.15.1", GitCommit:
↪"4485c6f18cee9a5d3c3b4e523bd27972b1b53892",
  GitTreeState:"clean", BuildDate:"2019-07-18T09:18:22Z", GoVersion:"go1.12.5",␣
↪Compiler:"gc", Platform:"linux/amd64"}
  The connection to the server localhost:8080 was refused - did you specify the right␣
↪host or port?
```

- **Verify Minikube installation**
  - Minikube installation instructions.

```
$ minikube

  Minikube is a CLI tool that provisions and manages single-node Kubernetes clusters␣
↪optimized for development workflows...
```

- **Launch Kubernetes.**
  - Look out for *kubectl is now configured to use "minikube"* near the end:

```
 $ sudo -E minikube start --vm-driver=none --extra-config=kubelet.resolv-conf=/var/
↪run/systemd/resolve/resolv.conf

   minikube v0.34.1 on linux (amd64)
   Configuring local host environment ...

   The 'none' driver provides limited isolation and may reduce system security and␣
↪reliability.
     For more information, see:
   https://gitlab.com/kubernetes/minikube/blob/master/docs/vmdriver-none.md

   kubectl and minikube configuration will be stored in /home/ubuntu
   To use kubectl or minikube commands as your own user, you may
   need to relocate them. For example, to overwrite your own settings:

     sudo mv /home/ubuntu/.kube /home/ubuntu/.minikube $HOME
     sudo chown -R $USER /home/ubuntu/.kube /home/ubuntu/.minikube

  This can also be done automatically by setting the env var CHANGE_MINIKUBE_NONE_
↪USER=true
  Creating none VM (CPUs=2, Memory=2048MB, Disk=20000MB) ...
  "minikube" IP address is 192.168.86.29
  Configuring Docker as the container runtime ...
  Preparing Kubernetes environment ...
    kubelet.resolv-conf=/var/run/systemd/resolve/resolv.conf
  Pulling images required by Kubernetes v1.13.3 ...
  Launching Kubernetes v1.13.3 using kubeadm ...
  Configuring cluster permissions ...
  Verifying component health .....
  kubectl is now configured to use "minikube"
  Done! Thank you for using minikube
```

Test that the connectivity in the cluster works

```
$ kubectl get pods -n kube-system
NAME                                   READY   STATUS    RESTARTS   AGE
```

(continues on next page)

(continued from previous page)

```
coredns-86c58d9df4-5ztg8           1/1     Running   0          3m24s
...
```

- **Verify Helm installation**
  - Helm installation instructions

```
$ helm version
version.BuildInfo{Version:"v3.1.2", GitCommit:
↪"d878d4d45863e42fd5cff6743294a11d28a9abce", GitTreeState:"clean", GoVersion:"go1.13.
↪8"}

...
```

Note: Until recently, we have been using Helm 2 in all our Ansible Playbooks for provisioning machines and development environments. Helm 2 used Tiller as a deployment service, and would be started in your environment by running the `helm init` command. **This is no longer the case with Helm 3.** More info here .

Use this ansible playbook to upgrade your existing Helm 2 to Helm 3.

Once Helm is installed, develop a helm chart for the project. Refer to *Helm instructions* for guidelines.

Install Helm charts from our own repository

The SKAMPI repository is in essence a configuration management repository, which basically just consists of a number of Helm charts and instructions for installing them on a kubernetes cluster.

Installing Helm charts from our own Helm Chart Repository is another option, specifically that enables installing different charts during run-time.

To add the SKA Helm chart repo to your local Helm, simply run

```
$ helm repo add skatelescope https://nexus.engageska-portugal.pt/repository/helm-chart
```

Working with the Helm chart repository, including how to package and upload charts to our repository, is described *here in detail*.

Deploy the TMC prototype and Webjive in kubernetes

The integration gitlab repository can be found at https://gitlab.com/ska-telescope/skampi.

Documentation on deployment can be found at https://developer.skatelescope.org/projects/skampi/en/latest/README.html

Add the helm chart to the skampi repository: *Integrating a chart into the SKAMPI repo*.

- **Verify k8s integration**
  - Launch the integration environment

```
$  make deploy_all KUBE_NAMESPACE=integration
```

and verify that the pods are able to startup without any errors

```
$ watch kubectl get all,pv,pvc,ingress -n integration

Every 2.0s: kubectl get all,pv,pvc -n integration          osboxes: Fri Mar 29␣
↪09:25:05 2019

