developer.skatelescope.org Documentation

Release 0.1.0-beta

Marco Bartolini

Feb 05, 2020

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 in this portal.

CHAPTER 1

Scope

This documentation applies to the bridging phase of the SKA project, further updates and scope changes will be published on this website. Part of the bridging phase goals will be to consolidate and enrich this portal with more detailed information. It is thus anticipated that in this phase the change rate of the documentation will be very frequent.

CHAPTER 2

SKA developer community

SKA software development is managed in an open and transparent way.

2.1 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

2.2 Howto join the SKA developer community

Todo:

- to be defined as soon as we have onboarding procedures for new members, and specific training events.

2.3 Agile teams and responsibilities

SKA software development is organized in agile teams.

2.3.1 Development team

See https://www.scaledagileframework.com/dev-team/

Todo:

- should we expand this section? The whole portal is dedicated to describe DEV practices and tools ...

2.3.2 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 github team including all team members under the SKA organization github account.
- Manage permissions on github 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.

2.3.3 Product Owner

See https://www.scaledagileframework.com/product-owner/

Todo:

- Define specifics activities for SKA POs.

2.4 SKA developer community decision making process

Todo:

- this is still TBD with the program management team.

- *SKA Code of Conduct*
- *Howto join the SKA developer community*
- *Agile teams and responsibilities*
- *SKA developer community decision making process*

Todo:

- SAFe process implementation
- Community forum

2.5 Working with git

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 Committing code

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

- Commit often.
- Have the **Jira story ID** at the beginning of your commit messages.
- Git logs shall be human readable in sequence, describing the development activity.
- Use imperative forms in the commit message.

2.5.3 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"
$ 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"
$ git config user.email
your@institutional.email
```

2.5.4 Branching policy

Albeit the SKA organisation does not want to be prescriptive about git workflows, two concepts are important to the SKA way of using GIT:

1. The master branch of a repository shall always be stable.
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.

Master based development

We suggest teams to start developing adopting a master-based development approach, where each developer commits code into the master branch at least daily. 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 his 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 his 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 our master branch with the click of a button.

Story based branching

We support adopting a story-based branching model, often referred to as **feature branching**. This workflow effectively leverages **pull 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 some piece of work a developer should conclude in the time of a sprint, the workflow would follow these steps:

- As a developer starts working from master on a new story, she creates a new branch.
- The new branch shall be named as the story, i.e. *story-AT1-26*.

```
$ git branch
* master
$ git checkout -b my-story-id
$ git branch
master
* my-story-id
```

- 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 her local story branch.
- When the story is ready for acceptance the developer pushes the story branch upstream.

```
$ git push -u origin my-story-id
```

- A pull request is created on the DVCS server to merge the story branch into the master branch.
- Reviewers interact with comments on the pull request until all conflicts are resolved and reviewers accept the pull request.
- Pull request is merged into Master.
- The CI pipeline is executed successfully on the master branch by the CI server.

Whenever a team deviates from one of the recommended policy, 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://guides.github.com/introduction/flow/

2.6 Working with GitLab

2.6.1 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 shall 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* .

2.6.2 Setup SSH key

Associate your ssh-key to your user at *Settings->SSH keys*.

2.6.3 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.

2.7 Working with Github (Deprecated)

2.7.1 Use institutional email

Create a github account using your **institutional email** address at https://github.com/join?source=login . If you already have an account on github, you shall 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* .

2.7.2 Setup SSH key

Associate your ssh-key to your user at *Settings->SSH and GPG keys* .

2.7.3 Join SKA Organisation

SKA Organisation can be found on github at https://github.com/ska-telescope/ , The scrum master of your team will make sure you can access it.

2.7.4 Desktop client

Less experienced developers can use the github desktop client at: https://desktop.github.com/ This definitely lowers the barrier of using git for a number of different users.

2.8 Transition your GitHub Repo to GitLab

The SKA Software team decided to move from GitHub to the GitLab platform as the main git-repository manager for its CI/CD tools. This page is a simple walk-through of all the steps you have to do for this switch.

2.8.1 Create GitLab Account

The fist step is the more obvious and the simplest one. You just have to go to https://gitlab.com/users/sign_in in your favorite browser and create a new account. For that, you have two main options:

1. Create an account with an email of your choice;
2. Sign in with GitHub Credentials to automatically have the same account information on the new profile. (Recommended)

2.8.2 Developers

In the local development you must update the git files of each project to the new GitLab repository. Therefore, you only have to write this in the terminal inside the project directory:

```
$ git remote set-url origin https://gitlab.com/ska-telescope/your-project-name.git
```

To check if everything went smoothly, type the command below and check if the both links are the same as the GitLab repo link that you just typed.

```
$ git remote -v
```

And it's done! You can now do everything that you are authorized to do on this GitLab repository.

2.8.3 Project Owners

Create GitLab Project

For the moment the only users on our GitLab that can create a repository, are the System Team members - please ask on the #team-system-support Slack channel. For more information on creating new projects, go to *this page*.

Mirror Repository

It was generally accepted that we still use Github to show our open-source projects because we have more visibility there. So, to do that we have to setup a mirroring process from Gitlab to Github.

The first step is to create a personal access token on Github. For this, just open their settings page and select "Developer Settings" option from de side bar menu (direct link: https://github.com/settings/tokens). From there click on 'Personal Access Tokens", press 'Generate new token' and fill the token's name and check the 'public_repo' option. Just like the picture below.

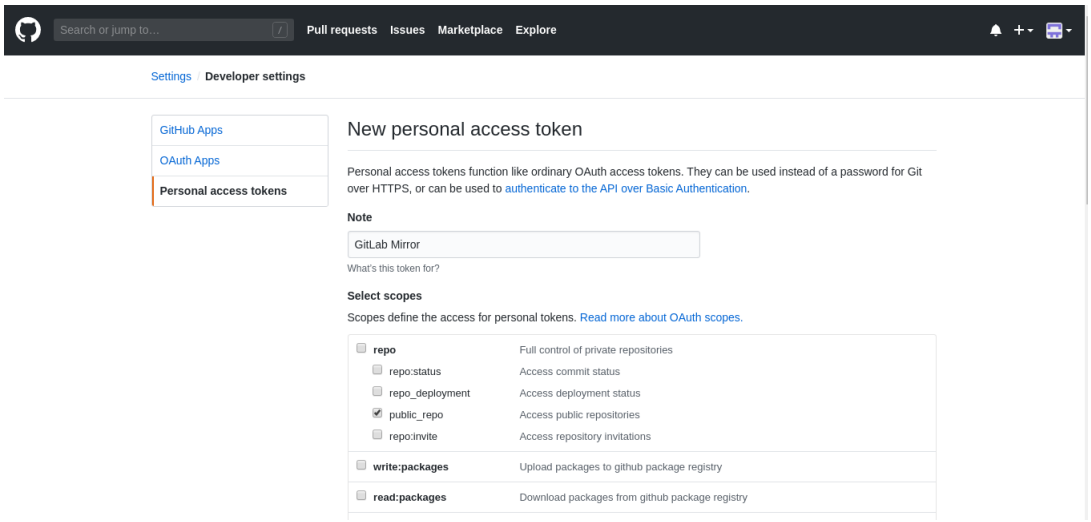


Fig. 1: Get GitHub Personal Token.

After you generate the token, go to your project settings page on Gitlab, click on 'Repository' section and fill the text boxes in the 'Mirroring repositories' like this:

- Git repository URL: https://<your_github_username>@github.com/ska-telescope/<your_github_project>.git.
- Mirror direction: Push
- Authentication method: Password
- Password: *Github_Password*

Finally, with the mirroring process finalized and with no errors, you can start pushing and use the available dashboard to check if everything is working properly.

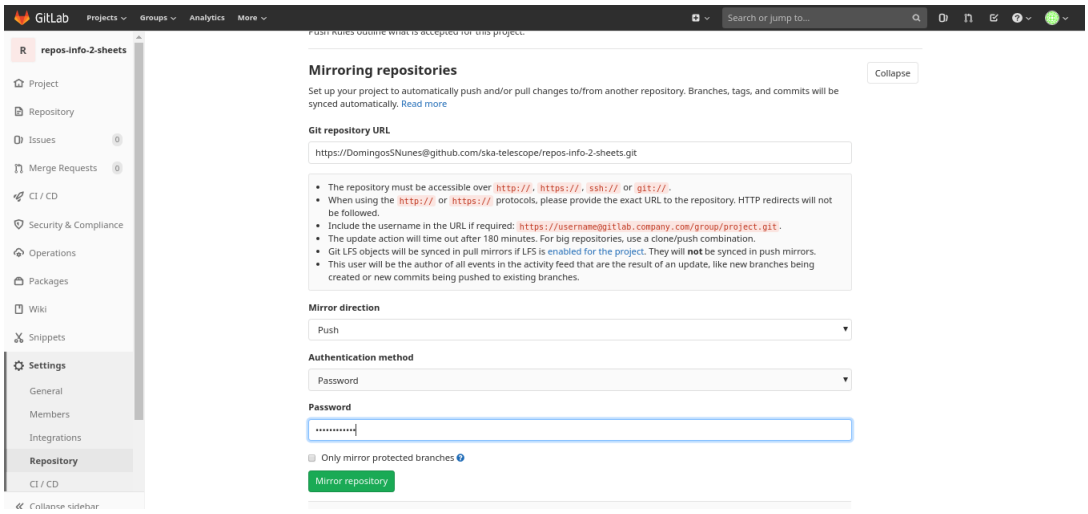


Fig. 2: The GitLab mirror setup.

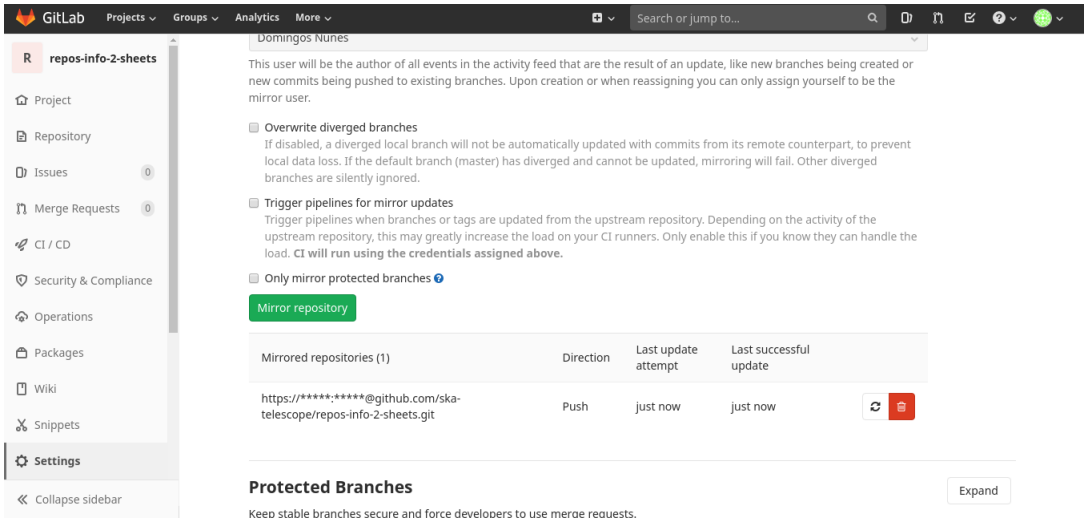


Fig. 3: GitLab Mirror Dashboard.

Read-Only GitHub Project

Differences Between GitHub and GitLab

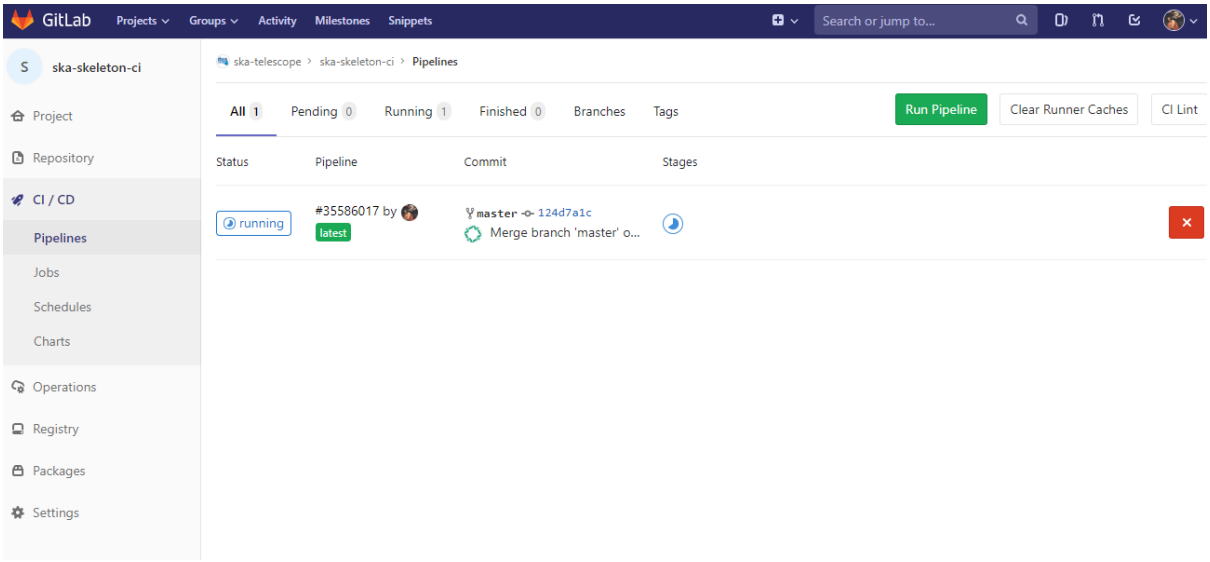
Since both GitHub and GitLab are built on top of Git, there are very few differences between the two systems. The first obvious difference is that GitLab has merge requests instead of pull requests. The function is pretty much identical, and the UI is pretty similar. GitLab provide an extensive tutorial on merge requests.

The other major difference is that GitLab provides automatic Continuous Integration Pipelines. If you have already used Jenkins, you'll find it pretty similar. There is an SKA guide to *Continuous Integration*.

2.9 Continuous Integration

2.9.1 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 github as soon as the mirroring finishes. The following pipeline was created manually pressing the button "Run pipeline" on a specific branch (i.e. master).



2.9.2 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">
```

In order to automate the process as much as possible for the teams, the *ci-metrics-utilities* 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 following code must be added at the end of `.gitlab-ci.yml`

```
create ci metrics:
  stage: .post
  image: nexus.engageska-portugal.pt/ska-docker/ska-python-buildenv:latest
  when: always
  tags:
    - docker-executor
  script:
    # Gitlab CI badges creation: START
    - apt-get -y update
    - apt-get install -y curl --no-install-recommends
    - curl -s https://gitlab.com/ska-telescope/ci-metrics-utilities/raw/master/
↪scripts/ci-badges-func.sh | sh
    # Gitlab CI badges creation: END
  artifacts:
    paths:
      - ./build
```

2.9.3 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*

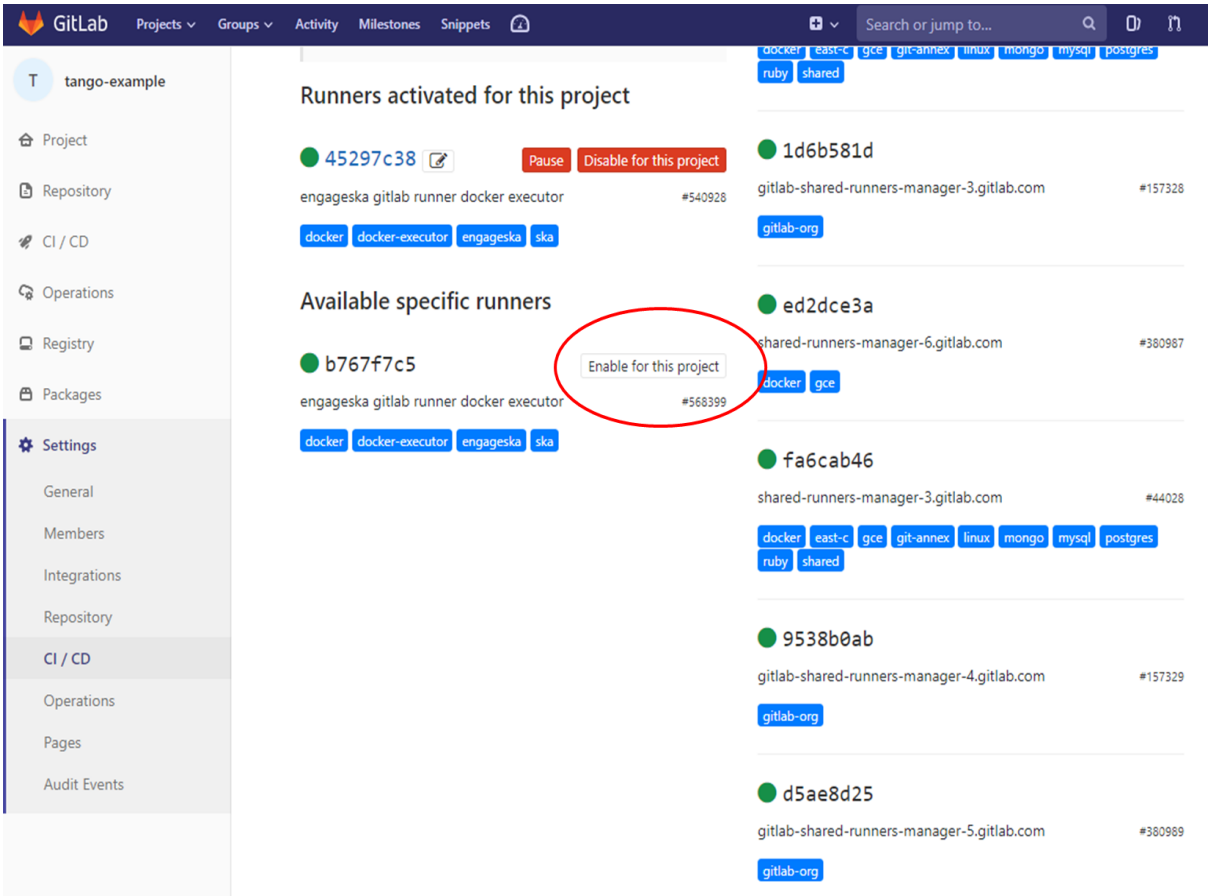
- **build-status**: *top level placeholder for the build process status*
  - **last**: *placeholder about the last build process*
    - **timestamp** (float): *the Unix timestamp with the date and time of the last build status*
- **coverage**: *placeholder about the unit test coverage*
  - **percentage** (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
  }
}
```

2.9.4 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 runner 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.

2.9.5 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 Release Procedure.

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.

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 here. 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.

2.9.6 Using environment variables in the CI pipeline to upload to Nexus

There are several environment variables available in the CI pipeline that should be used when uploading Python packages and Docker images to Nexus. This will make these packages available to the rest of the SKA project.

Python Modules

The Nexus PYPI destination as well as a username and password is avialable. For a reference implementation see the lmc-base-classes .gitlab-ci.yaml

Note the following:

- The Nexus 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: $TWINE_USERNAME
    TWINE_PASSWORD: $TWINE_PASSWORD
  script:
    # check metadata requirements
    - scripts/validate-metadata.sh
    - pip install twine
    - twine upload --repository-url $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 Nexus Docker registery 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 *$DOCKER_REGISTRY_FOLDER* is used.

