developer.skatelescope.org Documentation

Release 0.8.1

Marco Bartolini

Jan 19, 2022

SDP DESIGN

The `ska-sdp-integration` repository integrates the components of the SKA Science Data Processor (SDP). It contains the Helm chart to deploy the SDP as a subsystem of the SKA, or stand-alone for testing.

This documentation also gives a overview of the design of the SDP. It describes the components that make up the system, and the software modules from which they are built.

CHAPTER

ONE

OVERVIEW

The SDP is part of the evolutionary prototype of the SKA software. This documentation gives an overview of the current design of the SDP system, which is a partial implementation of the SDP software architecture.

The SDP software architecture, and that of the rest of the system, is described in the Solution Intent in the SKA Community Confluence (wiki). The Solution Intent contains other information about the intended behaviour of the system, such as the high-level requirements. It also captures the behaviour of the system as currently implemented, including the outcome of tests against the requirements.

The components of the SDP system and the relationships between them are described in the *Components* section. The software modules used to build the components and deploy the SDP are described in the *Modules* section.

CHAPTER

TWO

COMPONENTS

Fig. 1: SDP components and the connections between them.

The diagram shows the components of the SDP system and the connections between them. The components, grouped by function, are as follows.

Execution Control:

- The **SDP Master Tango Device** is intended to provide the top-level control of SDP services. The present implementation does very little, apart from executing internal state transitions in response to Tango commands.
- The **SDP Subarray Tango Devices** control the processing associated with SKA Subarrays. When a Processing Block is submitted to SDP through one of the devices, it is added to the Configuration Database. During the execution of the Processing Block, the device publishes the status of the Processing Block through its attributes.
- The **Processing Controller** controls the execution of Processing Blocks. It detects them by monitoring the Configuration Database. To execute a Processing Block, it requests the deployment of the corresponding Workflow by creating an entry in the Configuration Database.
- The **Configuration Database** is the central store of configuration information in the SDP. It is the means by which the components communicate with each other.

Platform:

- The **Helm Deployer** is the service that the Platform uses to respond to deployment requests in the Configuration Database. It makes deployments by installing Helm charts (a collection of files that describe a related set of Kubernetes resources) into a Kubernetes cluster.
- **Kubernetes** is the underlying mechanism for making dynamic deployments of Workflows and Execution Engines.

Processing Block Deployment:

- A **Workflow** controls the execution of a Processing Block (in the architecture it is called the Processing Block Controller). Workflows connect to the Configuration Database to retrieve the parameters defined in the Processing Block and to request the deployment of Execution Engines.
- **Execution Engines** are the means by which Workflows process the data. They typically enable distributed execution of processing functions, although Workflows may use a single process as a serial Execution Engine.

CHAPTER

THREE

MODULES

Fig. 1: SDP software modules.

The SDP is built from software modules which produce a number of different types of artefacts. The components of the system are built as Docker images which are deployed on a Kubernetes cluster using a Helm chart. The Docker images depend on libraries containing common code. The diagram shows the SDP modules and the dependencies beween them.

The source code is hosted in the SKA Science Data Processor group in GitLab, in the following repositories:

- ska-sdp-integration

  Integration of components into the SDP system. Contains the Helm chart to deploy the SDP, and this documentation.

- ska-sdp-config

  Library providing the interface to the configuration database.

- ska-sdp-workflow

  Library providing the high-level interface for writing workflows.

- ska-sdp-lmc

  Tango devices for local monitoring and control.

- ska-sdp-proccontrol

  Processing controller.

- ska-sdp-helmdeploy

  Helm deployer.

- ska-sdp-console

  Console used to interact with the configuration database.

- ska-sdp-science-pipelines

  Science pipeline workflows.

- ska-sdp-helmdeploy-charts

  Charts used by the Helm deployer to deploy workflows and processing components/functions.

CHAPTER

FOUR

REQUIREMENTS

To run the SDP, you need to have Kubernetes and Helm installed.

4.1 Kubernetes

There are a number of ways to install and run Kubernetes in your development environment, including Docker Desktop, Minikube (recommended) and microk8s.

4.1.1 Docker Desktop

Docker Desktop for Windows and macOS includes a single-node Kubernetes installation. You need to activate it in the settings. You may need to increase the memory limit from its default value of 2 GB to allow the SDP to run.

4.1.2 Minikube

Minikube provides a way to run a single-node Kubernetes installation on Linux, macOS and Windows. It can use a number of different drivers to run Kubernetes in a virtual machine or container. On Windows, the Hyper-V hypervisor can be enabled in the settings, after which a reboot is required.