NAME                                          READY    STATUS             RESTARTS   ␣
↪AGE
pod/databaseds-integration-tmc-webui-test-0   1/1      Running            3          ␣
↪117s
pod/rsyslog-integration-tmc-webui-test-0      1/1      Running            0          ␣
↪117s
podtangodb-integration-tmc-webui-test-0      1/1      Running            0          ␣
↪117s
pod/tangotest-integration-tmc-webui-test      1/1      Running            2          ␣
↪117s
pod/tmcprototype-integration-tmc-webui-test   4/5      CrashLoopBackOff   2          ␣
↪117s
pod/webjive-integration-tmc-webui-test-0      4/4      Running            0          ␣
↪117s
...
```

CHAPTER 3

Development tools and practices

3.1 SKA git repositories

The SKA uses git as its distributed version control system, and all SKA code shall be hosted in an SKA organisation. The gitlab organization *ska-telescope* can be found at https://gitlab.com/ska-telescope. All SKA developers must have a gitlab account and be added to the organisation. See *Working with GitLab* for further details.

3.2 Working with SKA Jira

Every team is tracking daily work in a team-based project on our JIRA server at https://jira.skatelescope.org. Our internal wiki, Confluence, has guidance on how we use JIRA. We rely on integrations between GitLab and JIRA to manage our work.

- *Working with Jira*

3.3 CI/CD: Continuous Integration and Deployment

CI/CD is at the heart of SKA development, and we use GitLab's automation extensively, so we can test and deploy our software more efficiently.

- *CI/CD*

3.4 Testing

Tests are a key part of producing working software. We suggest you look at our *Software Testing Policy and Strategy*, and our *BDD testing guide* and *BDD Walkthrough*.

3.5 Test Infrastructure

To support our testing and CI/CD pipelines, we have the EngageSKA and other clusters configured to allow testing to happen.

- *EngageSKA: CI/CD & Testing Infrastructure*
- *EngageSKA cluster*

3.6 Containerisation

To facilitate code portability and reliability and test running, we use containers. We also use kubernetes as our container orchestration system.

- *Containers: containerisation and deployment*
- *Containerisation Standards*
- *A Quick Introduction to Kubernetes*
- *Container Orchestration Guidelines*
- *Multitenancy*
- *Working with Docker: Proxy Cache*
- *Hosting a docker image on the Central Artefact Repository*

3.7 Documentation

While we prefer working code over documentation (as Agile developers), we also recognise that this is a large and long-lived project, so documentation has an important place.

- *Documenting a project*

3.8 Package Release Process

What you need to know in order to release an SKA software package.

- *Software Package Release Procedure*

3.9 Logging

Making sure your software project outputs useful logs for the SKA

- *SKA Log Message Format*

3.10 Monitoring Dashboards

You've deployed your code on one of our test systems. Now you want to monitor it.

- *Monitoring Dashboards*

3.11 Bug Reporting

What to do when you find a bug in SKA code.

- *Reporting Bugs*

3.12 Coding Guidelines

Guidelines to the coding standards we apply in the SKA. Not available for all languages.

- *Coding guidelines*
- *C++ Coding Guidelines*
- *SKA Javascript Coding Guidelines*
- *Python Coding Guidelines*
- *VHDL Coding Style Guidelines*

3.13 FAQ

Questions frequently asked by developers.

- *Frequently Asked Questions*

3.13.1 SKA Code of Conduct

SKA Organisation (SKAO) is committed to the highest standards of business ethics and as such expects everyone involved in SKAO-related business to uphold the standards and expected professional behavior set out in SKA Code of Ethics page .

The code of ethics applies to every SKA collaborators and it is the reference guide defining the culture of this online community of contributors.

- Download the SKA Code of Ethics

3.13.2 Definition of Done

Done-ness is defined differently at different stages of development and for different purposes.

Team Increment

Issue Type	Definition of Done
Story/Enabler	**Code** • Supplied with an *Acceptable License*. • Adheres to SKA language *specific style*. • Checked into SKA repository with a *reference* to a JIRA ticket ID. • Passes the CI/CD pipeline including compiling cleanly and being linted with no warnings: *Linting*. • Unit and module *tests* pass with adequate coverage (>= 75% with appropriate exclusions for boiler-plate code). • Component, integration and system *tests* (appropriate for the context) pass. • Regression *tests* pass. *Code Documentation* • Exposed Public *API* (where applicable) is cleanly documented. • Documented inline according to *language specific standards*. • Deployed to externally visible website accessible via the *Integration into the Developer Portal*. **Integration** • Deployed to a *Continuous Integration* environment (staging environment during Construction). • Migrations are implemented with defined automated processes for roll-forward and rollback as appropriate. **Process** • Peer-reviewed and integrated into the main branch via GitLab *Merge requests*. • Relevant NFRs are met • Satisfies acceptance criteria • Accepted by Product Owner
Spike	**Documentation** • Outcomes documented on the relevant SKA platform • Documentation linked to issue in Jira **Process** • Outcomes reviewed by relevant stakeholders • Satisfies acceptance criteria • Accepted by Product Owner

System Increment

Issue Type	Definition of Done
Feature/Enabler	**Child Stories/Enablers** • Completed by all teams and integrated in an *integration environment* (staging environment during Construction). **Documentation** • Solution Intent or project documentation updated to reflect the actual implementation. **Process** • Satifies acceptance criteria • Relevant NFRs are met • Demonstrated to relevant stakeholders • Accepted by Feature Owner
Spike	**Documentation** • Outcomes documented on the relevant SKA platform • Documentation linked to issue in Jira **Process** • Outcomes reviewed by relevant stakeholders • Satisfies acceptance criteria • Accepted by Spike Owner

Solution Increment

Issue Type	Definition of Done
Capability/Enabler	**Child Stories/Enablers** • Completed by all ARTs and integrated in an *integration environment* (staging environment during Construction) **Documentation** • Solution Intent or project documentation updated to reflect the actual implementation **Process** • Satifies acceptance criteria • Relevant NFRs are met • Demonstrated to relevant stakeholders • Accepted by Capability Owner

Release

Issue Type	Definition of Done
TBD	TBD

Formally Controlled Project Documentation

Documents that are matured to the extent that they quantify an impact on safety, security, quality, schedule, cost, profit or the environment should be validated and formally controlled as per the SKA Document Creation, Validation and Release Standard Operating Procedure (SOP) (SKA-TEL-SKAO-0000765). Until such time, the Lightweight Document Process and Repository may used to manage these documents.

Thereafter, these documents must be formally reviewed and placed in the project's configuration management system. Whilst there is an unavoidable overhead to this we aim to make it as efficient as possible. However, this level of documentation requires you to follow the process in the Configuration Management part of Confluence, specifically:

- Document number obtained by completing and forwarding the New Document Request Form to mailto:cm@skatelescope.org.
- Document is reviewed by suitable reviewer(s).
- Document is in eB and signed off.

3.13.3 Software Testing Policy and Strategy

List of abbreviations

ABBR	MEANING
CD	Continuous Delivery
CI	Continuous Integration
ISTQB	International Software Testing Qualifications Board
PI	Program Increment (SAFe context)
PO	Product Owner
SAFe	Scaled Agile Framework
SKA	Square Kilometre Array
SKAO	SKA Organisation
TDD	Test Driven Development

0 In a nutshell

Important: This is a quick summary of the do's and don't do's entailed by this document.

For developers and teams

- Each team should have a role responsible for the quality of the tests delivered by the team.
- All developers within a team are responsible for creating tests.
- The only source of truth are the tests running in the CI pipeline. "It works for me" is a no go.
- Pay attention to code coverage: you should be able to defend why the parts not covered by tests are not creating a significant risk.
- Unit/module and system levels tests are all needed.
- Relentlessly improve the quality of testware.

- Each team (typically via the role above) must work with the Testing CoP to increase the quality level of the whole codebase by sharing experiences and developing new standards.
- Practice a test-first/TDD approach: first the tests, then the production code.

For Product/Feature/Capability Owners

- Each feature/capability must have at least 1 automated acceptance test expressed in an appropriate, comprehensible way that is automated (possibly a BDD/Gherkin test).
- Product/feature/enabler/capability owners must ensure testing reflects the requirements of the feature/enabler/capability.
- Malfunctions in testware are high priority fixes to do.
- Bugs are logged in some backlog (team backlog, global one, SKAMPI).

1 Introduction

What follows is the software testing policy and strategy produced by Testing Community of Practice.

This is **version 1.2.1** of this document, completed on 2020-06-09.

1.1 Purpose of the document

The purpose of the document is to specify the testing policy for SKA software, which answers the question “why should we test?”, and to describe the testing strategy, which answers “how do we implement the policy?”.

The policy should achieve alignment between all stakeholders regarding the expected benefit of testing. The strategy should help developers and testers to understand how to define a testing process.

1.2 Scope of the document

This policy and this strategy apply exclusively to software-only SKA artifacts that are developed by teams working within the SAFe framework. As explained below, a phased adoption approach is followed, and therefore it is expected that the policy and the strategy will change often, likely at least twice per year until settled.

The document will evolve quickly during the SKA Bridging phase in order to reach a good level of maturity prior to the SKA1 construction start.

Each team is expected to comply with the policy and to adopt the strategy described here, or define and publish a more specific strategy in cases this one is not suitable.

1.3 Applicable documents

The following documents are applicable to the extent stated herein. In the event of conflict between the contents of the applicable documents and this document, the applicable documents shall take precedence.

1. SKA-TEL-SKO-0000661 - Fundamental SKA Software and Hardware Description Language Standards
2. SKA-TEL-SKO-0001201 - ENGINEERING MANAGEMENT PLAN

1.4 Reference documents

International Software Testing Qualification Board - Glossary https://glossary.istqb.org

See other referenced material at the end of the document.

2 Adoption strategy

Testing within the SKA will be complex for many reasons, including a broad range of programming languages and frameworks, dispersed geographical distribution of teams, diverse practices, extended life-time, richness and complexity of requirements, need to cater for different audiences, among others.