```
script:
- cd docker/tango/tango-cpp
- make DOCKER_BUILD_ARGS="--no-cache" DOCKER_REGISTRY_USER=$DOCKER_REGISTRY_FOLDER␣
↪DOCKER_REGISTRY_HOST=$DOCKER_REGISTRY_HOST build
- make DOCKER_REGISTRY_USER=$DOCKER_REGISTRY_FOLDER DOCKER_REGISTRY_HOST=$DOCKER_
↪REGISTRY_HOST push
```

2.10 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

2.11 Tango Development Environment set up

2.11.1 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.

2.11.2 Prerequisites

- VirtualBox installed
- git

2.11.3 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"
sudo reboot
```

2.11.4 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
```

2.11.5 Other information

Please visit the following github pages for more information:

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

2.12 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.

2.12.1 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:

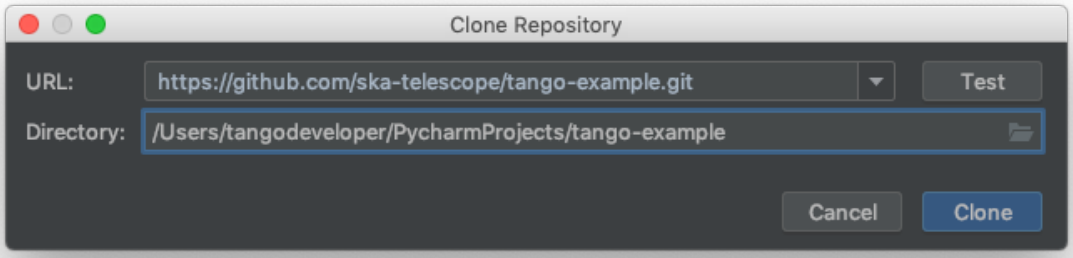
- Docker is installed, as described on the page Docker Docs.
- PyCharm Professional must be installed. *PyCharm Community is not sufficient!*

- 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:



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)

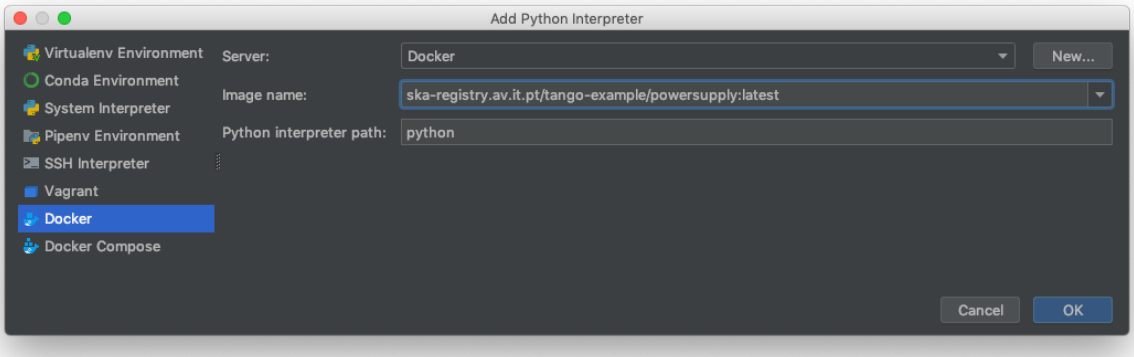
(continued from previous page)

```
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
```

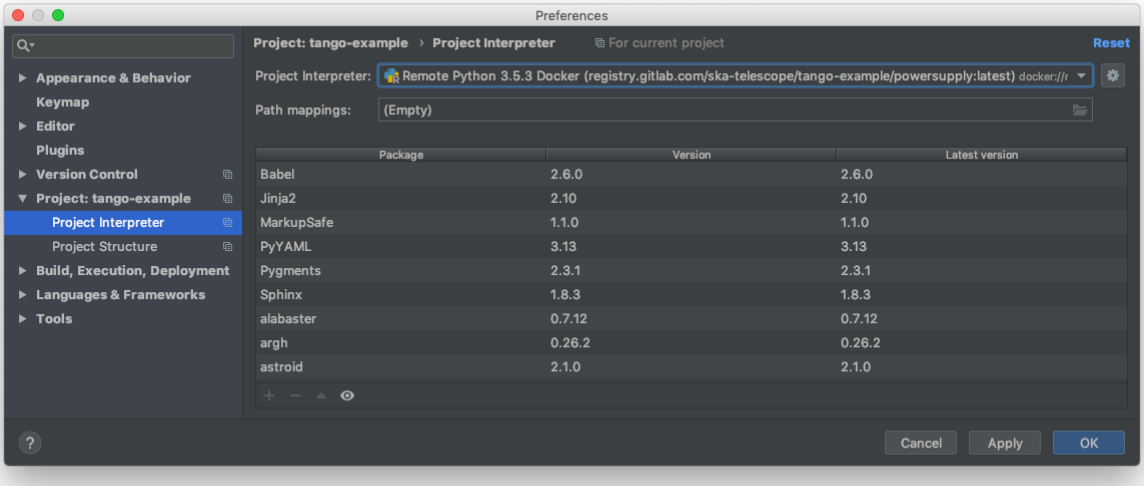
In the example above, the image name is tagged as *nexus.engageska-portugal.pt/tango-example/powersupply:0.1.0-65c0927* and *nexus.engageska-portugal.pt/tango-example/powersupply:latest*. Take a note of the image tagged as *latest* as you will need it when configuring the remote interpreter.

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:



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



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.

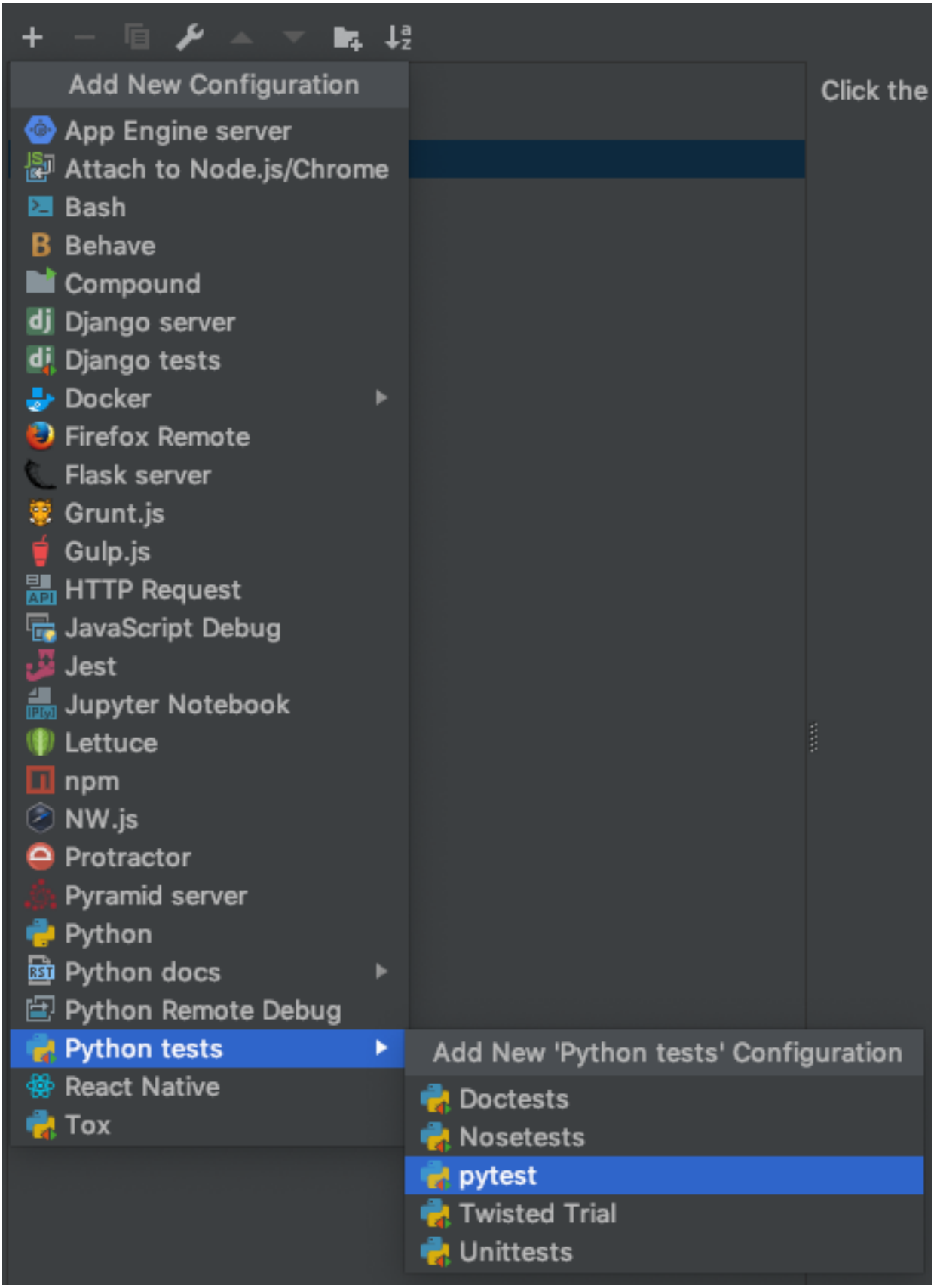
Running unit tests

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.

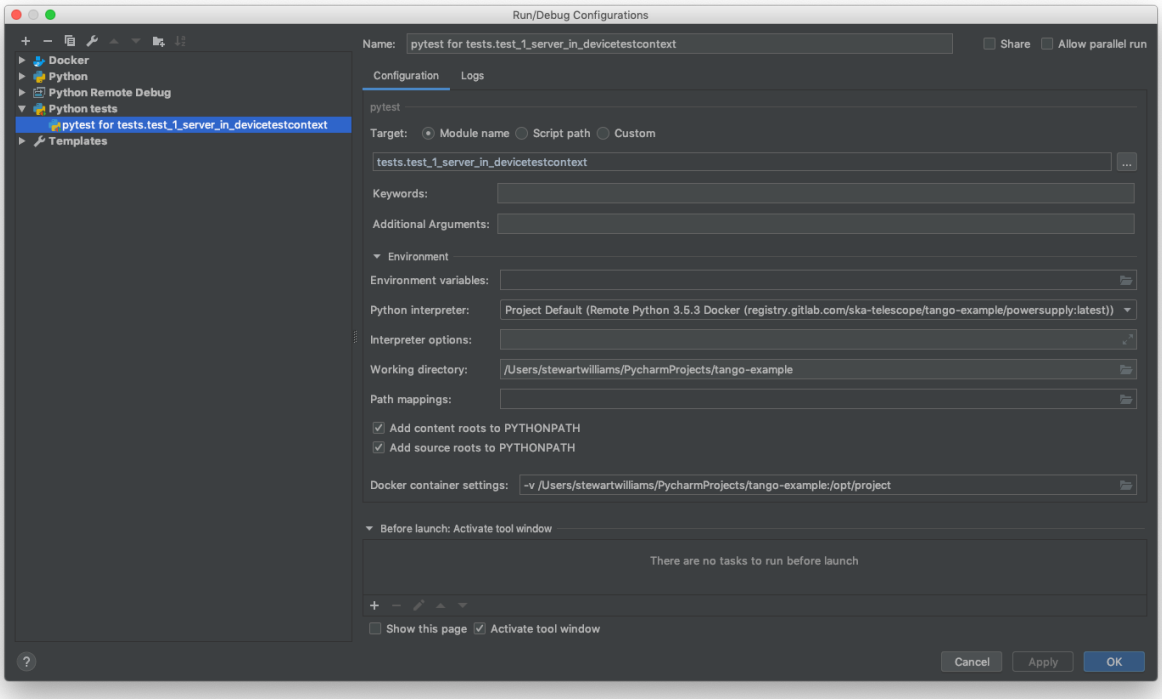
From the main menu, choose Run | Edit Configurations... and click on the '+' button to add a new configuration. From the menu that appears, select Python tests | pytest to add a new pytest test configuration. The menu selection looks like this:


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.

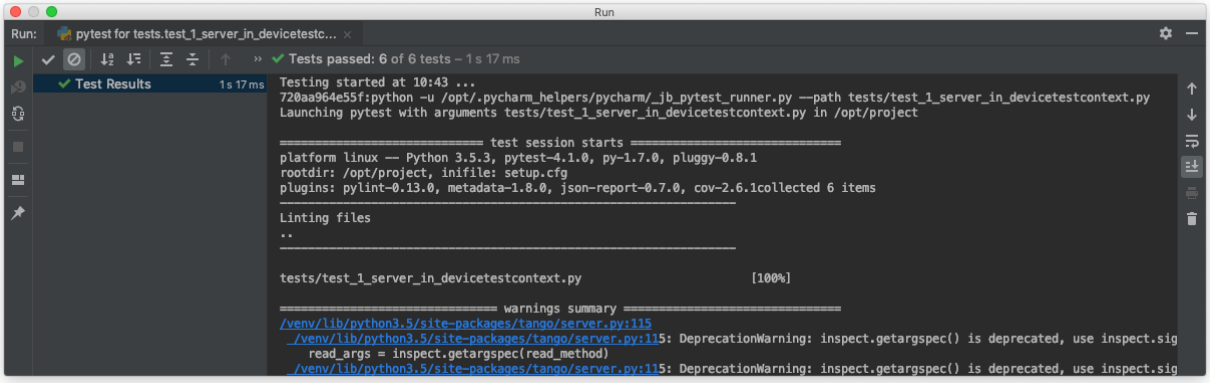
Note: If you change the project default interpreter to another configuration - a Docker Compose configuration, for instance - then you may want to revisit this run/debug configuration and explicitly select the Docker *:latest* interpreter rather than use the project default.

The configuration dialog should look like similar to this:



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:



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://github.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.

2.12.2 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
 ---> Using cache
 ---> Using cache
```

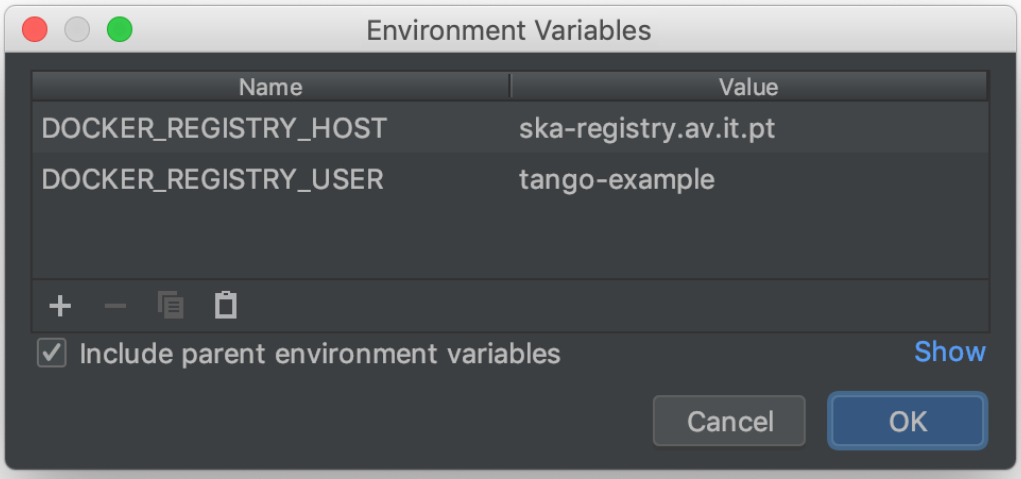
(continues on next page)

(continued from previous page)