In order to achieve an acceptable level of quality, many types of testing and practices will be required, including: coding standards, unit testing/code coverage, functional testing, multi-layered integration testing, system testing, performance testing, security testing, compliance testing, usability testing.

In order to establish a sustainable testing process, we envision a phased adoption of a proper testing policies and strategies, tailored to the maturity of teams and characteristics of the software products that they build: different teams have different maturity, and over time maturity will evolve. Some team will lead in maturity; some other will struggle, either because facing difficult-to-test systems, complex test environments or because they started later.

An important aspect we would like to achieve is that the testing process needs to support teams, not hinder them with difficult-to-achieve goals that might turn out to be barriers rather than drivers. Only after teams are properly supported by the testing process we will crank up the desired quality of the product and push harder on the effectiveness of tests.

We envision 3 major phases:

- Enabling teams, from mid 2019 for a few PIs
- Establishing a sustainable process, for a few subsequent PIs
- Keep improving, afterwards.

3 Phase 1: Enabling Teams

This early phase started in June 2019 and covers several Program Increments.

The overarching goal is to establish a test process that supports the teams. In other terms this means that development teams will be the major stakeholders benefiting by the testing activities that they will do. Testing should cover currently used technologies (which include Tango, Python, C++, Javascript), it should help uncovering risks related with testability of the tested systems and with reliability issues of the testing architecture (CI/CD pipelines, test environments, test data).

As an outcome of such a policy it is expected that appropriate technical practices are performed regularly by each team (eg. TDD, test-first, test automation). This will allow us in Phase 2 to crank up quality by increasing testing intensity and quality, and still have teams following those practices.

It is expected that the systems that are built are modular and testable from the start, so that in Phase 2 the roads are paved to enable increase of quality and provide business support by the testing process in terms of monitoring the quality.

One goal of this initial phase is to create awareness of the importance given to testing by upper management. Means to implement a testing process are provided (tools, training, practices, guidelines) so that teams can adopt them.

We cover these practices:

- TDD and Test-First.
- Use of test doubles (mocks, stub, spies).

- Use and monitoring of code coverage metrics. Code coverage should be monitored especially for understanding which parts of the SUT have NOT been tested and if these are important enough to be tested. We suggest that branch coverage is used whenever possible (as opposed to statement/line coverage).
- Development of system-level tests to cover acceptance criteria (of capabilities and features/enablers/capabilities that are related to the entire system).

Another goal of this phase is **identifying the test training needs** for the organization and teams and start providing some support (bibliography, slides, seminars, coaching).

In order to focus on supporting the teams, we expect that the testing process established in this initial phase should NOT:

- enforce strict mandatory policies regarding levels of coverage of code (regardless of the coverage criteria such as statements, branch, or variable usage-definition), of data, of requirements and of risks;
- rely on exploratory testing (which will be introduced later on);
- define strict entry/exit conditions for artefacts on the different CI stages to avoid creating stumbling blocks for teams;
- provide traceability of risks.

We will focus on these aspects in subsequent phases.

On the other hand, the testing process should help creating testable software products, it should lead to a well-designed test automation architecture, teams should become exposed and should practice TDD, test-first, and adopt suitable test automation patterns.

An easy-to-comply test policy is suggested, and a strategy promoting that testing should be applied during each sprint, automated tests should be regularly developed at different levels (unit, component, integration), regression testing should be regularly done, test-first for bugs and refactorings should be regularly done.

Basic monitoring of the testing process will be done, to help teams improve themselves and possibly to create competition across teams. Test metrics will include basic ones dealing with the testing process, the testing architecture and the product quality.

4 Testing policy

This policy covers all sorts of software testing performed on the code bases developed by each of the SKA teams. There is only one policy, and it applies to all software developed within/for SKA.

4.1 Key definitions

When dealing with software testing, many terms have been defined differently in different contexts. It is important to standardise the vocabulary used by SKA1 in this specific domain according to the following definitions, mostly derived from *ISTQB Glossary*.

Testing The process consisting of all lifecycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect bugs.

Debugging The process of finding, analyzing and removing the causes of failures in software.

Bug A flaw in a component or system that can cause the component or system to fail to perform its required function, e.g. an incorrect statement or data definition. Synonyms: defect, fault.

Failure/Symptom Deviation of the component or system from its expected delivery, service or result.

Error A human action that produces an incorrect result (including inserting a bug in the code or writing the wrong specification).

Test Case A test case is a set of preconditions, inputs, actions (where applicable), expected results, and postconditions, developed based on test conditions. Please, note the difference with "Test script". See also "Test Condition".

Test Script A test script is an implementation of a "Test Case" in a particular programming language and test automation framework. Please, note the difference with "Test Case". See also "Test Condition".

Test Condition A testable aspect of a component or system identified as a basis for testing. More informally, a test condition expresses how the SUT will be exercised and how it will be verified.

Note: The purpose of defining both "testing" and "debugging" is so that readers get rid of the idea that one does testing while they are doing debugging. They are two distinct activities.

4.2 Work organization

Testing is **performed by the team(s) who develop the software**. There is no dedicated group of people who are in charge of testing, there are no beta-testers.

Some testing is performed by a **temporary team of testers**, composed by individuals from different existing teams. This testing is typically system or integration testing that covers 2 or more artefacts that are developed by 2 or more teams.

4.3 Goals of testing

The overarching goal of this version of the policy is **to establish a testing process that supports the teams.**

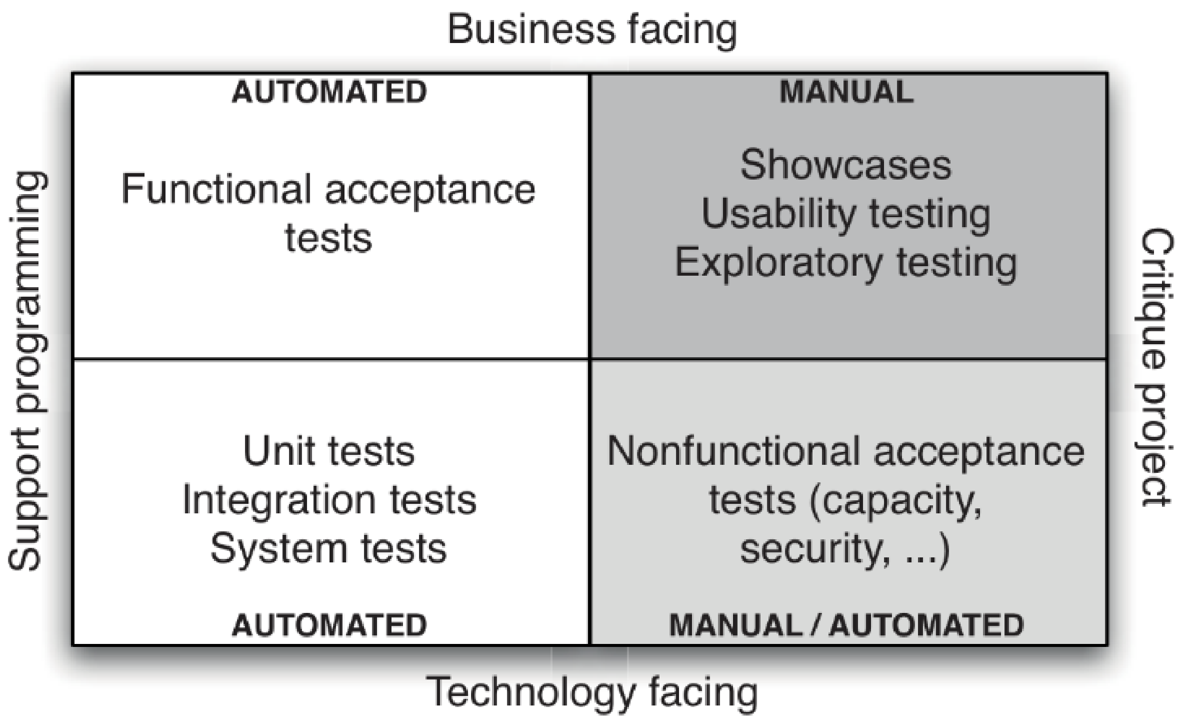
With reference to the test quadrants (*Figure 1*), this policy is restricted to tests supporting the teams, and it includes both quadrants on the left of the picture: tests that are technology facing, hence quadrant Q1 (bottom left) and functional acceptance tests, in Q2 (top left).

Figure 1: Test quadrants, picture taken from (Humble and Farley, Continuous Delivery, 2011).

The expected results of applying this policy are that effective technical practices are performed regularly by each team (eg. TDD, test-first, test automation). The testing process that will be established should help creating testable software products, it should lead to a well-designed test automation architecture, it should push teams to practice TDD, test-first, and adopt suitable test automation patterns. The process should also lead to collect a substantial set of automated tests for testing the system, mostly in the form of Gherkin tests.

The reason is that in this way the following higher level objectives can be achieved:

- team members explore different test automation frameworks and **learn how to use them** efficiently;
- team members learn how to implement tests via **techniques** like test driven development and test-first, how to use test doubles, how to monitor code coverage levels, how to do pair programming and code reviews;
- as an effect of adopting some of those techniques, teams reduce technical debt or keep it at bay; therefore they become more and **more efficient in code refactoring** and in writing high quality code;
- they will become more proficient in increasing quality of the testing system, so that it becomes **easily maintainable**;
- by adopting some of those techniques, teams will develop systems that are more and **more testable**; this will increase modularity, extendability and understandability of the system, hence its quality;



- team members become used to developing automated tests **within the same sprint** during which the tested code is written;
- reliance on automated tests will **reduce the time** needed for test execution and **enable regression testing** to be performed several times during a sprint (or even a day).
- System-level tests are needed to verify **acceptance criteria** of features, enablers and capabilities that are related to the system (as opposed to individual components or services). Writing them in Gherkin has the added value of providing the means to establish a **living documentation**, that is using test cases as a documentation of the business logic and policies implemented by the system.
- System-level tests can be linked to higher level requirements (L1, L2, IF requirements, in addition to acceptance criteria). In this way we can provide **traceability** of tests to requirements, and easily determine what is the degree of satisfaction of certain requirements based on test results.

Expected outcomes are that once testable systems are produced, a relatively large number of unit/module tests will be automated, new tests will be regularly developed within sprints, refactorings will be "protected" by automated tests, and bug fixes will be confirmed by specific automated tests, teams will be empowered and become efficient in developing high quality code. From that moment, teams will be ready to improve the effectiveness of the testing process, which will gradually cover also the other quadrants.

In this phase we can still expect a number of bugs to still be present, to have only a partial assessment of "fitness for use", test design techniques not to be mastered, non-functional requirements not be systematically covered, testing process not to be extensively monitored, systematic traceability of tests to requirements not to be covered, and monitoring of quality also not to be covered. These are objectives to be achieved in later phases, with enhancements of this policy.

4.4 Monitoring implementation of the policy

Adoption of this policy needs to be monitored in a lightweight fashion. We suggest that each team regularly (such as at each sprint) reports the following (possibly in an automatic way):

- total number of test cases
- percentage of automated test cases
- number of test cases/lines of source code
- number of logged open bugs
- bug density (number of open logged defects/lines of source code)
- age of logged open bugs
- number of new or refactored test cases/sprint
- number of test cases that are labelled as "unstable" or that are skipped
- code coverage (the "execution branch/decision coverage" criterion is what we would like to monitor, for code that was written by the team).

These metrics should be automatically computed and updated, and made available to every stakeholder in SKA.

Note: Some of these metrics are collected and displayed by the ARGOS system in https://argos.engageska-portugal.pt (username=viewer, password=viewer). See *Monitoring Dashboards* for more information.

5 Testing strategy

A testing strategy describes how the test policy is implemented and it should help each team member to better understand what kind of test to write, how many and how to write them. This testing strategy refers to the testing policy described above.

Because of the diversity of SKA development teams and the diversity of the nature of the systems that they work upon (ranging from web-based UIs to embedded systems), it seems reasonable to start with a testing strategy that is likely to be suitable for most teams and let each team decide if a refined strategy is needed. In this case each team should explicitly define such a modified strategy, it should be compatible with this one, and it should be made public.

5.1 Key definitions and concepts

Mostly derived from the *ISTQB glossary*.

Testing levels refer to the granularity of the system-under-test (SUT):

Unit testing The testing of individual software units. In a strict sense it means testing methods or functions in such a way that it does not involve the filesystem, the network, the database. Usually these tests are fast (i.e. an execution of a test set provides feedback to the programmer in a matter of seconds, perhaps a minute or two; each test case runs for some milliseconds). Normally the unit under test is isolated from its environment.

Module testing The testing of an aggregate of classes, a package, a set of packages, a module. Sometimes this is also called "component testing", but to avoid ambiguity with the notion of component viewed as runtime entities according to the SEI "Views and Beyond" we will use "module testing".

Component testing Here the word "component" refers to deployment units, rather than software modules or other static structures. Components can be binary artefacts such as jar, DLL or wheel files, and their "containers" like threads, processes, services, Docker containers or virtual machines.

Integration testing Testing performed to expose defects in the interfaces and in the interaction between integrated components or systems. In a strict sense this level applies only to testing the interface between 2+ components; in a wider sense it means testing that covers a cluster of integrated subsystems.

System testing Testing an integrated system to verify that it meets specified requirements.

Acceptance testing Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system.

Other definitions are:

Test basis All artifacts from which the requirements of a unit, module, component or system can be inferred and the artifacts on which the test cases are based. For example, the source code; or a list of requirements; or a set of partitions of a data domain; or a set of configurations.

Confirmation testing Testing performed when handling a defect. Done before fixing it in order to replicate and characterise the failure. Done after fixing to make sure that the defect has been removed.

Regression testing Testing of a previously tested program following modification to ensure that defects have not been introduced or uncovered in unchanged areas of the software, as a result of the changes made. It is performed when the software or its environment is changed.

Exploratory testing An informal test design technique where the tester actively designs the tests as those tests are performed and uses information gained while testing to design new and better tests. It consists of simultaneous exploration of the system and checking that what it does is suitable for the intended use.

End-to-end (e2e) tests These are system, acceptance or component tests that use as observation and control points an API whose entrypoints are all at the same abstraction level, and tests exercise all or most of the components of the SUT. End-to-end tests are different than other system tests; examples of non-e2e tests are tests that control the SUT from a high level interface and observe some of its components through a lower-level, component-specific, interface for example.

6.2 Scope, roles and responsibilities

This strategy applies to all the software that is being developed within the SKA.

Each team should have at least one **tester**, who acts as the "testing conscience" within the team. Because of the specific skills that are needed to write good tests and to manage the testing process, we expect that in the coming months a tester will become a dedicated person in each team. We see "tester" as being a function within the team rather than a job role. Each team member should contribute to this function although one specific person should be held accountable for testing and join the SKA Testing Community of Practice.

In most cases, **programmers are responsible** for developing unit tests, programmers and testers together are responsible for designing and developing module and integration tests, **testers and product owners** are responsible for designing component, system and acceptance tests for user stories, enablers, and features.

System-level tests are to be designed, implemented and maintained by development teams and/or groups of testers belonging to different teams. In the latter case, testers should come from the teams that contributed one or more components of the system under test.

The **product owner of the System Team** or a **feature/enabler/capability owner** might act as a coordinator of the development.

Malfunctions in the testware implementing system-level tests should be treated as **high priority bugs**.

5.3 Test specification

Programmers adopt a TDD, BDD or test-first approach and almost all unit and module tests are developed before production code on the basis of technical specifications or intended meaning of the new code. Testers can assist programmers in defining good test cases.

In addition, when beginning to fix a bug, programmers, possibly with the tester, define one or more unit/module tests that confirm that bug. This is done prior to fixing the bug.

For unit and module testing, code coverage figures should be monitored, especially **branch/condition coverage** (as opposed to statement coverage). In particular the tester should analyse what part of the code is not being covered by tests, assess how important those fragments are, and decide if they are worth being covered. If so, new tests should be designed. If not **the team should be able to defend** why the parts not covered by tests are deemed not being important.

Importance should be assessed in terms of possible failures and their impact on the project: reduction of value of the system, difficulty in diagnosing failures, delays in deployment, damage to stored data, malfunctions to other components, impact on users.

Furthermore, the product owner with a tester and a programmer define the acceptance criteria of a user story and on this basis the tester with the assistance of programmers designs acceptance, system, and integration tests. Some of these acceptance tests are also **associated** (with tags, links or else) **to acceptance criteria** of corresponding features/enablers. All these tests are automated, possibly during the same sprint in which the user story is being developed.

During a PI, testers together with feature/enabler and capability owners define the acceptance criteria and corresponding tests (either in Gherkin or other format). These tests are defined as the development of the feature/enabler proceeds, in a test-first fashion. In this way team(s) working on the feature/enabler will be able to check when their code passes the acceptance criteria.

System-level tests are derived from acceptance criteria of epics, capabilities or features/enablers. Appropriate tags should be added to tests and scenarios so that they can be linked to epics, capabilities and features/enablers.

All acceptance criteria should be covered by one or more tests/scenarios, including happy and sad paths of the use case entailed by the criteria. Each feature/enabler should be covered by **at least one acceptance test**, possibly written in Gherkin. In general it is the Feature Owner's call to decide which acceptance tests need to be written in Gherkin. The **decision criterion** is based on who are the stakeholders of the tests: if they are developers, then Gherkin is probably not needed. If they are non-developers (eg. management, astronomers, architects) then Gherkin might be useful.

System-level tests could also be linked (for traceability purposes) to high-level requirements (L1/L2/L3, Interface requirements) of the telescope software. They could also be present in Jira. The call is on the feature/enabler/capability owner to decide when this is appropriate.

5.4 Test environment

In the Developer's portal there is a description of how test environments can be provisioned: see Testing Skampi.

There is a **staging** environment, currently used only by the Skampi codebase.

5.5 Test data

Test data, or reference to them when more appropriate, are to be stored as resource files that can be used by test cases. When applicable, test data should be versioned.

5.6 Test automation

Different projects have different needs. What follows is a non-exhaustive list of test automation frameworks and support libraries:

- for developers using Python: pytest, pytest-bdd, assertpy, mock tox
- for developers using Javascript: Jest
- for testers that have to develop end-to-end tests: Selenium
- for developers using c or c++: googletest

Developing unit/module/integration tests for Tango devices might be particularly challenging. So far, teams have devised creative ways to use mocks in Python to cope with the problem: https://confluence.skatelescope.org/display/SE/How+to+use+mocks+with+Tango.

Another important approach to write unit/module tests in a way that they don't depend on the Tango infrastructure is to apply the **Humble Object** design pattern, that is extracting the domain logic from Tango-related code and move it to a separate python object that can be tested in isolation from Tango. See some preliminary examples in https://confluence.skatelescope.org/x/MA0xBQ.

Other routes have been followed to implement system-level tests using Gherkin: https://confluence.skatelescope.org/display/SE/How+to+implement+BDD+tests.

See additional details on Pytest configuration.

In any case attention should be payed to the quality of the testware: code of the test cases, code of assertions and fixtures, for data handling, code implementing tools used for testing. Pay attention to messages in assertions and for debugging.