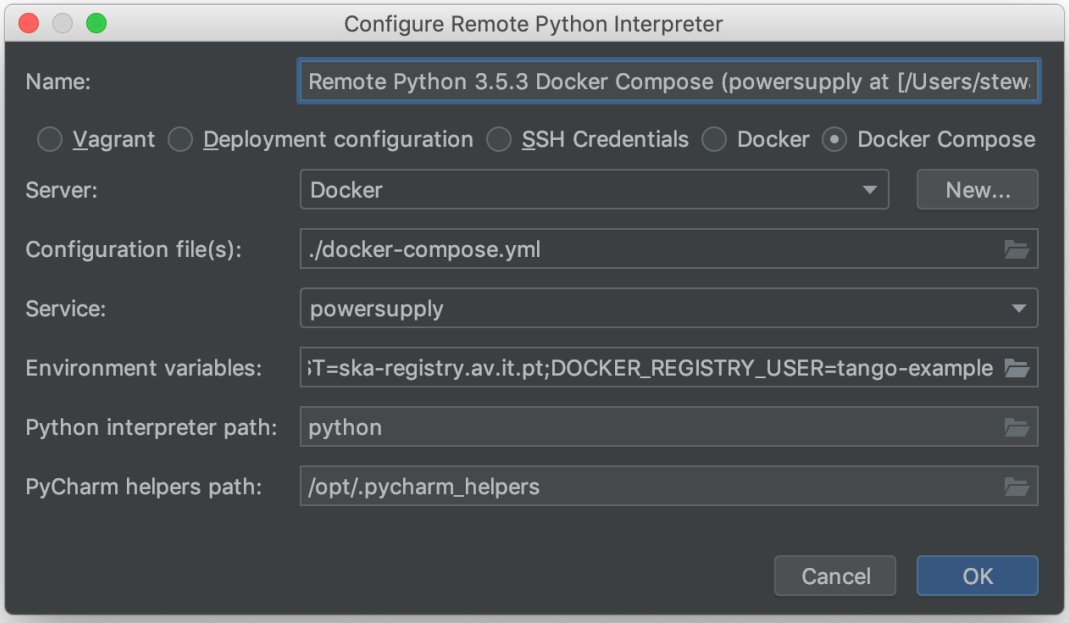
```
 ---> Using cache
 ---> Using cache
 ---> 1d24e7c0f8aa
Step 3/4 : RUN ipython profile create
 ---> Using cache
 ---> 93c8f22c5f87
Step 4/4 : CMD ["/venv/bin/python", "/app/powersupply/powersupply.py"]
 ---> Using cache
 ---> b54df79f52d6
[Warning] One or more build-args [DOCKER_REGISTRY_HOST DOCKER_REGISTRY_USER] 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:



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



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 intel-

lisense / code completion are now based on information gathered from the remote Docker image.

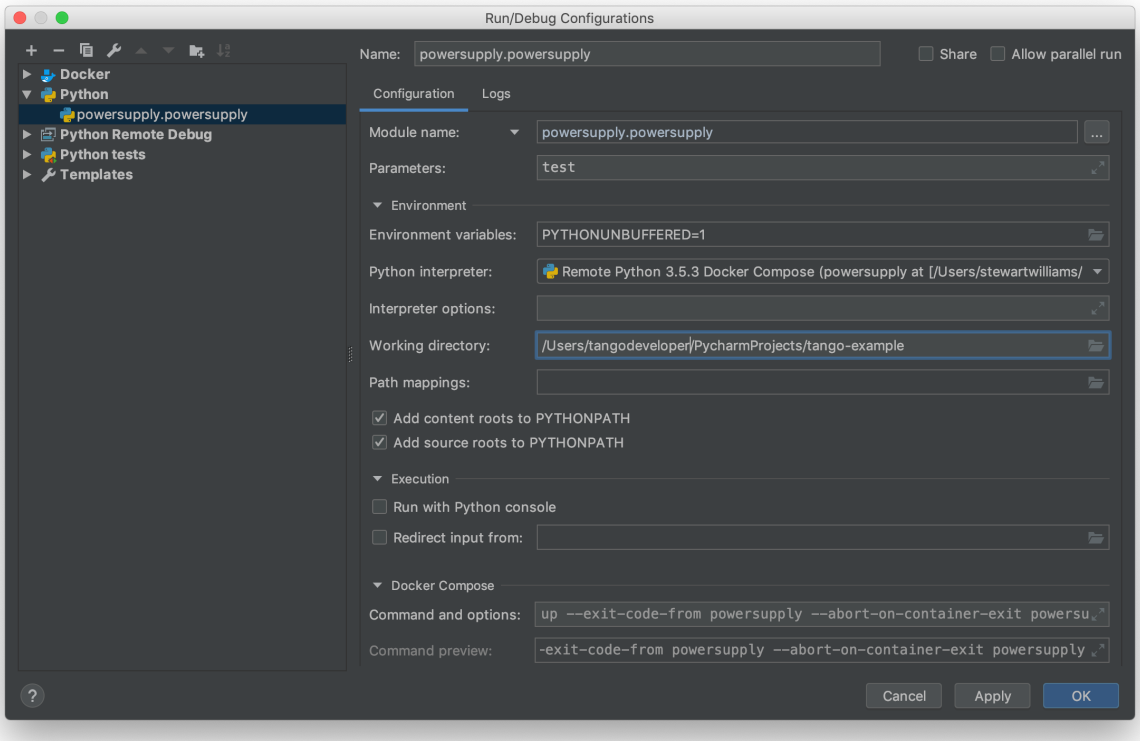
Just as for the Docker configuration, whenever you change the Python environment you should regenerate the project skeletons to make PyCharm aware of the changes. To do this, select File | Invalidate Caches / Restart... from the main menu.

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:

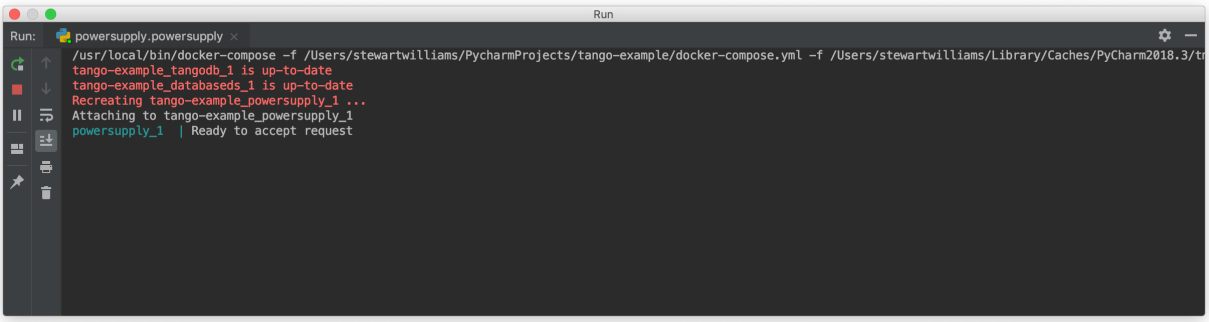
1. Edit the script/module to be executed to point to the `powersupply.powersupply` module.
2. Add `test` as an execution parameter; this tells the PowerSupply device to execute as the PowerSupply/test instance we registered earlier.
3. Change the working directory to the root of the project.

The final Run/Debug dialog should look like this:



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.



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
```

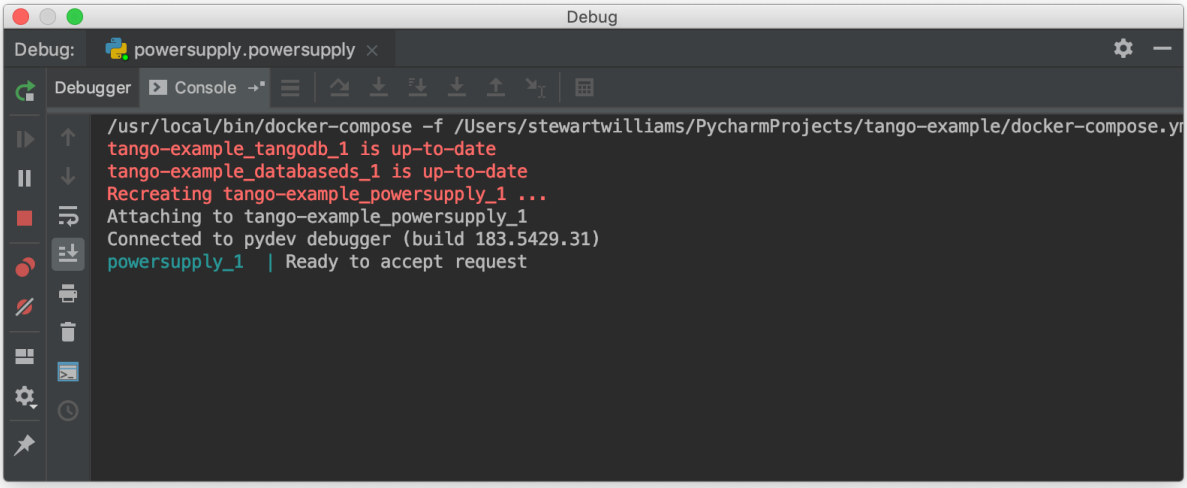
Debugging unit tests

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:



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)

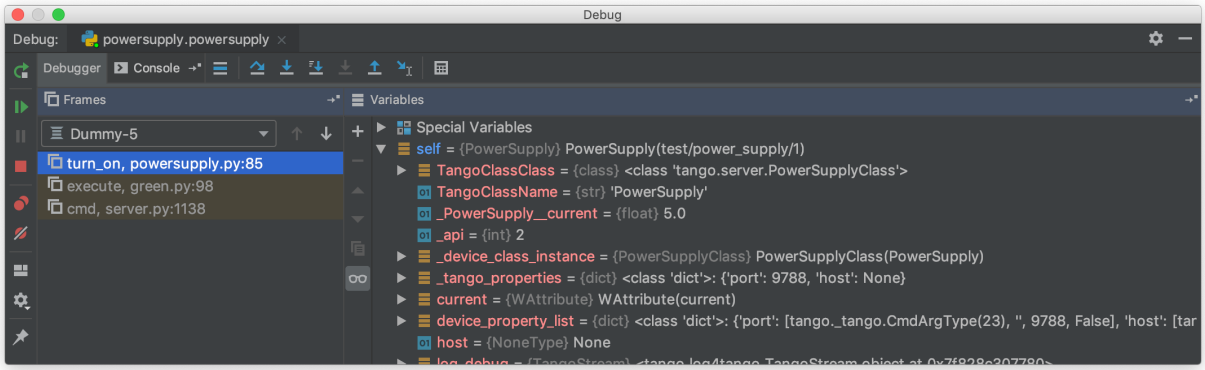
(continued from previous page)

```
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)
```

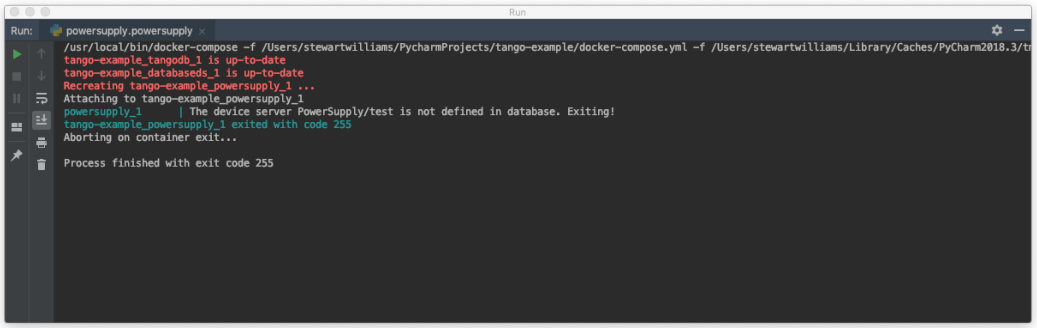
Restart the debugging configuration for the code change to take effect and re-execute the test in the interactive session. This time, the breakpoint is respected and execution is frozen, allowing program state to be examined in PyCharm. The debug panel in PyCharm will look something like this, showing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



Troubleshooting

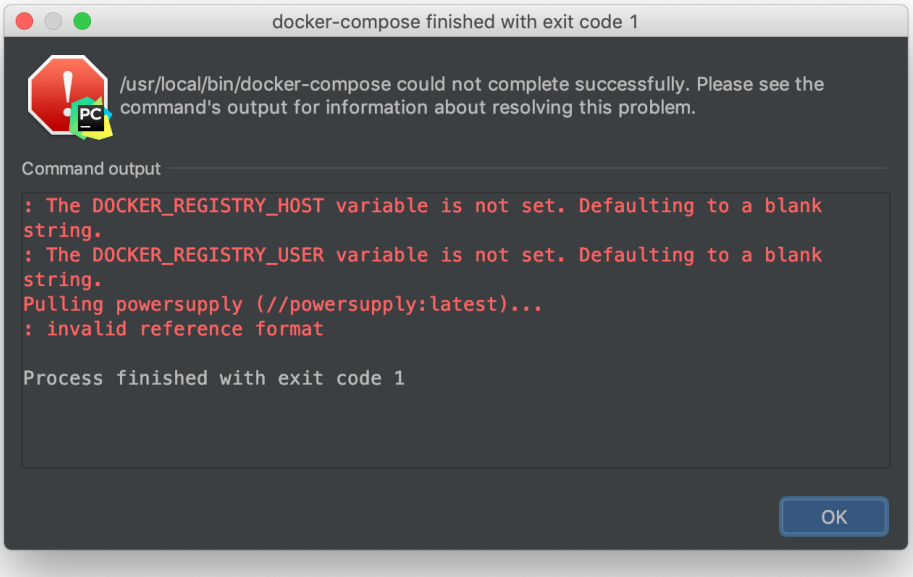
- **The device server is not defined in the database**

  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*.



- **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.



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

2.13 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.

2.13.1 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.

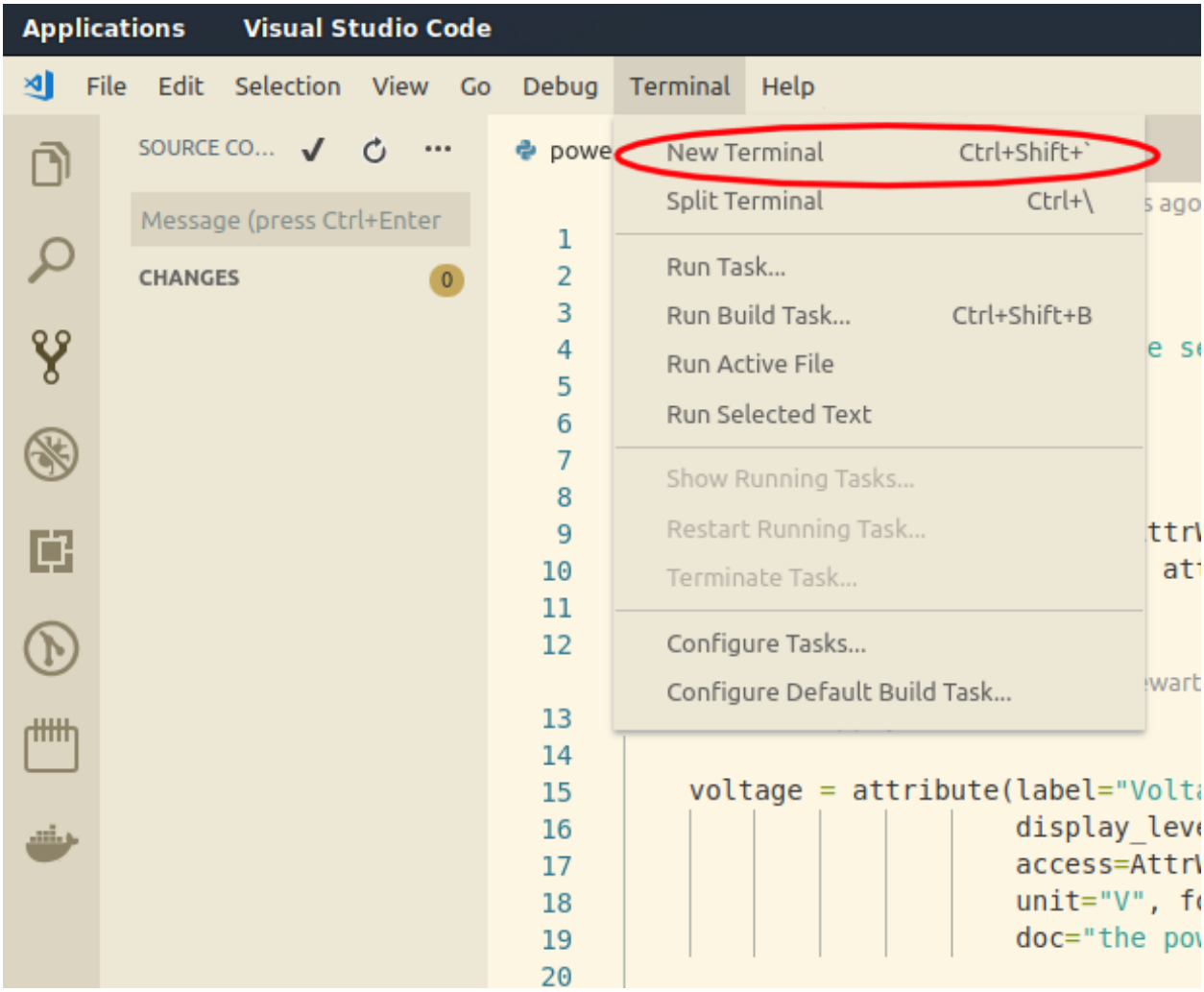
Clone the tango-example project and get VSCode to recognize it

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:



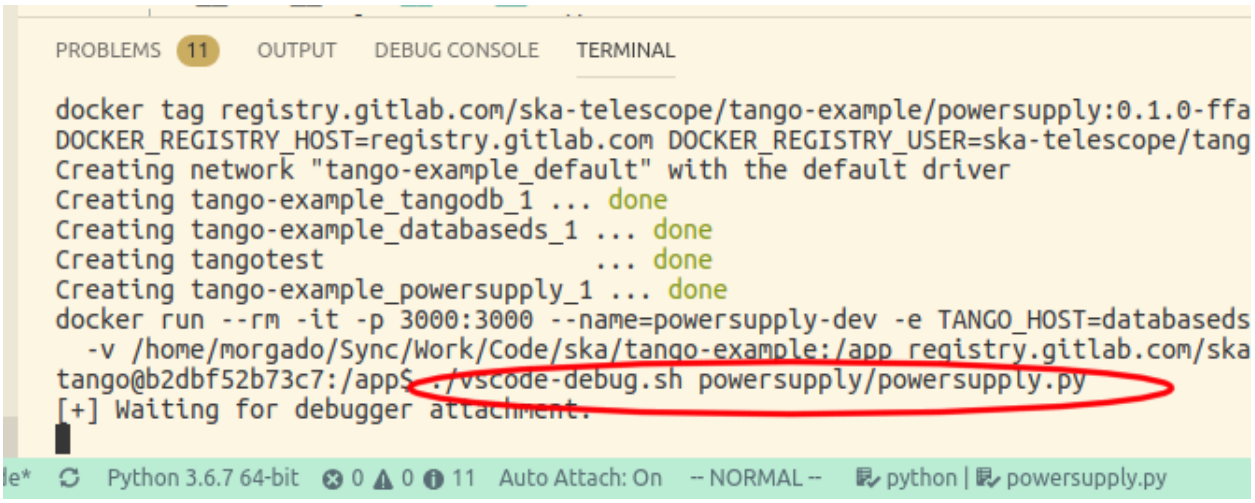
On the terminal tab build the image by typing `make build`. *This step is optional since the ''make interactive'' command described bellow, takes care of this task if needed*:

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`:

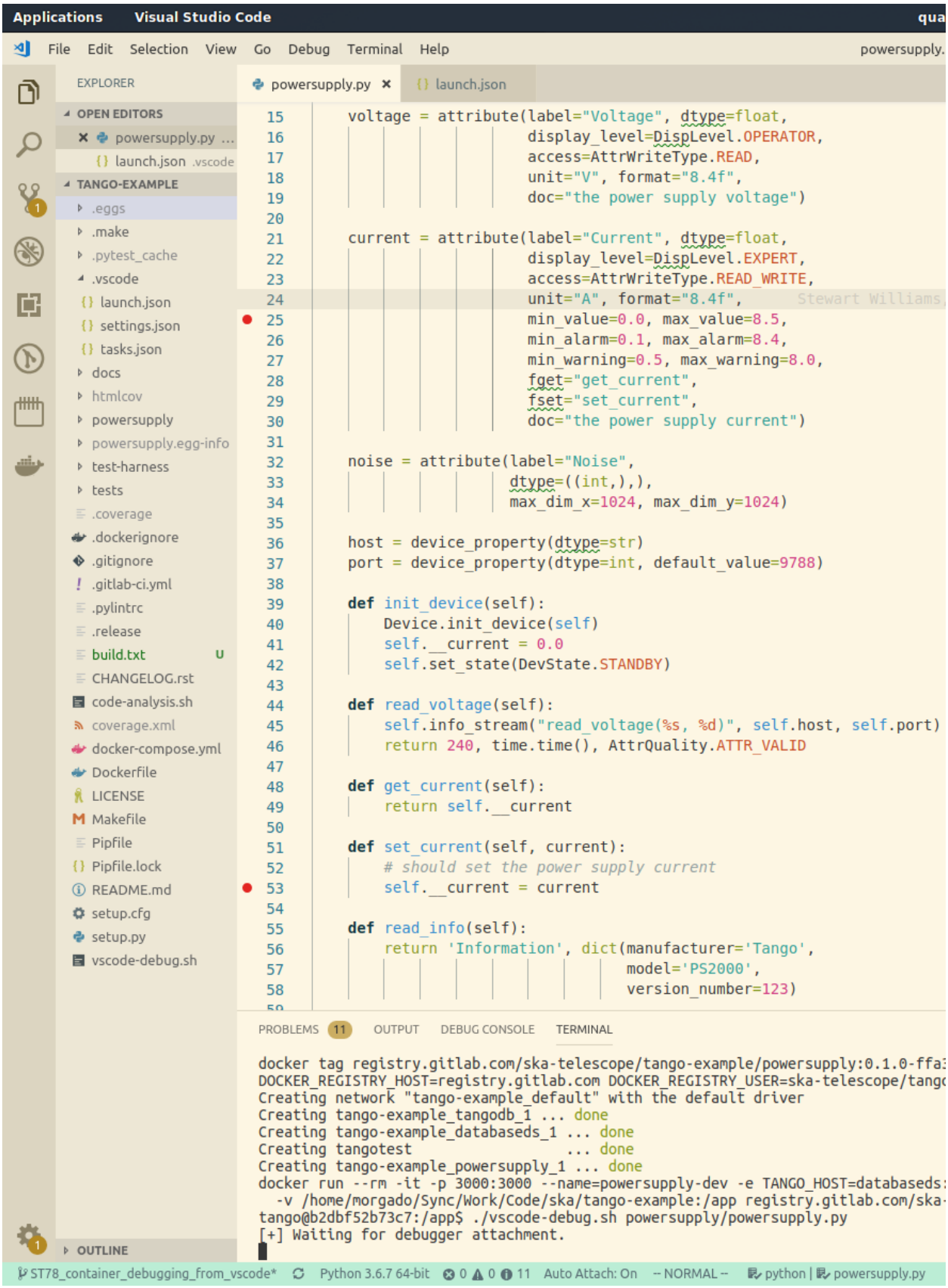
- Debug a particular file using the `vscode-debug.sh` utility inside the docker image. For instance `./vscode-debug.sh powersupply/powersupply.py`:



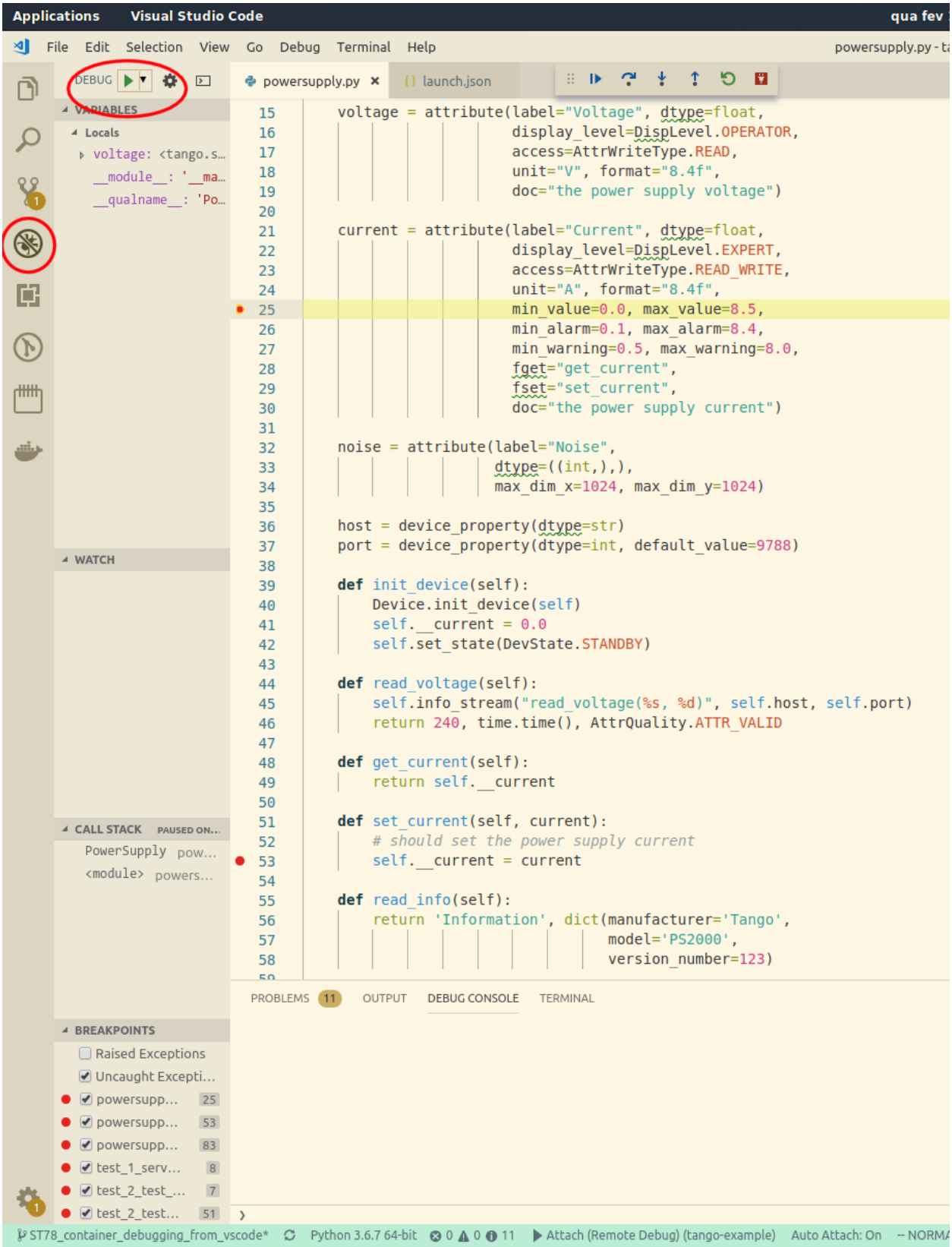
Notice that the terminal shell now shows a message stating that it is waiting for the debugger attachment:

```
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):


Applications
Visual Studio Code
File Edit Selection View Go Debug Terminal Help
EXPLORER
OPEN EDITORS
powersupply.py
launch.json .vscode
TANGO-EXAMPLE
.eggs
.make
.pytest_cache
.vscode
launch.json
settings.json
tasks.json
docs
htmlcov
powersupply
powersupply.egg-info
test-harness
tests
.coverage
.dockerignore
.gitignore
.gitlab-ci.yml
.pylintrc
.release
build.txt
CHANGELOG.rst
code-analysis.sh
coverage.xml
docker-compose.yml
Dockerfile
LICENSE
Makefile
Pipfile
Pipfile.lock
README.md
setup.cfg
setup.py
vscode-debug.sh
OUTLINE
powersupply.py
launch.json
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)
PROBLEMS 11
OUTPUT
DEBUG CONSOLE
TERMINAL
docker tag registry.gitlab.com/ska-telescope/tango-example/powersupply:0.1.0-ffa3
DOCKER_REGISTRY_HOST=registry.gitlab.com DOCKER_REGISTRY_USER=ska-telescope/tango
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.
ST78_container_debugging_from_vscode*
Python 3.6.7 64-bit
0 0 11
Auto Attach: On
-- NORMAL --
python | powersupply.py

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



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*

CHAPTER 3

Development tools

3.1 Working with git

Git is adopted as distributed version control system, and all SKA code shall be hosted in a git repository. The github organization *ska-telescope* can be found at https://github.com/ska-telescope . All SKA developers must have a github account and be added to the organization as part of a team.

- *Working with git*

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]

Todo:

- Create a new project
- Link to issue tracker

3.3 Development Environments

3.3.1 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.

- *Tango Development Environment set up*

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:

- *PyCharm*
- *Visual Studio Code*

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

1 Introduction

What follows is the software testing policy and strategy produced by Testing Community of Practice.

This is **version 1.1.0** of this document, completed on 2019-07-09.

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 SKA1 construction starts.

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 should start now (June 2019) and should cover at least the next 1-2 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 will be provided (tools, training, practices, guidelines), so that teams could adopt them.

We plan to cover at least 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).

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;
- systematically cover system-testing;
- 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 requirements and risks;
- be centered on "specification by example" yet.

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).

NOTE: the purpose of defining both "testing" and "debugging" is so that readers get rid of the idea that one does testing while he or she is doing debugging. They are two distinct activities.

4.2 Work organization

Testing is performed by the team who develops the software. There is no dedicated group of people who are in charge of testing, there are no beta-testers.

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 mostly those that are technology facing, hence quadrant Q1 (bottom left) and partly Q2 (top left), functional acceptance tests.

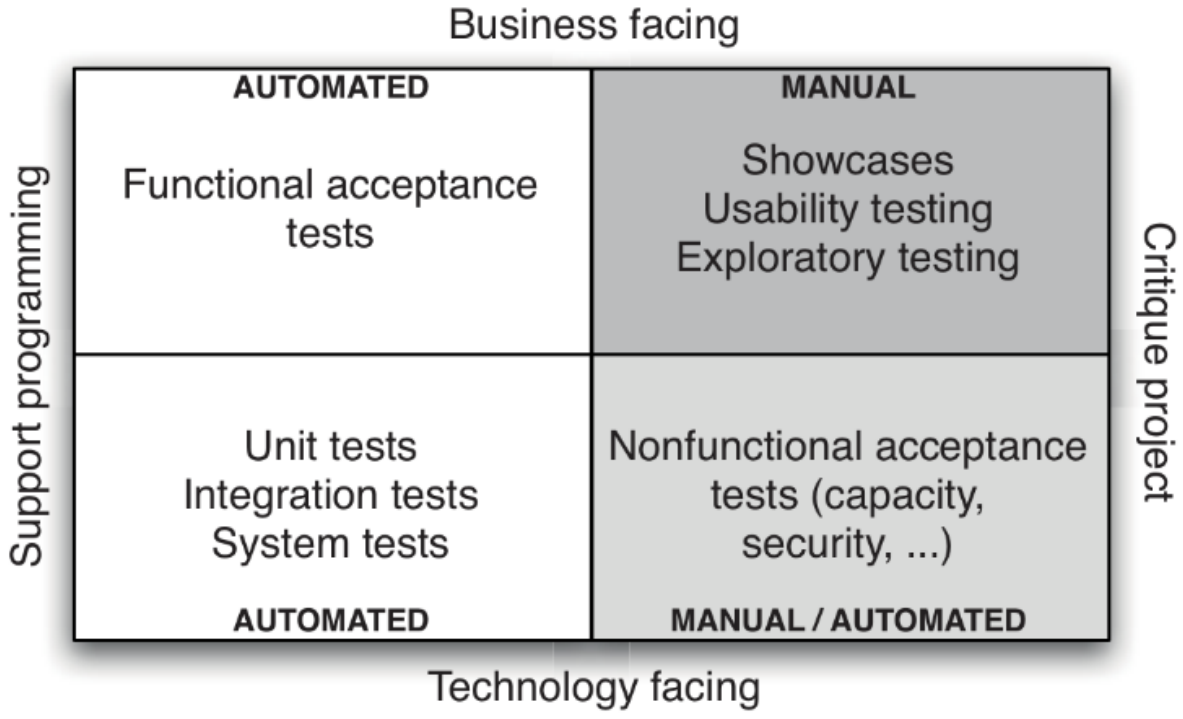


Figure 1: Test quadrants, picture taken from (Humble and Farley, Continuous Delivery, 2011).

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 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).

Emphasis on quadrant 1 of *Figure 1*, and low importance to the other quadrants, will allow the teams to be more focussed (within the realm of testing) and learn the basics. Some attention to quadrant 2 will let teams start addressing tests at a higher level, which bring along aspects like traceability (relationships between tests and requirements) and integration between subsystems.

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 already collected and displayed in https://argos.engageska-portugal.pt (username=viewer, password=viewer).

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, for Phase 1.

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 and make it 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 run within threads, processes, services or virtual docker components.

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 it does what it should.

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 a tester, which is 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. For the time being 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.

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.

5.3 Test specification

Programmers adopt a TDD 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.

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. All these tests are automated, possibly during the same sprint in which the user story is being developed.

5.4 Test environment

In this version of the strategy we do not cover provisioning of environments for running functional and performance tests of complex systems. We expect those teams to come up with suggestions and prototype solutions that could be included in this strategy later on.

5.5 Test data

In this version of the strategy we do not cover sophisticated mechanisms for handling data to support functional and performance tests of complex systems. We expect those teams to come up with suggestions and prototype solutions that could be included in this strategy later on.

5.6 Test automation

At the moment these elements are still under active investigation. As explained elsewhere in this portal, python developers generally rely on *pytest* and associated libraries (for assertions, for mocking); similarly developers uisng javascript rely on *Jest*. For component, system, and acceptance tests developers may rely also on Gherkin tests (aka, Behavior Driven Development tests).

5.7 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 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 during the same sprint are **fixed on the fly**, 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, to change it.

Logging occurs in JIRA by adding a new issue of type Bug to the product backlog and prioritized by the PO as every other story/enabler/spike. 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.

- SKA Bug tracking project

6 General references

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

Definition of Done

Done-ness is defined differently at different stages of development and for different purposes.

Story

- Code is supplied with an acceptable license
- Code is peer-reviewed and has been integrated into the main branch (via pull-request process)
- Code is checked into SKA repository with reference to ticket
- Code has tests that have adequate (between 75% and 90%) coverage
- Code compiles cleanly with no warnings
- Code adheres to SKA language specific style
- Code is deployed to continuous integration environment
- Code passes regression testing
- Code passes smoke test
- NFRs are met
- Story is tested against acceptance criteria
- Story is documented
- Story ok-ed by Product Owner

Code documentation

- Public API exposed is clearly documented
- Code is documented inline according to language specific standards
- Documentation is peer-reviewed by stakeholder (e.g. Product Owner for a feature or technical peer for an enabler) via pull-request mechanism.
- Documentation is deployed to externally visible website accessible via the developer portal.

Feature

- Feature has been demonstrated to relevant stakeholders
- Feature meets the acceptance criteria
- Feature is accepted by Feature owner
- Feature is integrated in an integration environment
- Code documentation is integrated as part of the developer portal
- Architectural documentation is updated to reflect the actual implementation

Formally Controlled Project Documentation

Some documentation (particularly architectural documentation) that impacts other parts of the project must to 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 are 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.

CHAPTER 4

Development practices followed at SKA

4.1 Testing policy and strategy

The SKA testing policy and strategy contains useful guidelines and practices to be followed when developing software for the SKA project.

- *Software Testing Policy and Strategy*

4.2 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*

4.2.1 Getting Started

I want to..

Add a new project to SKA organisation

- Our project details can be found in the section about how to *Create a new project*.

Develop a Tango device

- A sample PyTango device project that can be forked can be found at https://github.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

Our containerisation standards can be found in the containerisation section.

- **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 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://github.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
```

(continues on next page)

(continued from previous page)

```
   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
coredns-86c58d9df4-5ztg8            1/1     Running   0          3m24s
...
```

- **Verify Helm installation**
  - Helm installation instructions

```
$ helm

The Kubernetes package manager

To begin working with Helm, run the 'helm init' command:

$ helm init
...
```

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 github 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: Integration instructions.

- **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
...
```

4.2.2 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.

Todo:

- Testing Guidelines
- Writing Command-Line Scripts
- C or Cython Extensions

4.2.3 Python Coding Guidelines

This section describes requirements and guidelines.

Interface and Dependencies

- All code must be compatible with Python 3.5 and later.
- The new Python 3 formatting style should be used (i.e. `"{0:s}".format("spam")` instead of `"%s" % "spam"`).

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
"""
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'`.

Acknowledgements

The present document's coding guidelines are derived from project Astropy's codding guidelines.

4.2.4 SKA Javascript Coding Guidelines

Prerequisites:

As a javascript developer you will need the following:

- An account on GitHub
- Access to the SKA github account (https://github.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 GitHub 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 github.
- 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.

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

---

4.2.5 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.

4.2.6 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
```

(continues on next page)

(continued from previous page)

```
  artifacts:
    paths:
      - build

build_release:
  extends: .common
  stage: build
  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
```

(continues on next page)

(continued from previous page)

```
test:
  extends: .common
  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

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
```

(continues on next page)

(continued from previous page)

```
    - 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)
```

(continues on next page)

(continued from previous page)

```
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
# 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

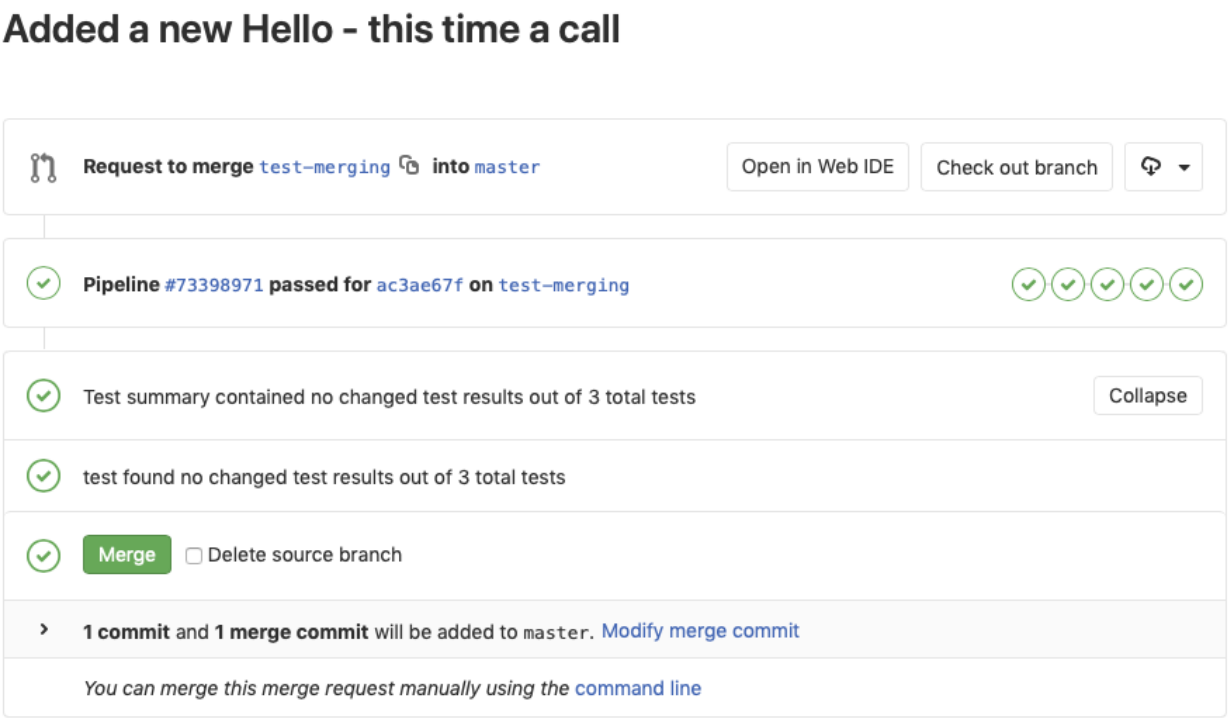
Within the template we give examples of how to write a Unit Test in The Google Test framework.

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:



You can see that there is not much detail - but the tests results are parsed and any differences are noted.

Pages (or Publishing Artifacts)

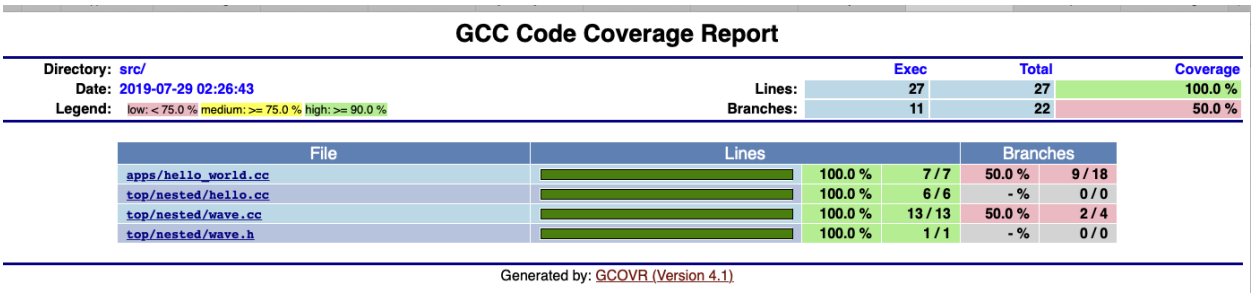
We use this step to run the test coverage tool gcov, and to publish the results:

```
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/	Lines:	27	27	100.0 %
Date:	2019-07-29 02:26:43	Branches:	11	22	50.0 %
Legend:	low: < 75.0 % medium: >= 75.0 % high: >= 90.0 %				

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

4.2.7 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.