The important thing is that **tests should clearly convey the intention**: what do they do. Details regarding "how they work" should be hidden inside appropriate abstraction layers (auxiliary fixtures, methods, classes, variables, configurations). In this way tests can be used as a source of documentation (written in a programming language rather than plain English) which is extremely useful for integration testing. However, for complex tests, some commenting may still be useful to guide subsequent test maintainers.

5.7 Continuous integration

How the CICD pipeline is organized and how it should be used is described in https://developer.skatelescope.org/projects/skampi/en/latest/testing.html#pipeline-stages-for-testing.

It is important that teams, when configuring their own CICD pipeline, make sure that the testing stage creates and stores important artefacts related to testing. These artefacts include:

- pytest execution and coverage reports
- pytest-bdd cucumber reports
- jest xunit execution and coverage reports
- googletest xunit execution and coverage reports.

These artefacts should always be created, no matter what the outcome of the test execution is.

5.8 Confirmation and regression testing

Regression testing is performed at least every time code is committed on any branch in the source code repository. This should be ensured by the CI/DI pipeline.

In order to implement an effective CI/CD pipeline, automated test cases should be classified also (in addition to belonging to one or more test sets) in terms of their speed of execution, like "fast", "medium", "slow". In this way a programmer that wants a quick feedback (less than 1 minute) would run only the fast tests, the same programmer that is about to commit/push his/her code at the end of the day might want to run fast and medium tests and be willing to wait some 10 minutes to get feedback, and finally a programmer ready to merge a branch into master might want to run all tests, and be willing to wait half an hour or more.

Confirmation tests are run manually to confirm that a bug really exist.

5.8 Bug management

We recommend the following process for handling bugs.

- Bugs found by the team during a sprint for code developed are **fixed on the fly** during the same sprint, with no logging at all. If they cannot be fixed on the fly, soon after they are found they are logged on the team backlog.
- Bugs that are found by the team during a sprint but that are related to changes made in previous sprints, are **always logged** on the team backlog (this is useful for measuring the quality of the testing process, with a metric called defect-detection-rate).
- Bugs that are reported by third parties (eg. non SKA and SKA users, other teams, product managers) are always logged, by whoever can do it, which becomes the **bug-report owner**. These bugs have to undergo a **triage stage** to confirm that they are a bug and find the team that is most appropriate to deal with them. At that point the bugs appear in the chosen team's backlog. When resolved, appropriate comments and workflow state are updated in the team's backlog, and the original bug-report owner is notified as well, who may decide to close the bug, to keep it open, or to change it.

Logging occurs in JIRA by adding a new issue of type Bug to the product backlog and prioritized by the Product Owner in the same way other story/enabler/spike work is managed. The issue type Defect should not be used, as it is meant to indicate a deviation from SKA requirements.

For system-wide bugs the JIRA project called SKB (SKA bug tracking system) is used. Triage of these bugs is done by the SYSTEM team with support by selected people.

6 General references

Relevant "How To Pages" are:

- How to mock Tango devices
- How to test asynchronous code
- How to implement BDD tests
- Examples of unit tests - OET
- Examples of unit tests - SKABaseClasses
- On Test-Driven Development

Relevant textbooks include:

- **Managing the Testing Process**: Practical Tools and Techniques for Managing Hardware and Software Testing, R. Black, John Wiley & Sons Inc, 2009
- **Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation**, J. Humble and D. Farley, Addison-Wesley Professional, 2010
- **xUnit Test Patterns: Refactoring Test Code**, G. Meszaros, Addison-Wesley Professional, 2007
- **Test Driven Development. By Example**, Addison-Wesley Professional, K. Beck, 2002
- **Agile Testing: A Practical Guide for Testers and Agile Teams**, L. Crispin, Addison-Wesley Professional, 2008
- **Growing Object-Oriented Software Guided by Tests**, S. Freeman and N. Pryce, Addison-Wesley Professional, 2009

3.13.4 Fundamental SKA Software Standards

These standards underpin all SKA software development. The canonical copy of this information is held in eB, but the essential information is on this page, which is extracted from the eB document.

Standards Applicable to all SKA Software

1. All **SKA software** shall have a copyright notice which is a description of who asserts the copyright over the **software**.
    a. Notes:
        i. **Derived software** and **bespoke software** will normally be comprised of code modules which have a mixture of copyright attributions. Some code modules will have joint copyright, and others have sole copyright, but the codebase in its entirety will have a mixture.
2. All **SKA software** shall have a **software license** which is a legal instrument governing the use or redistribution of **software**.
    b. Notes:
        ii. **Off-the-shelf software** will normally have licenses over which the SKA has no control.
        iii. **Derived software** may have mixture of licenses.
        iv. **Bespoke software** will normally have a permissive open source license.
3. The documentation associated with **SKA software** shall also carry a license unless it is covered by the **software license**.
4. All **software licenses** governing a body of **software** must be mutually compatible.
5. All **software licenses** for **SKA software** shall be agreed with the SKA Organisation prior to the **software** being adopted or developed.
    c. Notes:
        v. The SKA Organisation will always agree to a 3 clause BSD license for **software** (provided there are no compatibility issues) and will favour open-source permissive licenses with attribution since they minimize compatibility issues.
        vi. The SKA Organisation will always agree to a Creative Commons Attribution 4.0 International License for documentation (provided there are no compatibility issues).
        vii. This permissive open source recommendation is significantly more permissive than the SKA IP policy [RD1] which only requires contributors to "grant non-exclusive, worldwide, royalty-free, perpetual, and irrevocable sub-licenses to other SKA Contributors to use those innovations and work products for SKA Project purposes only."
        viii. It is understood that the IP licensing environment of **FPGA software** is often substantially different to that of the open source software environment, with many (or most) developments relying on IP (from the FPGA vendor, for example) that has more restrictive licensing. In accordance with this standard, use of this IP, and its associated license, must be agreed with the SKA Organisation.

Standards applicable to Off-the-shelf software

All **SKA Software** which is **off-the-shelf software** shall have:

1. A business case describing the requirements for the **software**, in comparison to other **software**.
2. A record of the evidence that demonstrates that the **software** meets these requirements.

3. A description of how the **software** will be supported during the expected lifetime of the **software**.
    a. Notes:
        i. The SKA Observatory has a predicted lifetime of 50 years, which is much longer than most **software** products and the companies that develop them. Hence this description may include: how many alternatives exist which also support the **software**'s data products, escrow agreements and commercial soundness of the company. Support includes:
            1. Managing unexpected behaviour of the **software** that is incompatible with the SKA Observatory's (possibly evolving) requirements.
            2. Managing the evolution of underlying systems, such as hardware and operating systems, that the **software** depends on.
            3. Managing changes to the existing supplier support arrangements (e.g. the original company being acquired, the product becoming not commercially viable etc.).
4. Evidence that the **software** has been developed to a standard of quality appropriate to the needs of the SKA Organisation.
5. Documentation that is appropriate to the needs of the SKA Organisation.
6. Been approved by the SKA Organisation as to its fitness for purpose and included in a public register of approved **SKA Software**.

Standards Applicable to Bespoke Software

Design

This section comprises standards relating to processes described by ISO 12207 [RD1], §7.1.2 (Requirements), §7.1.3 (Architecture) and §7.1.4 (Detailed Design). They complement any general System Engineering level standards (i.e. processes relating to ISO 15288 [RD2]) applicable to all SKA systems.

All **SKA Software** that is **bespoke software** shall have documentation, models and prototypes covering the following:

1. The requirements the **software** is intended to fulfil.
2. The **software** architecture used.
    a. Notes:
        i. The **software** architecture is the primary deliverable at CDR. Detailed design is only required to the extent needed for reliable cost estimation.
        ii. The recommended reference for architecture documentation is "Documenting Software Architectures: Views and Beyond, Second Edition" (Clements et al, 2011) [RD3]. This book should be consulted for best practices on documenting views, styles and interfaces. The ISO 42010 [RD4] standard is also relevant.
        iii. The architecture documentation should include, at minimum
            1. System Overview, including a description of the architectural styles used.
            2. A set of views describing key features of the architecture, and the mapping between views.
            3. Interface Documentation or references to applicable Interface Control Documents for the major interfaces.
            4. Rationale justifying how the architecture meets the requirements. Justification on the basis of models and evolutionary prototypes is highly recommended in many cases.

5. A consideration as to whether there is any existing **software** that meets, or can be modified to meet, the requirements.

iv. Emphasis should be on clear, unambiguous diagrams with accompanying descriptions and tables.

v. Refer to Chapter 11 of Clements et al for a description of interface documentation. Interfaces that are language or framework specific may be best documented in a format appropriate to that language or framework (e.g. generated from comments and code in an evolutionary prototype).

3. Detailed design of components.

   b. Note:

      vi. It is expected that a significant amount of the detailed design may be automatically generated from code and comments.

      vii. Detailed design documentation for **FPGA software** should include estimates of device utilization (DSPs, BRAMS, LUTs etc), details of clock rates and clocking domains and tracking of timing closure issues

The **software** design should be reviewed and the reviews should incorporate the following factors:

1. The SKA Organisation is responsible for L1 requirements and must agree and review all L2 and L3 requirements.
2. The SKA Organisation personnel should be involved in **software** architecture reviews
3. The **software** architecture should be reviewed to demonstrate that it meets key requirements and provides sufficient detail for cost estimation and implementation.
4. Both the architecture and detailed design reviews shall carefully consider the requirements relating to the long lifetime of the SKA Observatory. This includes, for example:
   a. Portability of the system across multiple architectures and operating systems.
   b. Consideration of the life-cycle of all dependencies, including development tools and run-time dependencies.
   c. The need for the system to be compatible with version 6 of the Internet Protocol.
   d. The careful design of API's and the need to exchange data by API's rather than relying on environmental assumptions about file systems, for example.
5. Detailed design shall be reviewed:
   e. By someone in addition to the principal developer of the module being considered.
   f. In a manner appropriate to the significance of the module.
      i. Note:
         1. The significance of the code relates to the impact any changes to the design has on other parts of the system.
         2. The review process must not be overly bureaucratic. Development teams should be empowered to design and develop the code efficiently and modify the internal design when required.

Construction

This section comprises standards relating to processes described by ISO 12207 (2008) §7.1.5 (Construction).

The construction of all **SKA Software** which is **bespoke software** shall include:

1. The construction of all source code shall follow a defined documented process that is approved by the SKA Organisation.

a. Note:

   i. The Software Engineering Institute Personal Software Process and Team Software Process are relevant processes.

   ii. The process documentation shall include a workflow description that follows accepted best practices. For example, it is recommended that:

      1. Work management practices shall include the following:
      2. All work tasks shall be described in a ticketing system.
      3. Work tickets shall have a description of the task, an estimate of the resource required and amount of the task that has been completed.
      4. All code commits shall relate to a ticket in the ticketing system.
      5. The developing organisation shall be able to use the ticketing system to generate progress metrics.
      6. Code management practices shall include the following:
      7. With the exception of trivial cases (e.g. possibly documentation changes) code must only be added to or merged with the main development branch by a pull-request-like mechanism.
      8. The pull request (or similar mechanism) must only be accepted after the code has been cleanly compiled and passes all appropriate tests. This process should be triggered automatically.
      9. Pull requests must only be accepted after the code changes have been reviewed by more than one developer (inclusive of the primary developer).
      10. Pull requests must only accepted by suitably qualified individuals.

2. All construction **software** development shall utilise an SKA Organisation approved version control system.

   b. Note:

      iii. The SKA Organisation approved version control system is Git.

3. All documentation, source code, software source code, firmware source code, HDL source code, unit tests, build scripts, deployment scripts, testing utilities and debugging utilities must reside in the version control system.

4. Release tags for code shall adhere to the Semantic Versioning 2.0.0 specification [RD8].

5. **Software** shall be written in an SKA approved language and adhere to SKA language specific style guides.

   c. Note:

      iv. The primary approved language shall be Python.

      v. The coding standards for Python will be adapted from the LSST DM code style guides [RD7].

      vi. Use of other languages must be justified by, for example:

         3. Impossibility of running Python in the chosen run-time environment.
         4. Python doesn't provide the necessary performance.

      vii. Many other languages are likely to have extensive usage. For example:

         5. C/C++ (for high performance computation on conventional CPU's).
         6. Java (e.g. for business logic in web systems and **derived software**).
         7. VHDL (for FPGA development).
         8. CUDA (for GPU software).
         9. OpenCL (for software that targets both GPU and FPGAs)

10. JavaScript (for Web client systems).

6. SKA Organisation employees must have access to the repository while the **software** is under development, be able to sign-up for notifications of commits and, if necessary, give feedback to the developers.
7. Test **software** verifying the system **software** at multiple levels (from the complete system down to individual module unit tests). Tests shall include verifying specific requirements at different levels and, as far as practicable, be able to be run automatically.

    d. Note:

    viii. Tests shall be able to run in a continuous integration environment.

    ix. For software targeting CPU's this should include unit tests at the class, function or source file level to test basic functionality of methods (functions) with an agreed minimal coverage (between 75 and 90%). Unit tests created for fixing defects or making specific enhancements should be checked-in with a reference to the issue for which the tests were created.

    x. For **FPGA software** this should include:

    11. Each module shall be associated with a specific test bench.
    12. Modules shall undergo simulation with a predefined pass/fail criteria.
    13. Release builds shall be made up of verified functional blocks and handled in a scripted framework.
    14. Simulated and released code shall match the committed code. For example, committing the code shall not change register contents (even version numbers) in the source code.

8. **Software** simulations/stubs/drivers/mocks for all major interfaces to enable sub-system and system level tests.
9. Automated documentation generation - including, but not limited to parts of detailed design documentation.

    e. Note:

    xi. Automated documentation generation software is generally **off-the-shelf software** and so subject to the conditions in section 4.

    xii. Not all documentation can be automatically generated, but it should be used wherever it is reasonably practicable.

    xiii. The SKA Organisation shall accept ReST format documentation generated using Sphinx.

10. A complete definition of other **software** (both off-the-shelf and bespoke) that the **software** requires to build and deploy.
11. Deployment scripts or configurations, which allow the **software** to be deployed cleanly and in as automated a fashion as is practicable, starting with a bare deployment environment.

    f. Note:

    xiv. For **FPGA software**, this means configuring an un-programmed FPGA device in the target SKA system. Deployment may require the use of the host based software delivered as part of the LMC system.

12. The ability to log diagnostic information using *SKA Log Message Format*.
13. The ability, dynamically at runtime, to suppress or select logging of messages at different Syslog severity levels on at least a per-process basis (and a per-thread basis or per class basis if appropriate).
14. The ability to log diagnostics at all major interfaces at a RFC 5424 Debug severity level.
15. Alarms, where applicable, shall be based on the IEC 62682 standard [RD6].

Acceptance and handover

This section comprises standards relating to processes described by ISO 12207 [RD1], §6.4.8 (Acceptance Support), §7.1.6 (Integration) and §7.1.7 (Qualification).

SKA software which is **bespoke software** will only be accepted by the SKA Organisation after it has been appropriately integrated and validated.

1. The integration, validation and acceptance of all source code shall follow a defined documented process that is approved by the SKA Organisation.
2. This process must make clear, for all times during the handover:
    a. Who is responsible for making **software** changes.
    b. What the expected turnaround time for **software** changes is.
3. At the completion of the process all code shall have been:
    c. shown to pass appropriate, system, sub-system and unit level tests.
    d. shown to cleanly compile and/or build using an SKA Organisation provided build environment.
    e. checked into an approved SKA Organisation acceptance repository.
4. **Software** shall be integrated, as far as possible, prior to the integration of other aspects of the system.
    f. Note:
        i. During the SKA construction, this means that it is intended for this to take place in advance of the SKA Array Release schedule.
        ii. The intention is that this will be done by a series of integration "Challenges" which predate integration at an ITF, and continue through the array release period.
5. During the handover period, there shall be a 'bug fix' workflow defined that is streamlined to allow critical fixes to be deployed quickly.
6. When the SKA Organisation takes over maintenance of the **software** the complete repository, including commit history, shall be delivered to the SKA Organisation.
7. Where code requires specialised hardware for testing, provision of this hardware, or demonstrably equivalent hardware, shall be included as part of the handover.

Support Infrastructure

To develop and integrate **software** the SKA Organisation shall provide:

1. A central, globally visible, set of repositories that can be used by any SKA developers.
    a. Note:
        i. These repositories will clearly define how to handle large binary data files.
2. A globally accessible website for the storage and access of documentation.
3. A continuous integration and test framework that is open to use by developers.
    b. Note:
        ii. It is intended that this will include support for at least the 4 types of **bespoke software** described in the scope section (Tango, SDP and NIP data driven **software**, **FPGA software** and Web Applications).

iii. The development of this will be done in conjunction with the pre-construction and construction consortia. The SKA Organisation will serve as an overall coordinator.

4. Communication tools to enable **software** developers to access expertise from all the SKA **software** developer community.

   c. Note:

      iv. This will include issue tracking, discussion fora etc.

5. A list of approved **off-the-shelf software**.

   d. Note:

      v. To add **software** to this approved list, please email details of the **software**, the justification for its use, and the scope of its usage to the Head of Computing and Software at the SKA Organisation.

      vi. The intention of this approved list is to aid standardisation.

3.13.5 SKA developer community decision making process

Note: This page reports the main outcome of ADR-1 . More detailed discussion happened on confluence and jira as described below.