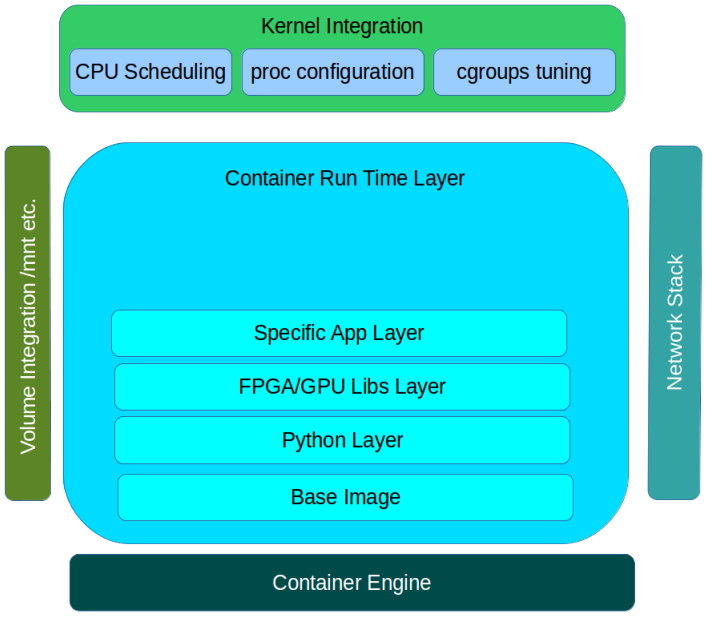


Fig. 1: The basic anatomy of a container and how it interfaces with host at run time.

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

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://github.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 && \
    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 Nexus Container Registry available at nexus.engageska-portugal.pt .

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 \
   nexus.engageska-portugal.pt/ska-docker/ska-python-runtime:1.2.3

Signatures for nexus.engageska-portugal.pt/ska-docker/ska-python-runtime:1.2.3

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

Administrative keys for nexus.engageska-portugal.pt/ska-docker/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) \
```

(continues on next page)

(continued from previous page)

```
  --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`.

4.2.8 Container Standards CheatSheet

This section provides a condensed 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.

Naming 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.
- Applicaiton 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.

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

4.2.9 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 v1.14.1 and Helm v2.13.1, 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
- 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

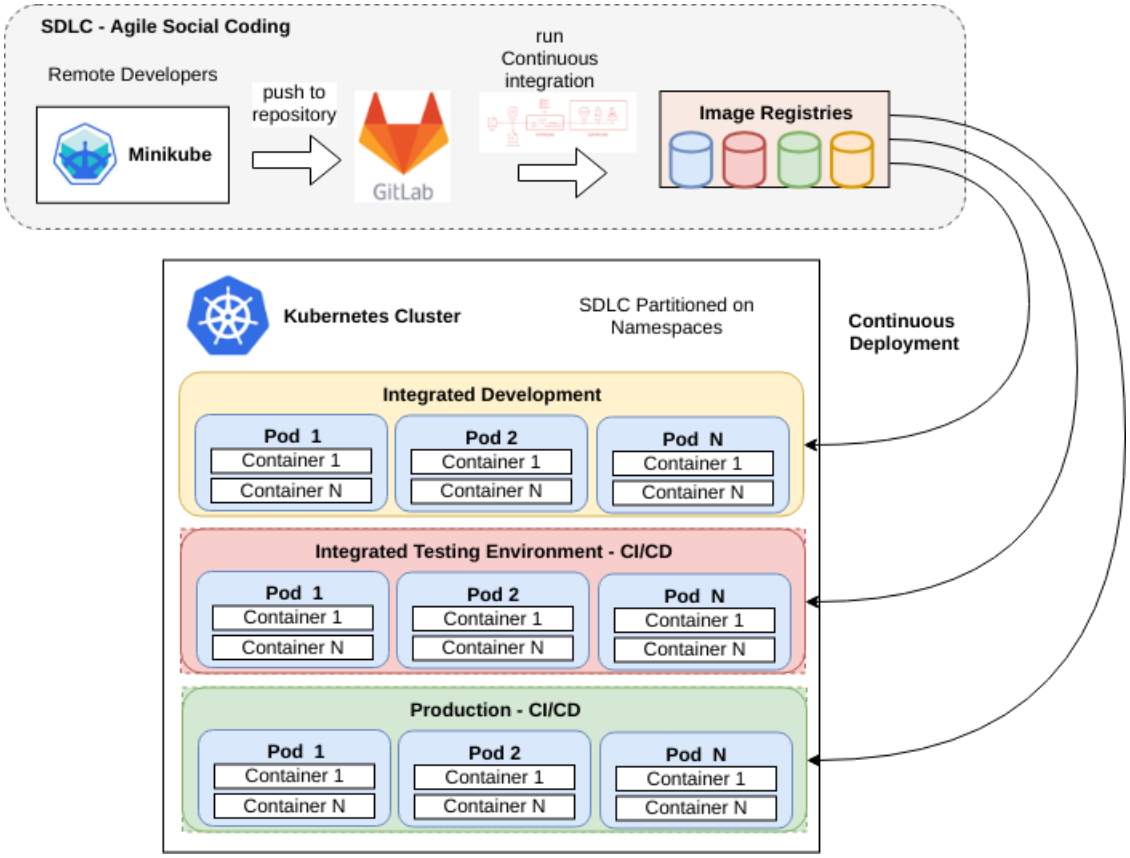


Fig. 2: How Kubernetes fits into the Cloud Native SDLC

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.

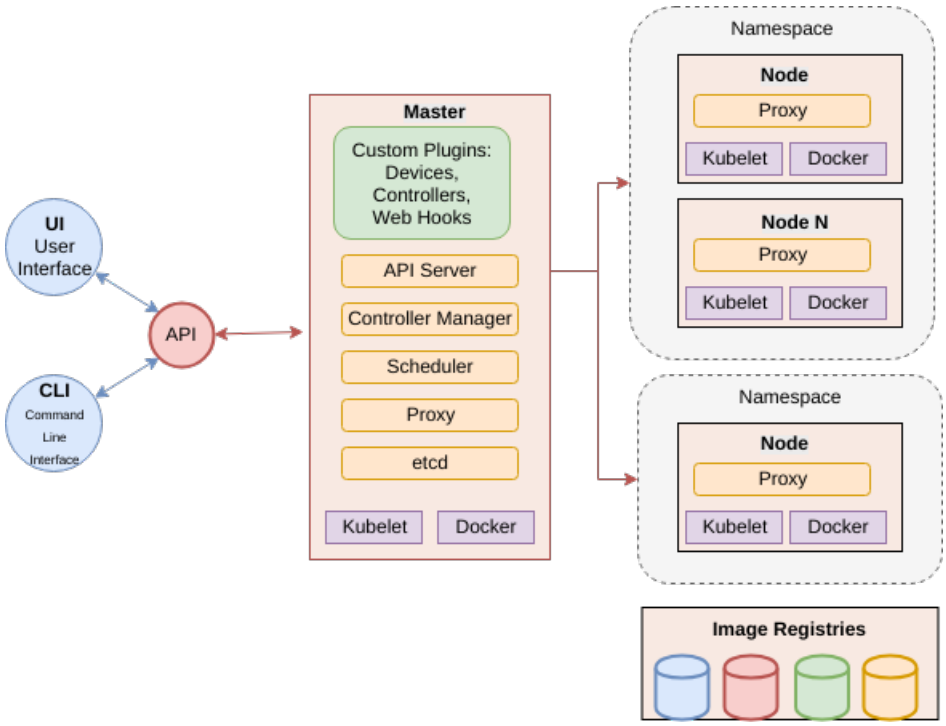


Fig. 3: The architecture of Kubernetes at the centre of the Cloud Native platform

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.

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

NAME                                            TYPE        CLUSTER-IP    EXTERNAL-IP␣
↪  PORT(S)                                        AGE
service/databaseds-integration-tmc-webui-test   ClusterIP   None          <none>      ␣
↪  10000/TCP                                      55s
```

(continues on next page)

(continued from previous page)

```
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
        LICENSE             # OPTIONAL: A plain text file containing the license for␣
↪the chart
        README.md           # OPTIONAL: A human-readable README file
        requirements.yaml   # OPTIONAL: A YAML file listing dependencies for the chart
        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 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.

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://github.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 -}}
```

(continues on next page)

(continued from previous page)

```
{{/*
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
```

(continues on next page)

(continued from previous page)

```
    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 }}
```

(continues on next page)

(continued from previous page)

```
  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
        - /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 nu-

merical 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
```

(continues on next page)

(continued from previous page)

```
    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
        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 }}"
```

(continues on next page)

(continued from previous page)

```
    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

---
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
```

(continues on next page)

(continued from previous page)

```
      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
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.

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:
      - sh
    args:
      - -c
      - "( retry --max=10 -- tango_admin --ping-device test/power_supply/1 ) && echo
↪'test OK'"
    env:
```

(continues on next page)

(continued from previous page)

```
    - 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

      initContainers:
      # initContainers can initialise data, and do pre-flight checks
      - name: init-container
        image: alpine
```

(continues on next page)

(continued from previous page)

```
        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

      # Ambassador pattern used as a proxy or shim to access external inputs
      # gets date from Google and adds it to input
      - name: ambassador-container
```

(continues on next page)

(continued from previous page)

```
        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

```
---
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
          subPath: mmapexample.py
          readOnly: true
        - mountPath: /dev/shm
          name: dshm
```

(continues on next page)

(continued from previous page)

```
      # 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 -*-
    """ example mmap python client
    """

    import datetime
    import time
```

(continues on next page)

(continued from previous page)

```
    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
                                   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'))
```

(continues on next page)

(continued from previous page)

```
        # 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 github.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;
          mkdir /lib64 && ln -s /lib/libc.musl-x86_64.so.1 /lib64/ld-linux-x86-64.so.
↪2;
          ipcmk --shmem 1KiB;
```

(continues on next page)

(continued from previous page)

```
          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
  labels:
    app: pod-sharing-examples
spec:
```

(continues on next page)

(continued from previous page)

```
  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"]
        ports:
        - name: ncat
          containerPort: 1234
```

(continues on next page)

(continued from previous page)

```
          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
        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
↪"]
```

(continues on next page)

(continued from previous page)

```
      # 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
```

(continues on next page)

(continued from previous page)

```
      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:
    - podSelector:
    # enable the DatabaseDS interface
        matchLabels:
```

(continues on next page)

(continued from previous page)

```
          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 tolleration

Repeat obs6 as obs7 but add a tolleration to the NoSchedule taint:

node select rack01, with Tolleration to Taint NoSchedule

```
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: obs7-rack01-taint-and-tolleration
  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-tolleration
        command: ["sleep", "365d"]
      nodeSelector:
        rack: rack01
      tolerations:
      - key: "key1"
        operator: "Equal"
        value: "value1"
        effect: "NoSchedule"
```

Now with the added a Tolleration to the Taint, we have the following:

```
NAME                                READY STATUS RESTARTS AGE IP              NODE␣
↪NOMINATED NODE
obs7-rack01-taint-and-tolleration-0 1/1   Running 0       33s 192.168.105.184 k8s-
↪minion-0 <none>
obs7-rack01-taint-and-tolleration-1 1/1   Running 0       32s 192.168.72.27   k8s-
↪master-0 <none>
obs7-rack01-taint-and-tolleration-2 1/1   Running 0       31s 192.168.105.182 k8s-
↪minion-0 <none>
```

For a StatefulSet that has nodeSelector and Tollerations:

```
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.

4.2.10 Software Package Release Procedure

Versioning Procedure

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 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 python packages, for now, and have them published on the *SKA* pypi registry which is hosted on *Nexus*.

Versioning scheme to use

The scheme chosen to be adopted by the *SKA* developer community is the semantic versioning scheme. More information regarding this scheme can be found on the Semver site.

Python Packaging Procedure

How to mark a release

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 should be N.N.N

Minimum requirements to meet

For proper Python packaging, the following metadata must be present in the repository:

- Package name
- Package version
- Github repo url
- Description of the package
- Classifiers

All of this should be specified in the *setup.py* module that lives in the root directory.

Additional metadata files should be included in the root directory, that is the:

- 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 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 used for pure python packages
```

This will build a *Python* wheel that will be published to *Nexus*. 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 *SKA* pypi registry on *Nexus*.

```
publish to nexus:
  stage: publish
  tags:
    - docker-executor
  variables:
    TWINE_USERNAME: $TWINE_USERNAME
    TWINE_PASSWORD: $TWINE_PASSWORD
  script:
    # check metadata requirements
    - scripts/validate-metadata.sh
    - pip3 install twine
    - twine upload --repository-url $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
```

Installing a package from *Nexus*

For developers who want to install a python package from the *SKA* pypi registry hosted on *Nexus*, they should edit the project's Pipfile to have the following section(s), for example:

```
[[source]]
url = 'https://nexus.engageska-portugal.pt/repository/pypi/simple'
verify_ssl = true
name = 'nexus'

[packages]
'skaskeleton' = {version='*', index='nexus'}
```

Helm Chart Packaging, Publishing and Sharing

Working with a Helm Repository

Working with a Helm chart repository is well-documented on The Official Helm Chart Repository Guide.

Adding (our) repository

Note: Our Helm chart repository URL is https://nexus.engageska-portugal.pt/repository/helm-chart

In order to add the Helm chart repo to your local list of repositories, run

```
$ helm repo add [REPONAME] https://nexus.engageska-portugal.pt/repository/helm-chart
```

where [REPONAME] is a name you choose to identify the repo on your local machine. In order to standardise, we would recommend you use `skatelescope`.

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.

Package and publish Helm Charts to the SKA Helm Chart Repository

Package a chart

Packaging a chart is relatively trivial. Let's say you create a chart called `my-new-chart`, with the following directory structure:

```
$ tree
  my-new-chart/
  ├── Chart.yaml
  ├── charts
  ├── templates
  │   ├── NOTES.txt
  │   ├── _helpers.tpl
  │   ├── deployment.yaml
  │   ├── ingress.yaml
  │   ├── service.yaml
  │   └── tests
  │       └── test-connection.yaml
  └── values.yaml
```

You can now package it with the simple command `helm package my-new-chart` and it will create the package file:

```
$ helm package my-new-chart
$ tree
  my-new-chart-0.1.0.tgz
  my-new-chart/
  ├── Chart.yaml
      ...
  └── values.yaml
```

Now, in order to publish the chart, you can simply run a curl command with an `--upload-file` flag:

```
$ curl -v -u $USERNAME:$PASSWORD --upload-file new-helm-package-v-0.1.0.tgz --url␣
↪https://nexus.engageska-portugal.pt//repository/helm-chart/
```

Note: You will of course need credentials (`$USERNAME` and `$PASSWORD`) to run the above `curl` command with, and this user should have privileges for reading and writing on the Repository. We have a Gitlab Runner already set up with a user that has the required privileges - see *how to do this* below.

If you now run `helm repo update` you (or your colleagues) should see your new application also listed under the repo:

```
$ 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
skatelescope/my-new-chart       0.1.0           1.0             A Helm chart for␣
↪deploying the WebJive on Kubernetes
```

Bulk package and publish using Gitlab CI

Read the Helm documentation (note: this link is for Helm v2 because we are still using it) in order to learn how to publish your application to a Helm repository. This is an example of a branch-specific job in the CI pipeline with manual input required. See comments below the code block.

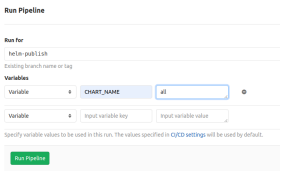
```
stages:
        - helm-publish

...

publish-chart:
  stage: helm-publish
  when: manual # the script will require you to specify the name of the chart that you want to package - CHART_NAME
  only:
        - helm-publish
  tags:
        - helm-repoman
  script:
        - cd charts
        - if [ $CHART_NAME == 'all' ]; then for d in */; do helm package "$d"; done; else helm package $CHART_NAME; fi
        - for f in *.tgz; do curl -v -u $RAW_USER:$RAW_PASS --upload-file $f $RAW_HOST/repository/helm-chart/$f; rm $f; done
```

Here are a few comments, referring to the line numbers above:

ln 1: Remember to add the stage at the top of your `.gitlab-ci.yml` **ln 8**: Setting `when: manual` will cause the pipeline to wait for the user to trigger the job.

When you try to run the blocked job for the first time, Gitlab will ask you to input variables - see next image. You need to specify `CHART_NAME` and use the exact name of the chart, for instance `my-new-app`.

You can either specify the chart name that is to be published, or input `all` so that every chart **in the top-level** `charts/` folder is published.



ln 9: Restricting the job to run only on the `helm-publish` branch is suggested because this job is run manually and thus will block a pipeline, waiting for manual input, which is not necessarily good for CI, especially if publishing new charts is not required.

ln 11: The runner's name is `helm-repoman` - it's available for all projects in the SKA group.

4.2.11 Hosting a docker image on the *Nexus* registry

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 *Nexus*.

Tagging the docker image

To explicitly tag a docker image run the following command:

```
$ docker tag <source_image> nexus.engageska-portugal.pt/<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 *NEXUS*

Once the docker image has been built and tagged, it can then be uploaded to the *Nexus* registry by executing the following command:

```
$ docker push nexus.engageska-portugal.pt/<repository_name>/<image_name>:<tag_name>
```

4.2.12 SKA Log Message Format

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                      ?
```

(continues on next page)

(continued from previous page)

```
CRITICAL Critical      2                     ?
======== ============= ===================== ====================
```

For guidelines on when to use a particular log level, please refer to the official Python logging HOWTO.

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

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`

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?

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`

[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

CHAPTER 5

Development guidelines

5.1 Getting Started Guide

A pocket-guide to documentation on adding a new project, development, containerisation and integration.

- *Getting Started*

5.2 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*

5.3 Python coding guidelines

A Python skeleton project is created for use within the SKA Telescope. This skeleton purpose is to enforce coding best practices and bootstrap the initial project setup. Any development should start by forking this skeleton project and change the appropriate files.

- *Python Coding Guidelines*

5.4 Javascript coding guidelines

A React based javascript skeleton project is created for use within the SKA Telescope. Similar to the Python skeleton above its purpose is to enforce coding best practices and bootstrap the initial project setup for browser based javascript applications.

- *SKA Javascript Coding Guidelines*

5.5 VHDL coding guidelines

VHDL coding guidelines are described at:

- *VHDL Coding Style Guidelines*

5.6 C++ Coding Standards

A CPP skeleton project is created for use within the SKA Telescope. The skeleton purpose is to demonstrate coding best practices, boostrap initial project set up within the SKA Continuous Integration (CI) Framework.

- *C++ Coding Guidelines*

5.7 Containerisation Standards

A set of standards, conventions and guidelines for building, integrating and maintaining Container technologies.

- *Containerisation Standards*

5.8 Container Orchestration Guidelines

A set of standards, conventions and guidelines for deploying application suites on Container Orchestration technologies.

- *Container Orchestration Guidelines*

5.9 SKA Software Packaging Procedure

This details a procedure that all *SKA* developers shall follow to ensure that they make use of the existing CI/CD pipelines to automate the building of their software packages for release.

- *Software Package Release Procedure*

5.10 Hosting a docker image on Nexus

This details steps that all *SKA* developers shall abide to when building and hosting their docker images on the Nexus registry.

- *Hosting a docker image on the Nexus registry*

5.11 Logging guidelines

A standard logging format exists for logging in the evolutionary prototype into an ELK stack logging system designed for the SKA software.

- *SKA Log Message Format*

5.11.1 List of projects

The following table is automatically extracted from our github organisation page at [https://gitlab.com/ska-telescope]

Gitlab repository	Documentation	Description
testgit	testdoc	test description
testgit	testdoc	test description

5.11.2 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.

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'*

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.

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'

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.

5.11.3 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

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.

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.

Integrating into the Developer Portal

The developer portal is hosted on ReadTheDocs. On the *List of projects* 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.

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.

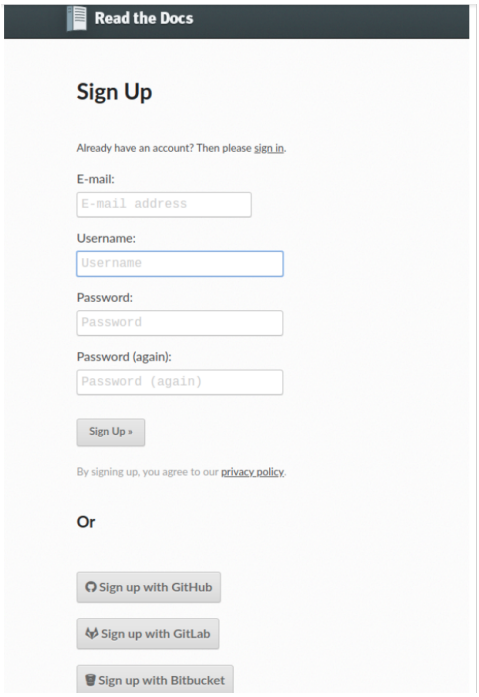
Sign up / sign in with GitLab account

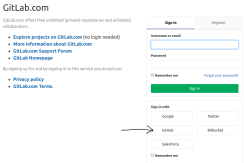
Log in with OAuth

As of October 2019, most developers are also registered on the SKA GitHub, and some may sign in using the OAuth credentials provided by GitHub, when signing into GitLab. This will be the preferred route.

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.


Read the Docs
Sign Up
Already have an account? Then please sign in.
E-mail:
E-mail address
Username:
Username
Password:
Password
Password (again):
Password (again)
Sign Up »
By signing up, you agree to our privacy policy.
Or
Sign up with GitHub
Sign up with GitLab
Sign up with Bitbucket


GitLab.com

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.

5.11.4 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.

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.

Licenses

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.

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
```

(continues on next page)

(continued from previous page)

```
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.

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

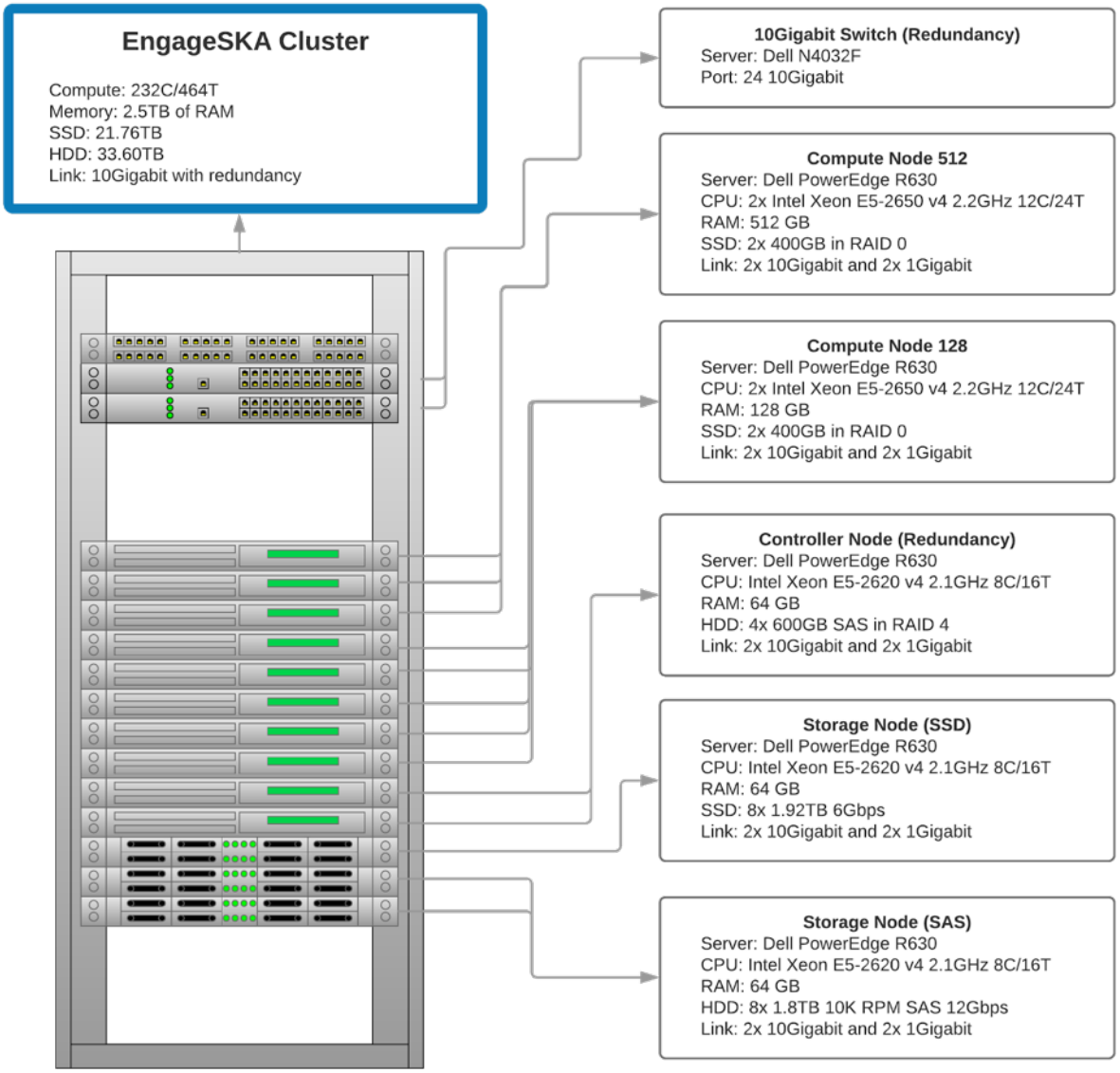
```
Copyright <YEAR> <COPYRIGHT HOLDER>