Everyone in the SAFe development teams is empowered and encouraged to reason about the architecture of the system, and to make decisions about it.

An **Architectural Decision Record** (ADR from now on) captures one single architectural decision, defining its scope, the rationale behind it, and its architectural relevance.

A process is defined and here described to raise, discuss, and form a consensus around a decision. The process is driven by the SKA Lead Software Architect, together with the architect of the specific subsystems. Everyone in the SAFe teams can participate in the process, and it is expected that the Architecture Community of Practice plays a major role, keeping a constant involvement. Architectural relevance

There is endless literature about what is architecture and what is not architecture in the software world. It is very hard to have a consistent definition, and it also depends on the point of view of the reader.

There are many reasons for a decision to be considered architectural in the SKA software project:

- A decision impacts several sub-systems. **All ICD changes** fall into this category.
- A decision impacts several development teams. All decisions about standardisation around common practices fall into this category.
- A decision has a significant impact on one or several quality attributes of the system. This could for instance be performance, maintainability, security etc.
- A decision deviates from the current **Software Architecture Documentation**.
- The detailed design of a sub-system can certainly be seen as an architectural activity. It is certainly encouraged to share doubts and concerns about the design of a sub-system in the form of an ADR in order to bootstrap a wider conversation.

An ADR can also be opened in the absence of a crucial decision, or a gap in the system design, where the missing decision is an impediment in the proceeding of the development activity. ADR approval process

The process works as follows:

1. **An architectural decision gets opened**

   - **Everyone** can create a Jira issue in the **ADR** project at: https://jira.skatelescope.org/projects/ADR

- At this point it should at least have a summary and a description with enough information.
- It is encouraged that ADRs contain a **proposed decision**. Basically describing a proposal of how a (sub)system shall be implemented.
- The ADR is assigned to the relevant stakeholder to manage its progress. This will usually be someone within the Architecture CoP.

2. **The decision will be analysed.** At this stage alternatives (which might be simply yes/no) and their impact on the system should be developed and analysed. This should generally involve talking to possibly impacted teams and stakeholders, looking to build consensus.
   - The decision might be **discarded** at this stage for many different reasons. This does not mean that the propsed implementation is not proceeding, just that it is not documented as an ADR.
   - A slot in the next available **Architecture synch** meeting will be used to socialise the newly created ADR and discuss the related proposals. **Everyone** is welcome to attend.
   - A new confluence page is created using the ADR template in the Solution Intent Home to record in depth analysis of the architectural issue.
3. (**The decision might escalate to an ECP.** If it turns out that the decision would have impact beyond the software systems, an ECP needs to get raised, which might in some cases block the analysis of the decision.)
4. **The decision is made.** A conclusion is selected and actions for communicating the decision are identified. From this point on the decision can be used in development.
   - The ADR Jira issue and confluence page are updated, clearly capturing the decision.
   - A slot in the next available Architecture synch meeting will be used to socialise the decision and the related impacts.
   - Outcoming actions are not to be registered as part of the decision, but they should be linked to the ADR in the form of Jira issues in other projects whenever it is possible. These might be:
     - Documents to be updated.
     - Communications to key stakeholders.
     - If **any item under configuration** control is affected by this decision an **ECP must be raised** accordingly.
     - Stories or Features in the development backlogs implementing the decision.

3.13.6 Incident Management

This section a set of guidelines for managing Incidents within the context of SKA Systems and Software.

Table of Contents

These guidelines are developed from the Manageing Incidents section of the Google SRE Book, and the GitLab GitLab Incident management Handbook . They form the basis of an Incident Response Plan, that all teams should be familar with.

Incidents come in all shapes and sizes, therefore it follows that the response mounted to remediate the associated problems will be highly variable too. The Incident Management practices outlined here are meant to serve as guidelines to developing good habits and practices around dealing with the unexpected.

Identifying an Incident

An **Incident** is defined as:

anomolous system conditions that lead to some form of service outage, unexpected system behaviour or degraded system performance that significantly impacts the delivery of correctly functioning services to the end user.

As a general rule of thumb, do not invoke a full response for something that is easily recognisable as taking 30 minutes to solve, and has limited impact. However, if it will take longer to solve and/or has significant impact on your user community such as Nexus, Confluence or Jira then consider a more focused response. It is better to declare an Incident and then find a simple fix for it than for something that appears minor at first glance to spiral out of control into a large problem due to things quickly unravelling. The bottom line is to exercise common sense, and be willing to postmortem the process with external stakeholers to evaluate the problem and the process in the pursuit of relentless improvment.

Incident Workflow

The following sections describe a framework for working through an incident, the roles and duties of key personnel involved, and the tools used for managing effective communications throughout.

Responding effectively to these kinds of issues is key to restoring degraded or lost service, and learning from the experience so that repeat occurances are eliminated or minimised.

Central to responding effectively is not just having a clearly defined plan for how to respond in emergency, but also rehersing and putting the plan into action.

These guidelines are written using Systems Team examples, however they are generally applicable and should be used by all teams within the SKA Project to when formulating and practicing their own Incident Response. The overall incident workflow could be summarised in steps like below:

- Have a clear channel of communication with your user community
- Identify the problem and assemble the team and resources
- Incident Response
- Incident Reports & the Postmortem

Have a clear channel of communication with your user community

For all the services within your teams responsibility, have a well understood channel of communication for your users to notify them of problems with these services. Triage reported issues, and put non-critical items that are cross-team related in the backlog (ensuring your user community knows this), or in your own teams project space (System Team), and identify those that are an Incident. A common and well known communication channel for your user community means that they know how to get in touch with you, but also crucially they know how to find out about ongoing issues and service status. This will help manage the flow and decemination of information as an Incident unfolds, and can save large amounts of time and confusion during the response through delivering a consistent message about the current state and the efforts to resolve it.

Use this channel of communication proactively. Make sure that when an Incident has been identified, notifiy the affected user groups immediately with information about what the Incident is, when it occurred, what (if known) is being done about it, and any indication on how long or when the Incident will be resolved. Proactive communication will save time in dealing with individual contact, and improve relations and confidence within your user community. It is likely that if you have an engaged community, then they will be helping by letting you know when there are issues, so foster that feedback culture.

Communication should be based on the most convenient and readily monitored solution (email, phone, web page, slack, Zoom, carrier pidgeon, etc.) used by the user community. The Systems Team uses Slack for most communication with the following channels:

- The #team-system-support channel for System Team specific services such as Nexus, ElasticStack, Prometheus, GitLab Runners, Kubernetes and OpenStack
- If the issue impacts on teams ability to do integration testing then it is #proj-mvp
- For major outages that are likely to have impact project wide then consider making announcements in multiple related channels to capture all the affected user communities.

It may be necessary to use multiple forms of communication, and/or channels. What spread of these channels is required will be part of the ongoing impact assessment as the Incident unfolds. It is also important to gather all the incident related to communication in one place so that it is shared and discussed from a central location. This may be a temporary slack channel linked to JIRA ticket as long as it is shared with other ways of communications to give/take updates on the incident.

Identify the problem and assemble the team and resources

Identifying problems and how to handle them is largely dependent on the initial impact assessment. Impact assessment is something that must be reevaluated at intervals (for instance, hourly) as the Incident unfolds and knowledge builds around the roadmap to resolution.

Agree a time limit on the initial response. If it can be solved in 30 minutes by the subject matter expert with minimal impact then timebox this and strictly reevaluate the situation at the end of that time allocation.

In all other case (including the elapsed initial response) be prepared to gear up a full Incident response and get the process in motion.

Once the problem/s is identified, then line up the team and resources that will mount the response. It is crucial that no one other than the assigned resources work on the resolution - there must be no conflict or miscommunication about what is happening, or what steps are being carried out to resolve the Incident - there is to be no “Freelancing” on the problem.

Incident Response

- Tackle the immediate problem/s i.e. restore service whether it is fixing the failing system or enacting the recovery/fail over plan. Preserve the evidence of any system components involved in order to support the Postmortem.
- Apply separation of responsibilities and actions - having clearly defined roles and responsibilities during an Incident limits the chances of confusion as to who is doing what and frees the Ops lead (the technical or expert lead) up to concentrate on creating and delivering the solution for resolving the Incident issues.