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
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
```

CHAPTER 6

Projects

6.1 AIT cluster

6.1.1 Cluster specs



6.1.2 Access the cluster

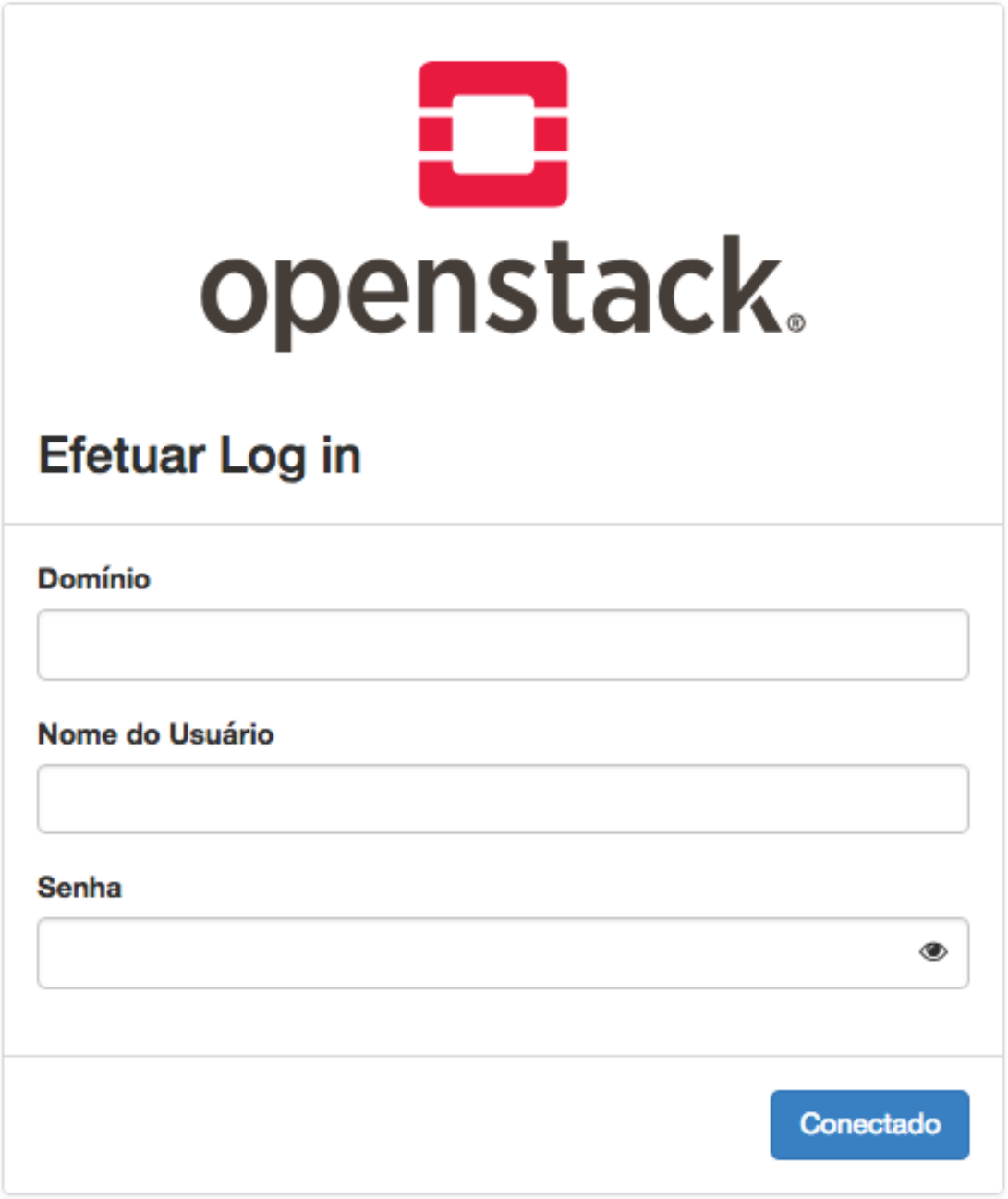
The EngageSKA cluster locates at the Datacenter of Institute of Telecommunication (IT) in Aveiro. To have access to the cluster, it is required to be inside the facilities or have VPN credentials to access the IT network remotely.

6.1.3 Access to the network using VPN

At the moment, VPN credentials are sent individually and is required to send an email to Dzianis Bartashevich (bartashevich@av.it.pt) with the knowledge from Marco Bartolini (M.Bartolini@skatelescope.org).

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

6.1.4 Access to the OpenStack platform (Virtualization)



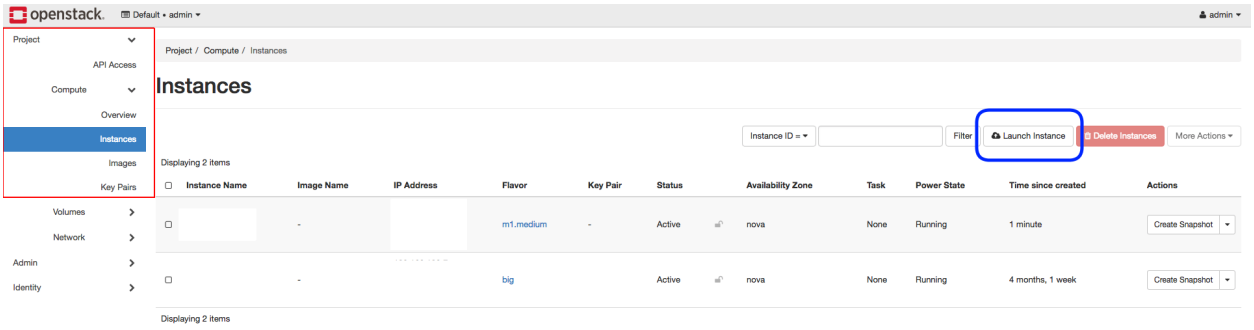
The OpenStack platform requires authentication in order to use it.

At the moment, OpenStack credentials are sent individually and it is required to send an email to Dzianis Bartashevich (bartashevich@av.it.pt with the knowledge from Marco Bartolini (M.Bartolini@skatelescope.org). In the next phase, OpenStack could support GitHub authentication.

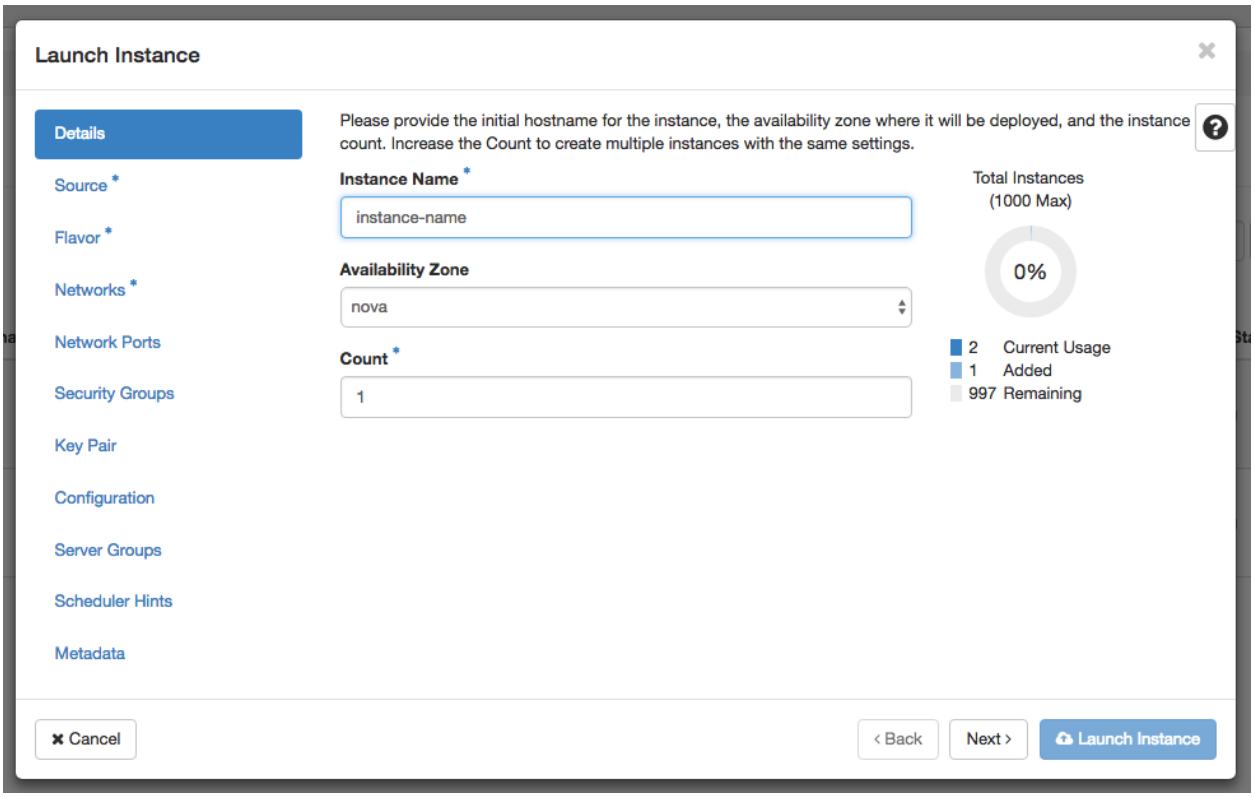
To access the OpenStack platform go to http://192.168.93.215/dashboard (require VPN) and login with your credentials.

Virtual machine deployment

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

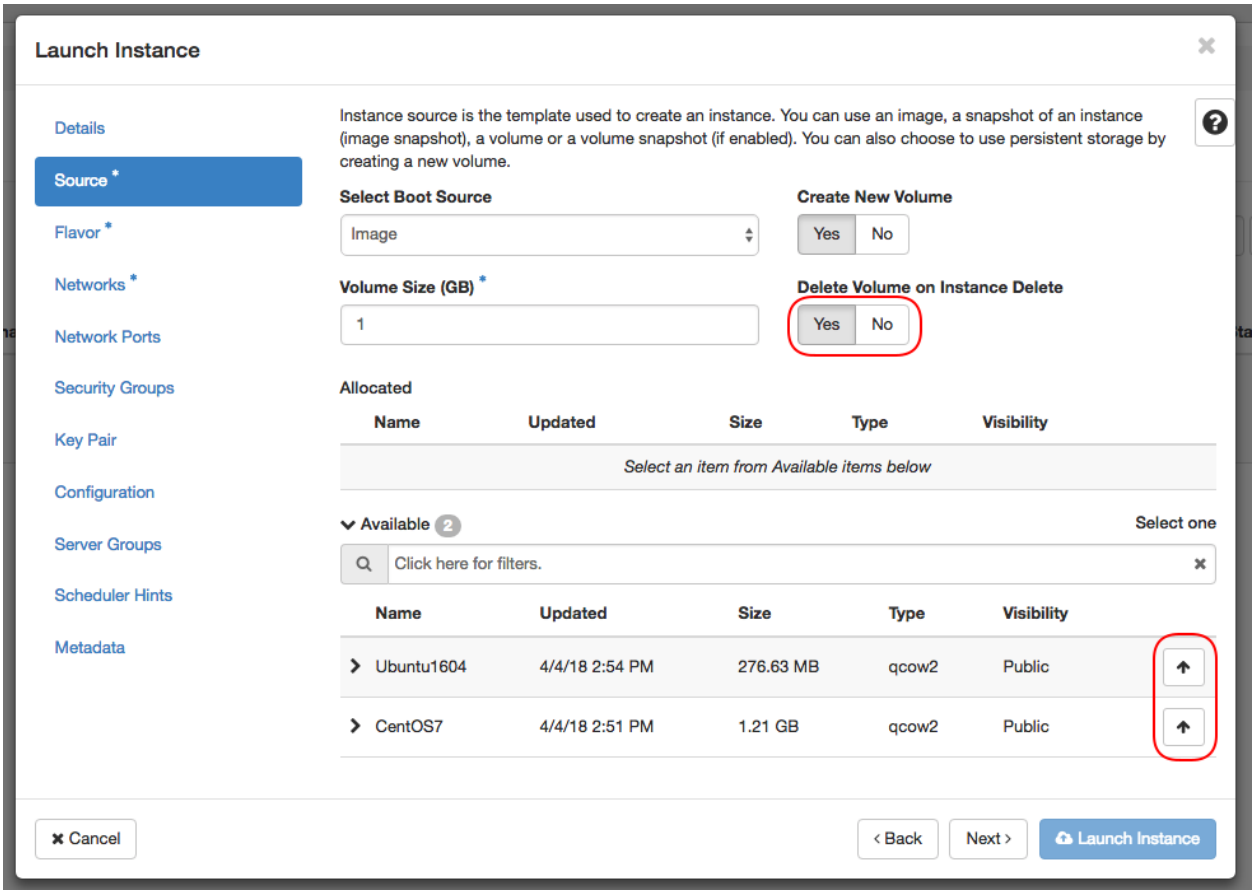


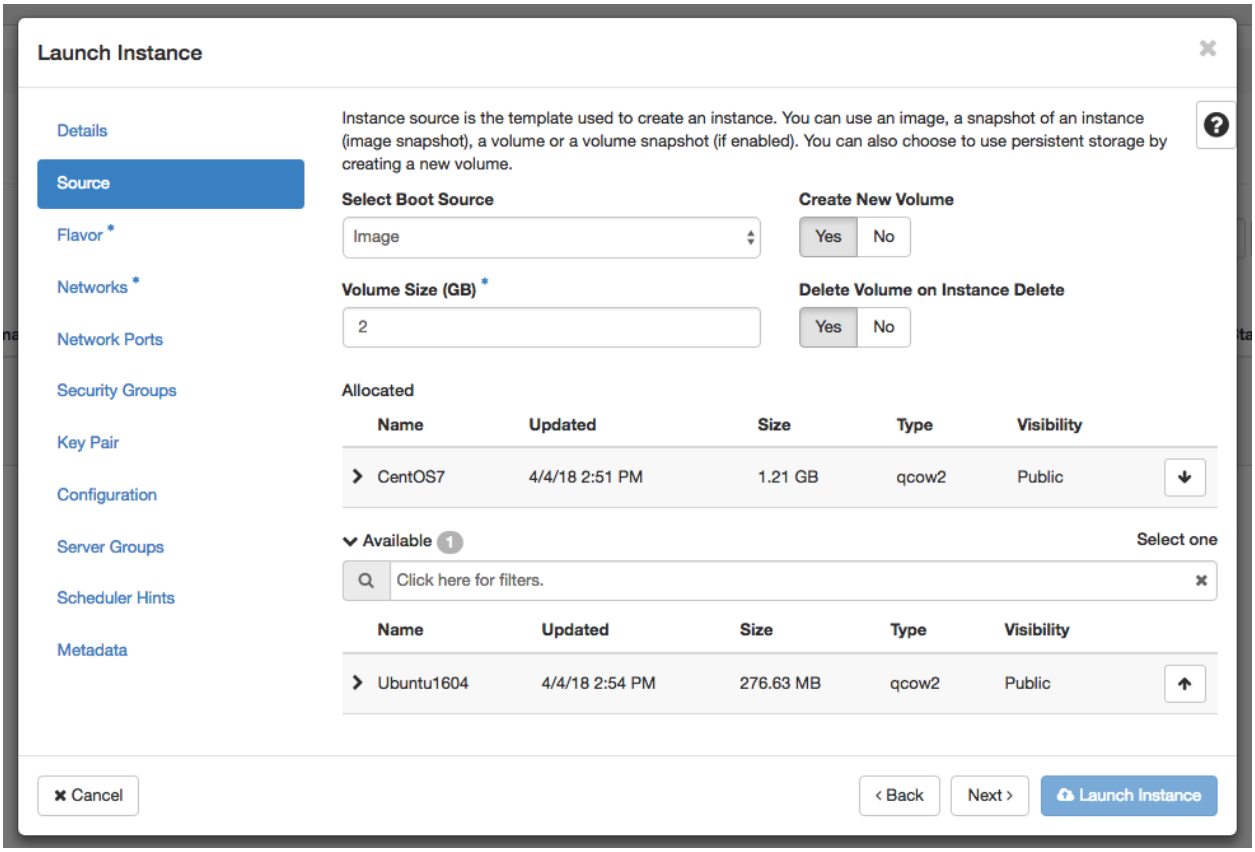
- At this stage a menu will pop-up and will ask to specify virtual machine caracteristics, chose an name for virtual machine:



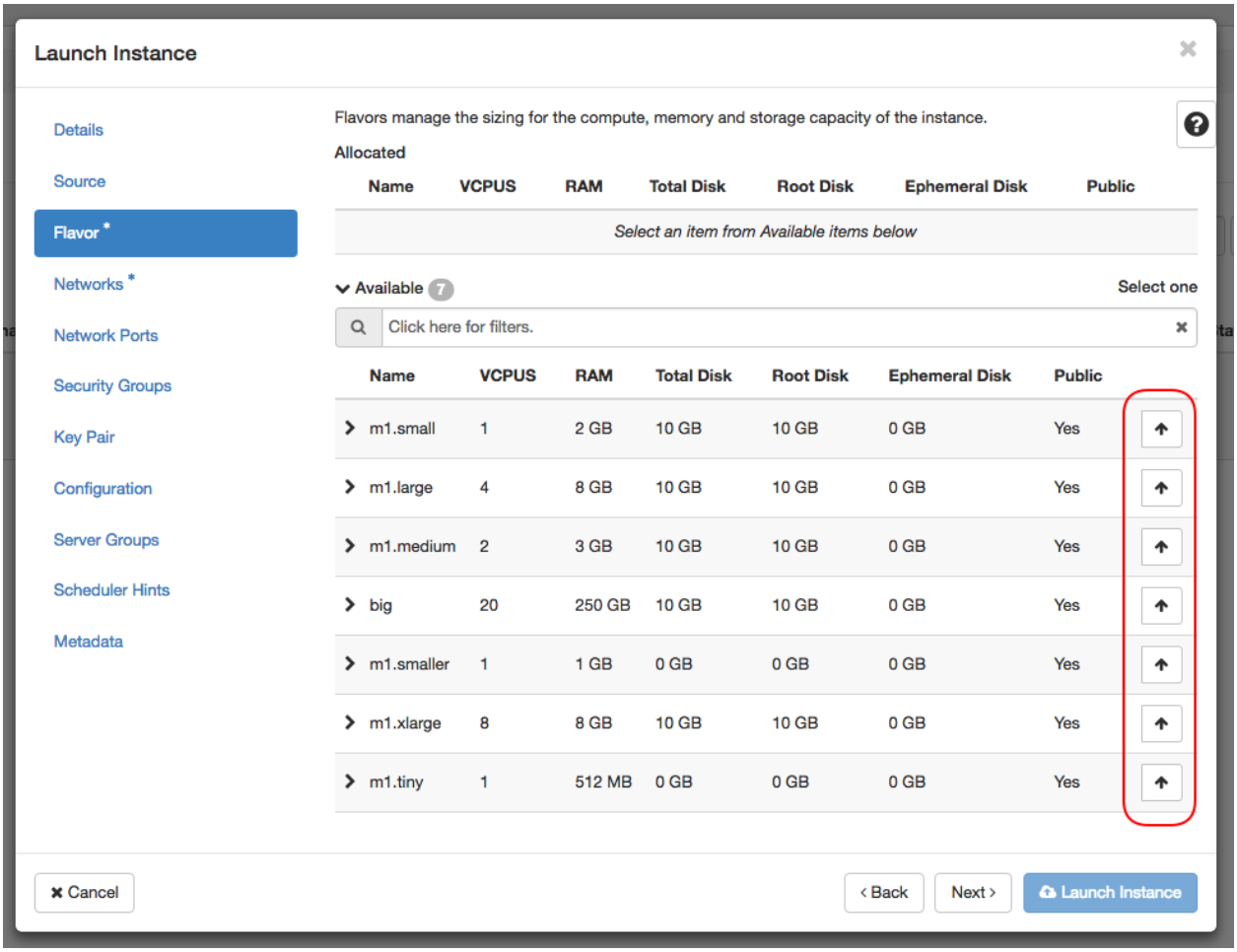
- Select the Operating System you want your VM to have:

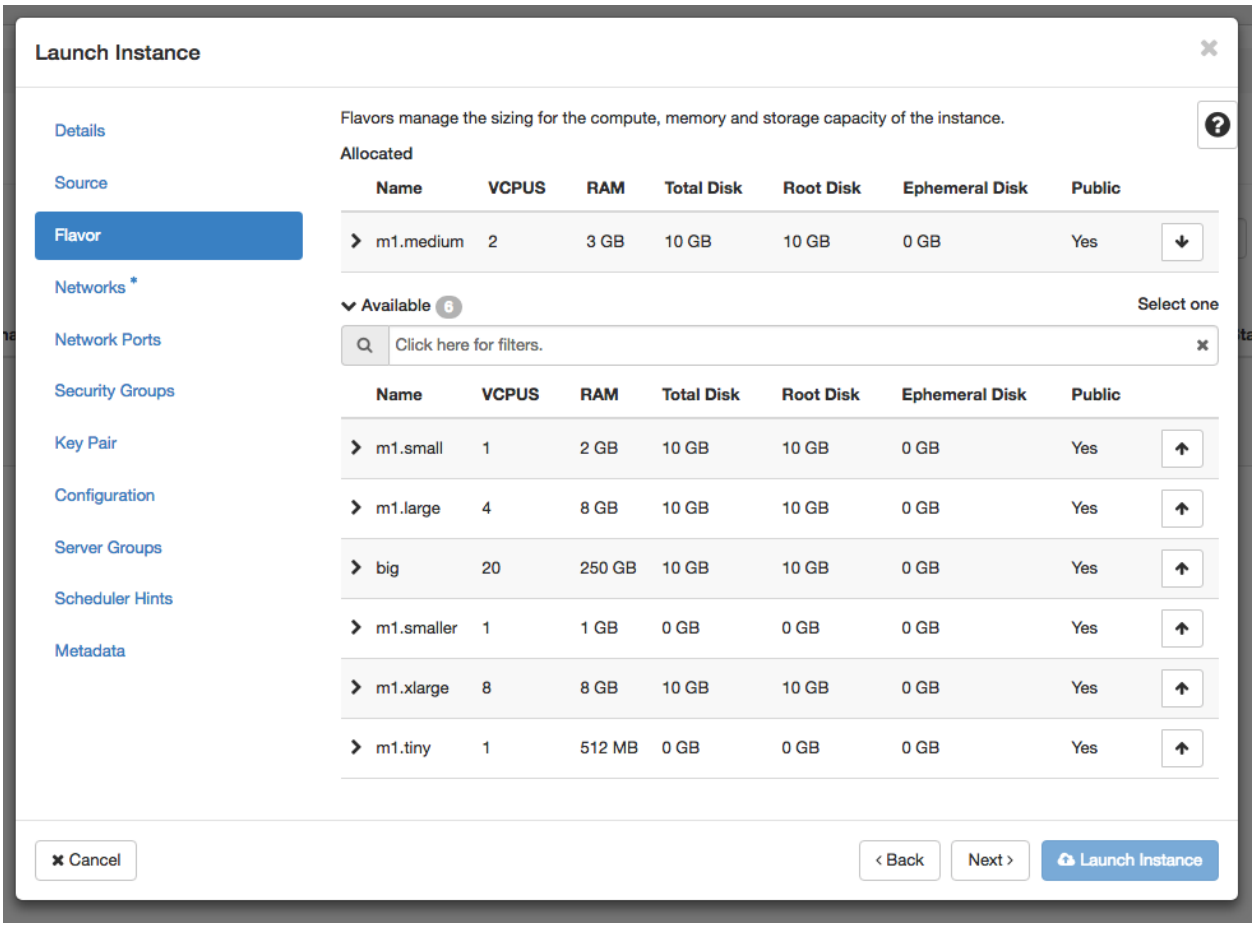
NOTE: Please choose the option "Yes" at "Delete Volume on Instance Delete" so when you decide to delete the instance the volume will be also deleted and not occupy unnecessary space



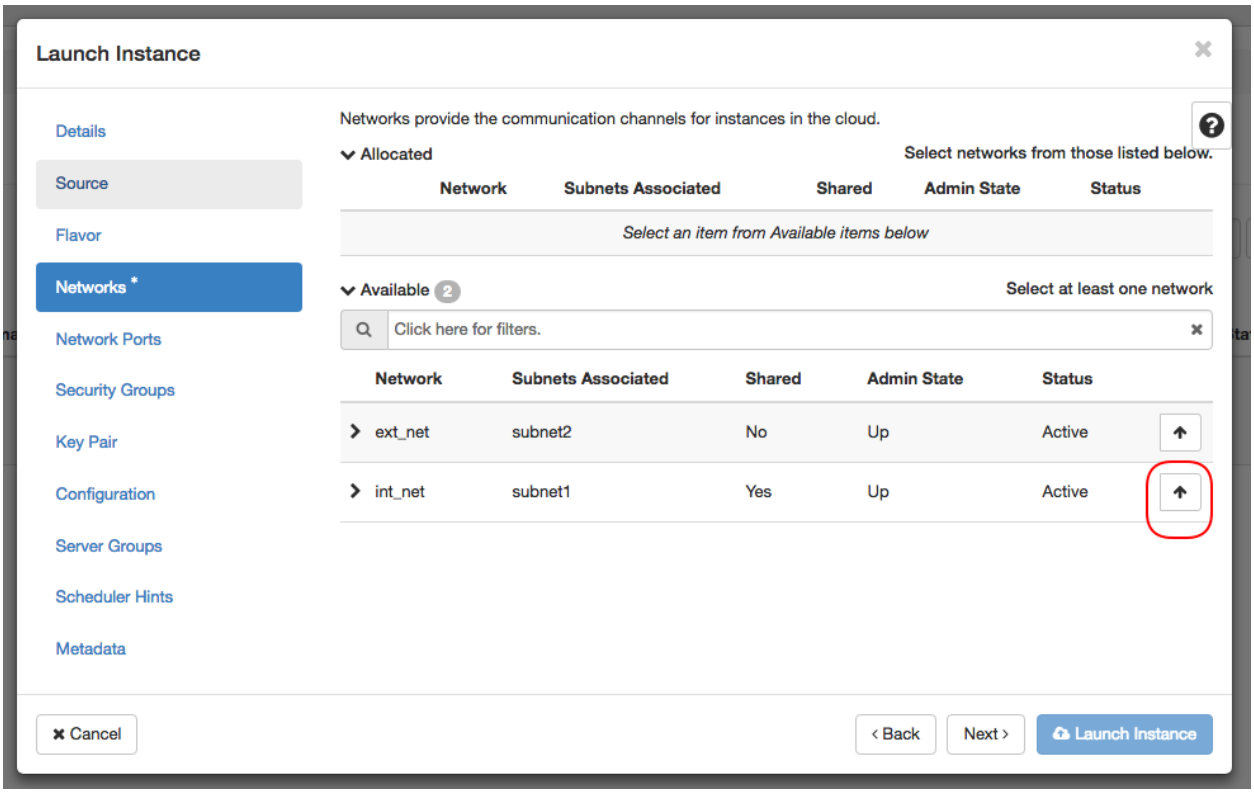


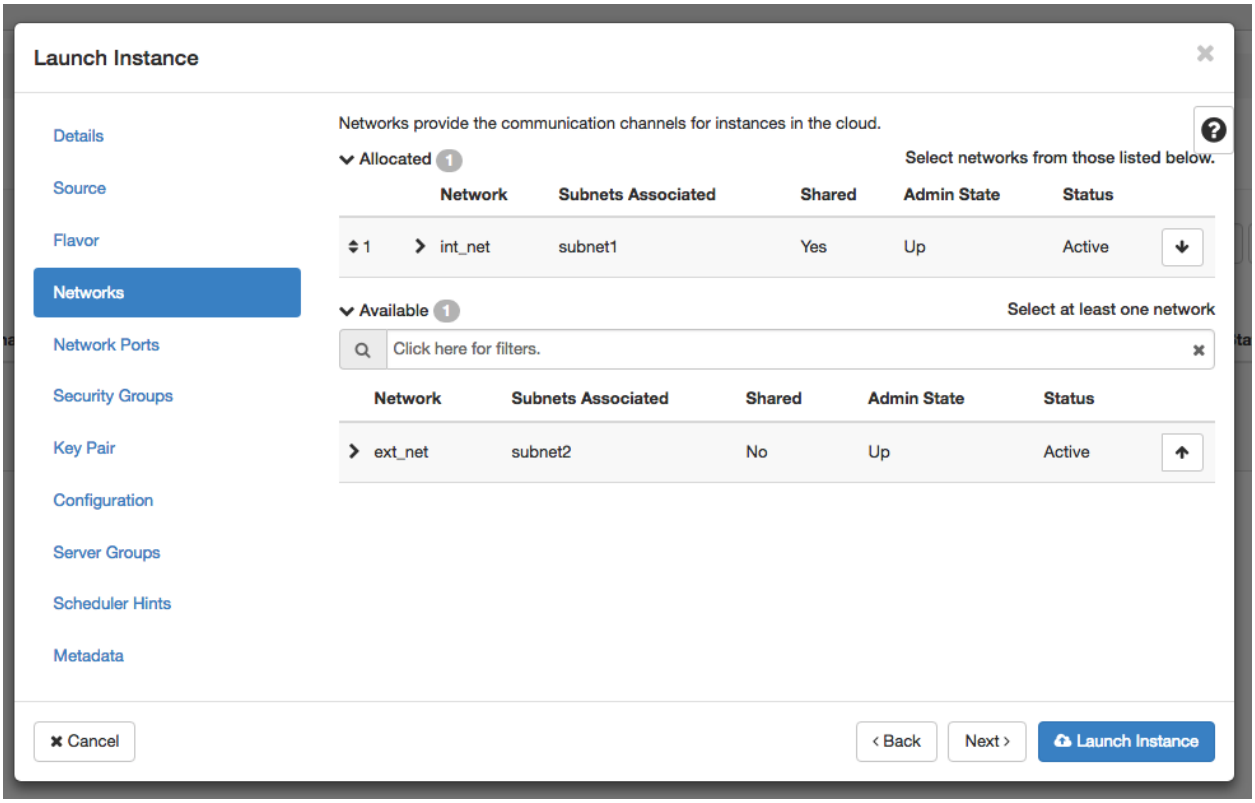
- Select the flavor which you want your VM to have:


Launch Instance
Details
Source
Flavor *
Networks *
Network Ports
Security Groups
Key Pair
Configuration
Server Groups
Scheduler Hints
Metadata
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
Cancel
Back
Next
Launch Instance

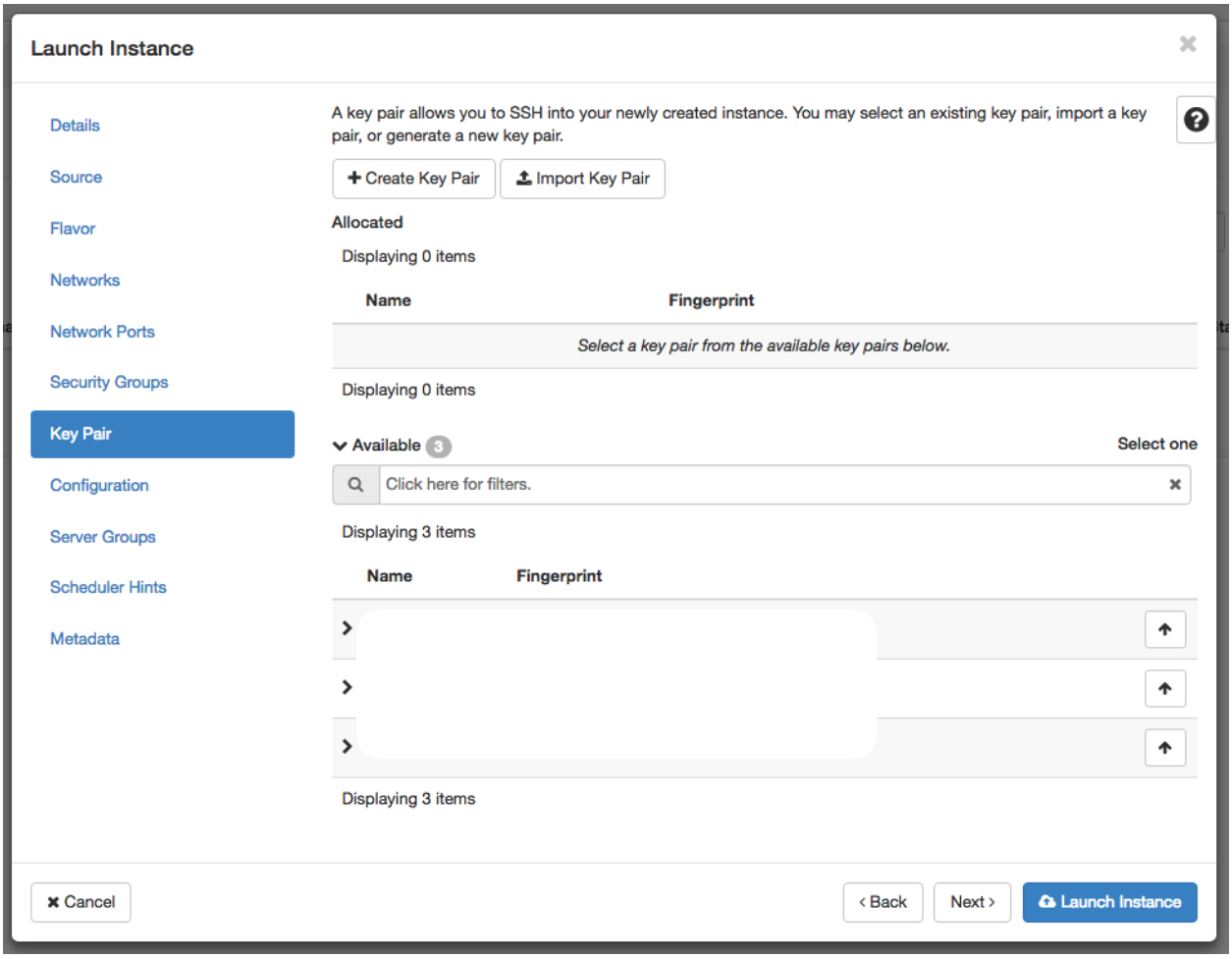


- Select private network (int-net):

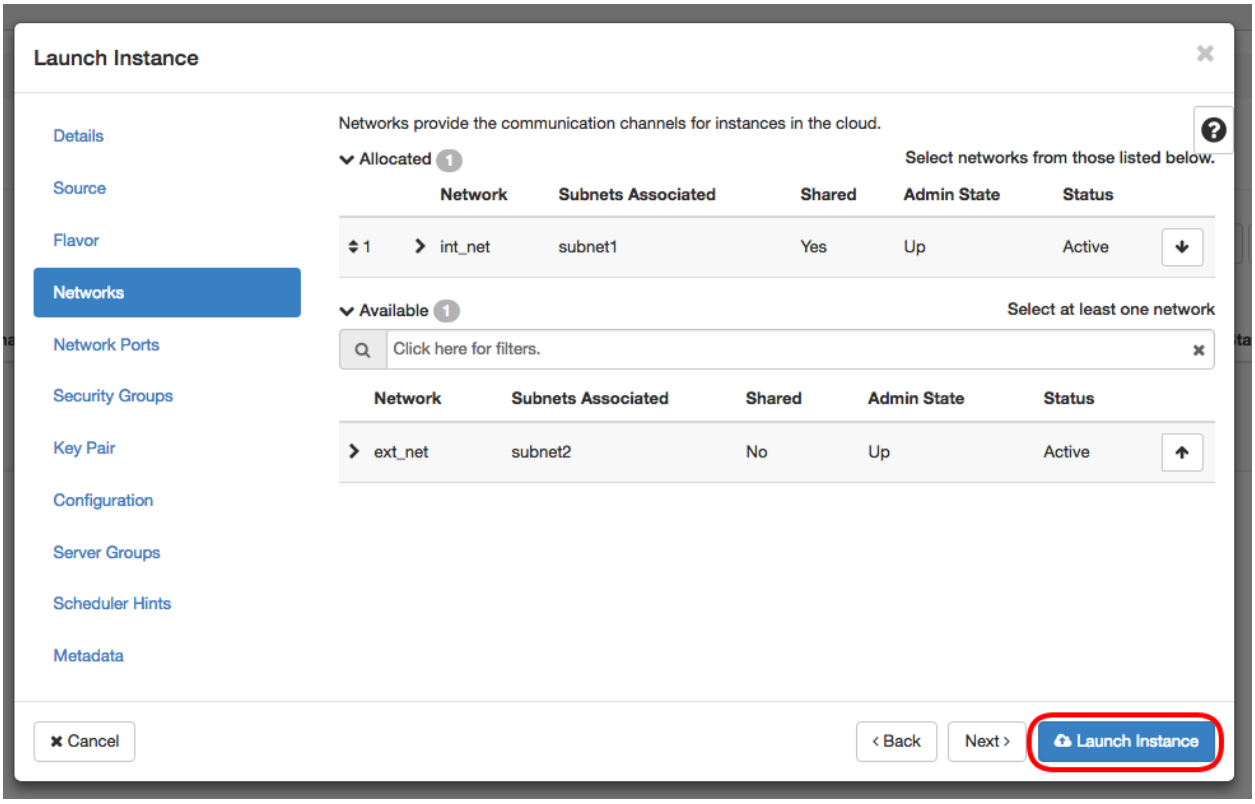




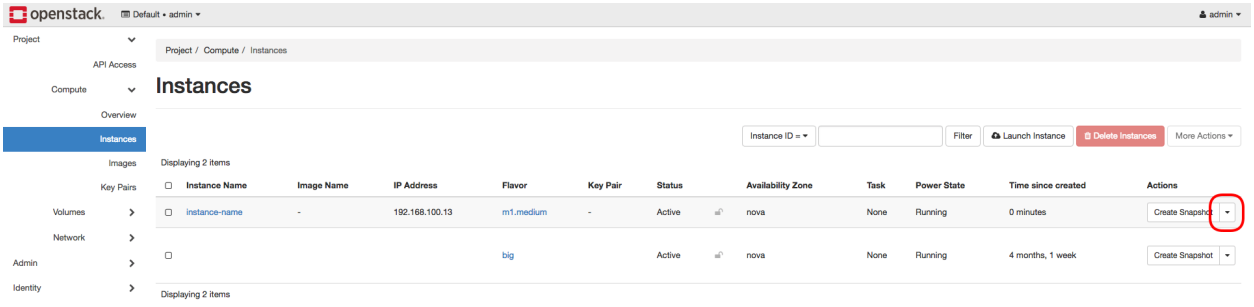
- Create or use ssh key to enable ssh access to the VM:



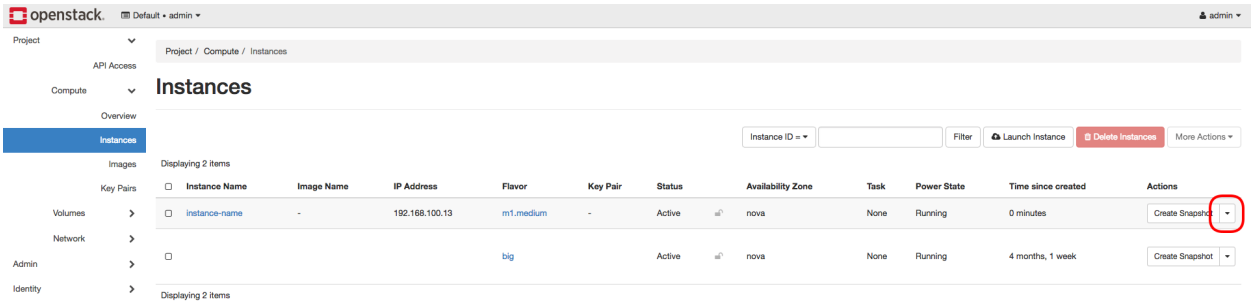
- In the end press on "Launch Instance" button at the bottom. This initiates the virtual machine deployment. It could take a while:

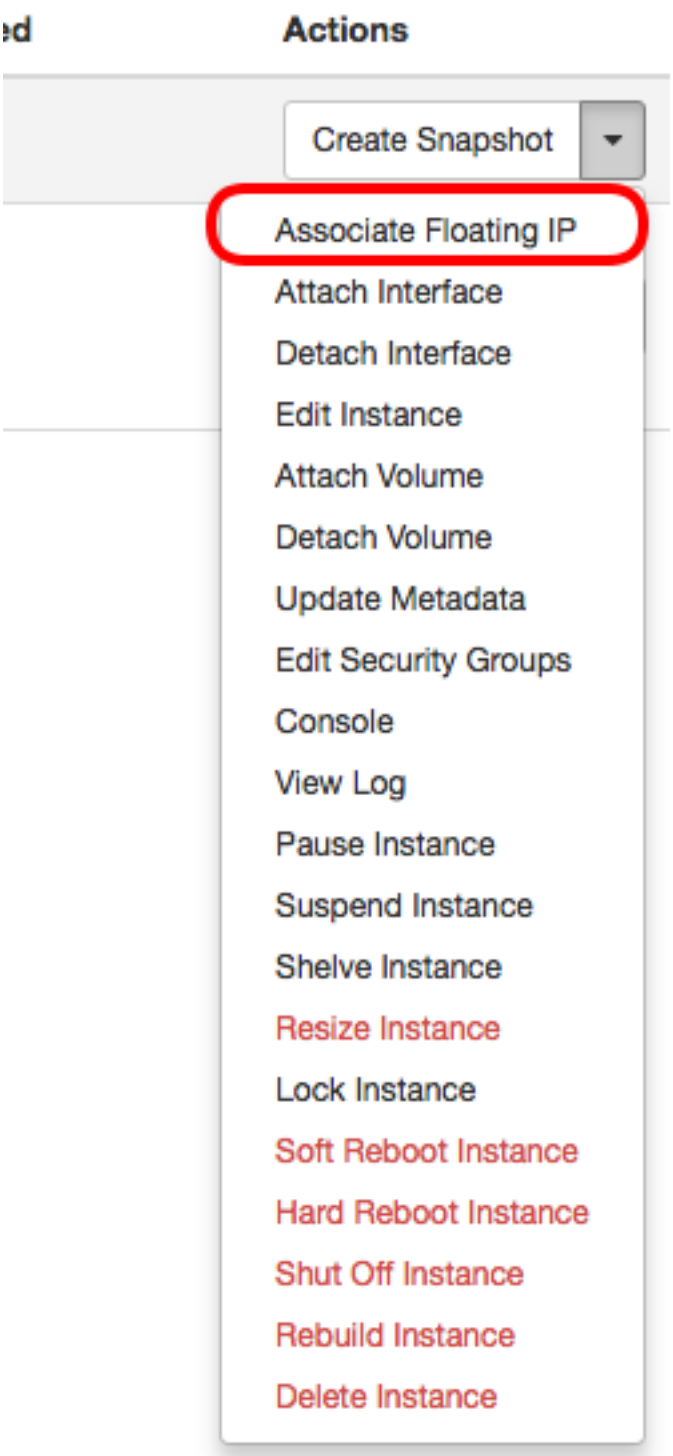


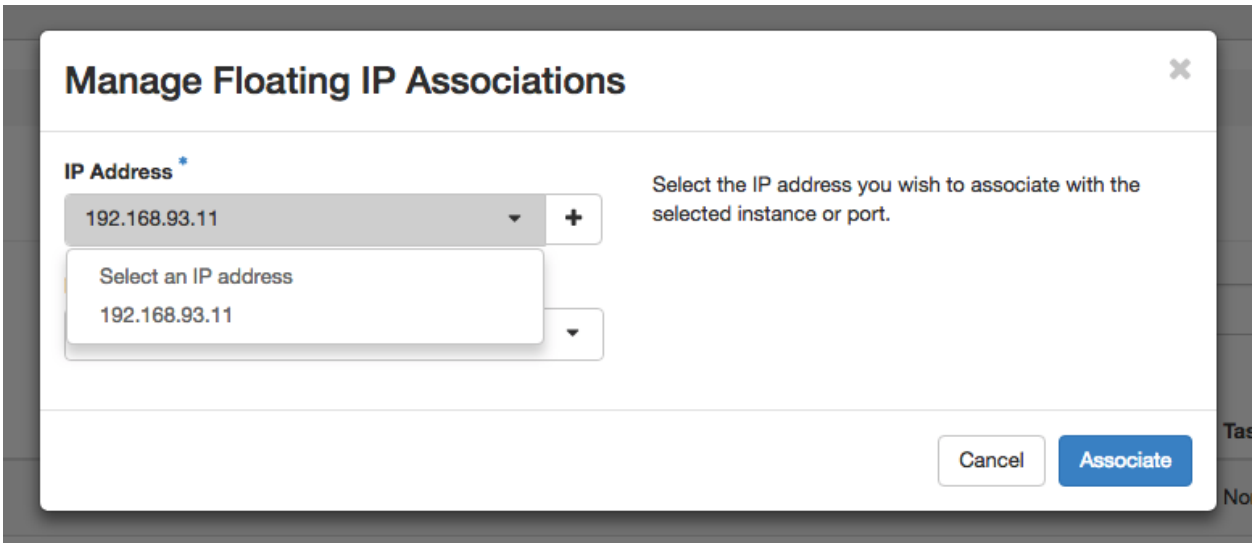
- When the Power State become “Running”, the virtual machine has been successfully deployed and is ready to be used:

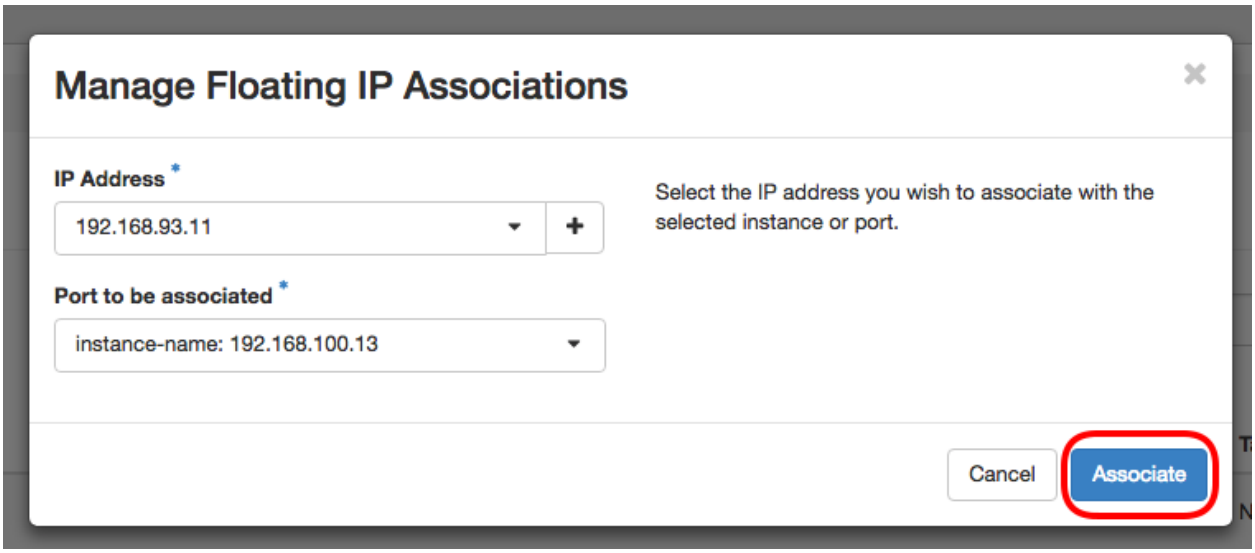


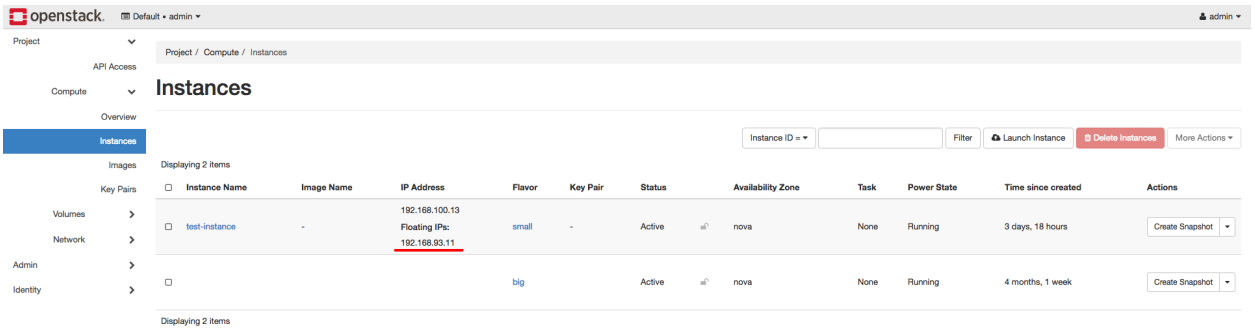
- Since the VM is deployed inside private network you will need to associate Floating IP from your network have the access:




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







- Now using any SSH client connect to the instance through VPN using the Floating IP address

Docker machine deployment

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

1. Instalation

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 provider 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...
```

(continues on next page)

(continued from previous page)

```
Detecting the provisioner...
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:~$
```

6.1.5 Access to the bare metal

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

CHAPTER 7

Developer Services

- *AIT cluster*

7.1 Share your Knowledge

Todo:

- Here is a space where anyone on SKA can share useful links to other docs. Please fill it in

CHAPTER 8

Share Your Knowledge

- *Share your Knowledge*

CHAPTER 9

Commitment to opensource

As defined in SKA software standard, SKA software development is committed to opensource, and an open source licensing model is always preferred within SKA software development.

Todo:

- Committment to opensource
- Exceptions

9.1 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.

9.2 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.

9.3 Programming Languages

9.3.1 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.

9.3.2 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.

9.4 Follow us on social media

- Facebook https://facebook.com/SKAtelescope
- Twitter https://twitter.com/SKA_telescope
- Instagram https://instagram.com/ska_telescope

CHAPTER 10

Follow Us

- *Follow us on social media*