- Command - a clear structure of authority and responsibility means that the Ops Team (the subject matter experts) can focus on solving the problems, and freeing them from managing logistical issues.
- Operations work - the Ops Team are solely responsible for making changes to the system to correct the situation. This includes ensuring any other conflicting system support activities are halted as coordinated by the Incident Commander.
- Live Incident State Document - Create a Jira ticket that is periodically (half hourly) updated with a distilation of how the Incident unfolds from identifaction to resolution. The ticket is closed out only when the Postmortem is concluded linking to the Incident Postmortem Confluence document.
- Communication - after the initial Incident appraisal, breakout a new slack channel (and link to it as appropriate) to compartmentalise the conversation around the response, and to create a chronological record of how the Incident unfolds from identification through to resolution. Cross link to any other communication channels to create as complete a picture as possible of what is happening. This also forms a virtual Incident Command Post for the Incident response team to gather round. This can be augmented with a dedicated Zoom Room published in the channel, but it must be backed up with written in channel notes that track decisions, and discoveries.
- Planning - track the changes made to the system, triage what are filed as bug reports, and what needs to be unwound (and how) during post-incident clean up.
- Clear, Live Handoff - long running Incidents may require resolution over multiple shifts and days. There must be a coherrent handing over of at shift boundaries
- Cleanup - organise the removal of temporary measures, and reverting services and system management to standard operational practices. Ensure that all relevent data is preserved and offloaded to safe storage.
- Postmortem - when the Incident is over, perform a Postmortem to understand what went wrong and what corrective measures should be put in place to reduce/eliminate further occurances.

Roles

The roles separate the responsibilities for the mounted response to the Incident. The roles are inherently scalable, meaning that an individual may fulfil more than one role (the Commander, Comms, and Planning roles are typically combined), and most roles can be inhabited by more than one person, however the Ops Lead role must always be separated from the remaining roles as it is critical that the Ops Lead is freed from all other responsibilities (and distractions) in order to concentrate on solving the problems at hand and it is coordinating a team across disciplines, offices and timezones to mount an effective response.

All people assigned to the Incident treat the response as the highest priority task in their schedule.

- Incident Commander - is the person in charge of the response to an incident, with the responsibility and authority for organising the response team, and other resources (even hardware) and directing the high level strategy. The Incident Commander organises the resourcing of other team members, assigning roles and handles the liason between the Ops team members and other internal or external stake holders. The Incident Commander ensures that there is complete separation of responsibilities so that there is no risk of overlap or confusion around inflight tasks. The Incident Commander is the sole maintainer of the Live Incident State Document and is the authority on the current state of the operation.
- Ops Lead - is the lead technical or subject matter expert evaluating the Incident, diagnosing the issues, formulating the response. Ops in this sense, is Operation Support for the system/solution/environment context where the Incident is taking place.
- Ops Team Member and/or Subject Matter Experts (sourced from other teams where necessary) provides support for the Ops Lead and follows the Ops Leads direction for working through the problem resolution.
- Comms - formulates and executes the plan for communication of the Incident and response to the affected user community, and the public message if required.

- Planner - supports Ops by tracking changes being made (system divergence that can evolve due to emergency action), filing bug reports, plotting the path for any system state cleanup required.
- Postmortem Team - stake holders including user community representatives, and the Incident Response Team.

Supporting Resources

Throughout the Incident Response, there maybe additional resources required ranging from Subject Matter Experts to Hardware, coffee and pizza. Ensure that there are contact details, processes and procedures in place to source these in advance.

Preparation, Planning and Practice

Relative to your team, the Incident Response Plan is only as effective as it is workable and relevant. The operation of the plan needs to be tested, and your team needs to practice fulfilling the various roles in the plan so that when it needs to swing into action all players understand what needs to happen and can purely focus on the problem at hand.

Incident Reports & the Postmortem

When the Incident is over, gather the Postmortem Team and follow the SKA Post Incident Postmortem review process. To capture the relevant stakeholders and information fresh before they disperse it is important to finish the postmortem as soon as possible so the incident is served out of the oven.

3.14 Definition of Done

The definition of done is used to guide teams in planning and estimating the size of stories and features:

- *Definition of Done*

3.15 Fundamental SKA Software & Hardware Description Language Standards

These standards underpin all SKA software development. The canonical copy is held in eB, but the essential information is here:

- *Fundamental SKA Software Standards*

3.16 Incident Management

The incident management workflow is used to guide teams in dealing with anomolous conditions that lead to some form of service outage, unexpected system behaviour or degraded system performance:

- *Incident Management*

3.16.1 System Design

- Continuous Delivery describes system design and practices necessary to realize continuos delivery.
- Design Patterns: Elements of Reusable Object-Oriented Software describes the most common design patterns to be found in a software system.

3.16.2 Programming

- Code complete is a practical introduction to software craftmanship.
- The pragmatic programmer is a good introduction to sound programming practices.
- Clean code introduces quality software practices showcasing different examples and good principles from the agile world.
- Extreme programming explained can be extremely useful to teams and developers embracing a more agile way of working for the first time.

3.16.3 Programming Languages

Python

- Python in a Nutshell is a comprehensive Python reference to have on your desk while developing any sort of Python application.
- Python testing with pytest covers pytest framework and related testing best practices in the Python ecosystem.
- Fluent Python is a useful guide to writing effective, idiomatic Python code. This book is recommended reading for intermediate Python developers and those coming from other languages, showing how Python features and idioms should be used most effectively.

C++

- Effective C++ contains useful recipes for sound implementations of common C++ patterns.
- Effective STL focuses on the correct usage of the standard teamplate library.
- Effective Modern C++ follows from the previous series, expanding on the transition to C++11 and C++14 and their new constructs.

3.16.4 The SKA Organisation

Coming soon. . .

3.16.5 SAFe for SKA

Coming soon.

3.16.6 Follow us on social media

- Facebook https://facebook.com/SKAtelescope
- Twitter https://twitter.com/SKA_telescope
- Instagram https://instagram.com/ska_telescope

3.16.7 Agile teams and responsibilities

SKA software development is organized in agile teams.

Development team

See https://www.scaledagileframework.com/agile-teams/

Todo:

- should we expand this section? The whole portal is dedicated to describe DEV practices and tools . . .

Scrum Master

The Scrum Master of each team is responsible for the process the team follows. A generic description of this role can be found on the SAFe website. The SKA Scrum Masters are also responsible for:

- Meet the team, make sure they know each other and find a nice way to present interests, skills and get to know each other. Lead the team to find a name they like.
- Make sure all team members can access SKA confluence and jira.
- Make sure all team members have access to SKA video conferencing tools.
- Create a team page on the SKA confluence portal describing who belongs to the team and his/her role. This page will serve as an entry point for team related information.
- Use the Team Jira board to plan and report team activity happening in the development sprints.
- Run sprints planning/retrospective/reviews cycles and daily stand-up meetings with the team, making sure the team follows an improvement process.
- Work with the team in order to understand the SKA Definition of Done and development practices.
- Setup and maintain a slack channel for the team according to the slack usage guidelines.
- Setup and maintain a gitlab team including all team members under the SKA organization gitlab account.
- Manage permissions on gitlab repositories the team is working on.
- Maintain consistency between the team composition on the various tools and platforms, and make sure that team members are using those in an appropriate way.
- Take part in the Scrum Of Scrums meeting, coordinating his/her activity with all the SMs participating in the development effort.

Product Owner

See https://www.scaledagileframework.com/product-owner/

Todo:

- Define specifics activities for SKA POs.

CHAPTER 4

About the SKA

For information about the SKA, have a look at this section.

- *The SKA Organisation*
- *SAFe for SKA*
- *Follow us on social media*
- *Agile teams and responsibilities*
- *Follow us on social media*

4.1 Contribute to the developer portal

4.1.1 Sphinx documentation

Quick start

Direct Contribution

You can use the *Edit on GitLab* (TODO: need to add a picture!) button to directly contribute to a page. After logging in, this will open GitLab's edit page window in which depending on your permissions:

- could use GitLab's Edit Window to make your changes and push to a branch, then create a Merge Request
- or use fork option(TODO: need to add a picture!) and then follow the same steps: use GitLab's Edit Window to make your changes and push to a branch, then create a Merge Request from your fork to the developer portal project. Note; creation of the fork will be handled automatically.

Then, you can follow the Merge Request page for status updates, make new contributions directly or by setting up your local development as described below.

Static build

First install dependencies:

` pipenv install --dev `

Build the documentation locally:

` pipenv run make html `

This will create a subdirectory */build/html*. To browse the documents created open */build/html/index.html* in a web browser.

CHAPTER 5

Contribute to the SKA Developer Portal

We encourage all members of the development community to submit improvements to the Developer Portal. These pages describe how you can contribute changes to the Developer Portal.

- *Contribute to the developer portal*