Minikube can be started with the default configuration by doing:

```
$ minikube start
```

If you need to increase the amount of memory that the Minikube VM uses, you can pass this as an option:

```
$ minikube start --memory='4096m'
```

Alternatively, you may configure this as a default by doing:

```
$ minikube config set memory 4096
```

If you are developing SDP components and you would like to build and test them in Minikube, you need to configure Docker to use the daemon inside the VM. This can be done by setting the environment variables:

```
$ minikube docker-env
$ eval $(minikube -p minikube docker-env)
```

4.1.3 Microk8s

Canonical supports microk8s for Linux, macOS and Windows.

4.2 Helm

Helm is available from most typical package managers, see Introduction to Helm.

4.3 K9s

K9s is terminal-based UI for Kubernetes clusters which provides a convenient interactive interface. It is not required to run the SDP, but it is recommended for its ease of use.

4.4 Commands Help Guide

To find out more about the available commands, here are some useful links:

- Helm - https://helm.sh/docs/helm/helm/
- Kubectl - https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands
- Docker - https://docs.docker.com/engine/reference/commandline/cli/

CHAPTER

FIVE

RUNNING THE SDP STAND-ALONE

Before running the SDP, your local development environment needs to be set up. Details can be found in the *requirements* section.

5.1 Create the namespace for SDP workflows

The SDP deploys its workflows and their execution engines into a separate Kubernetes namespace from the controllers. Before deploying the SDP you need to create this namespace, which by default is called `sdp`:

```
$ kubectl create namespace sdp
```

5.2 Deploying the SDP

Releases of the SDP Helm chart are published in the SKA artefact repository. To install the released version, you need to add this chart repository to helm:

```
$ helm repo add ska https://artefact.skao.int/repository/helm-internal
```

The chart can be installed with the command (assuming the release name is `test`):

```
$ helm install test ska/ska-sdp
```

You can watch the deployment in progress using `kubectl`:

```
$ kubectl get pod --watch
```

or the `k9s` terminal-based UI (recommended):

```
$ k9s
```

Wait until all the pods are running:

```
default      databaseds-tango-base-test-0    1/1          0 Running    172.17.0.12    ␣
↪m01   119s
default      sdp-console-0                   1/1          0 Running    172.17.0.15    ␣
↪m01   119s
default      sdp-etcd-0                      1/1          0 Running    172.17.0.6     ␣
↪m01   119s
default      sdp-helmdeploy-0                1/1          0 Running    172.17.0.14    ␣
↪m01   119s
```

(continues on next page)

(continued from previous page)

```
default      sdp-lmc-configuration-6vbtr     0/1          0 Completed  172.17.0.11    ␣
↪m01   119s
default      sdp-lmc-master-0                1/1          0 Running    172.17.0.9     ␣
↪m01   119s
default      sdp-lmc-subarray-01-0           1/1          0 Running    172.17.0.10    ␣
↪m01   119s
default      sdp-opinterface-0               1/1          0 Running    172.17.0.13    ␣
↪m01   119s
default      sdp-proccontrol-0               1/1          0 Running    172.17.0.4     ␣
↪m01   119s
default      sdp-wf-configuration-2hpdn      0/1          0 Completed  172.17.0.5     ␣
↪m01   119s
default      ska-tango-base-tangodb-0        1/1          0 Running    172.17.0.8     ␣
↪m01   119s
```

You can check the logs of pods to verify that they are doing okay:

```
$ kubectl logs <pod_name>
```

For example:

```
$ kubectl logs sdp-lmc-subarray-01-0
...
1|2021-05-25T11:32:53.161Z|INFO|MainThread|init_device|subarray.py#92|tango-device:test_
↪sdp/elt/subarray_1|SDP Subarray initialising
...
1|2021-05-25T11:32:53.185Z|INFO|MainThread|init_device|subarray.py#127|tango-device:test_
↪sdp/elt/subarray_1|SDP Subarray initialised
...
$ kubectl logs sdp-proccontrol-0
1|2021-05-25T11:32:32.423Z|INFO|MainThread|main_loop|processing_controller.py
↪#180||Connecting to config DB
1|2021-05-25T11:32:32.455Z|INFO|MainThread|main_loop|processing_controller.py
↪#183||Starting main loop
1|2021-05-25T11:32:32.566Z|INFO|MainThread|main_loop|processing_controller.py
↪#190||processing block ids []
...
```

If it looks like this, there is a good chance everything has been deployed correctly.

5.3 Testing it out

5.3.1 Connecting to the configuration database

The `ska-sdp` chart deploys a 'console' pod which enables you to interact with the configuration database. You can start a shell in the pod by doing:

```
$ kubectl exec -it sdp-console-0 -- bash
```

This will allow you to use the `ska-sdp` command:

```
# ska-sdp list -a
Keys with prefix /:
/master
/subarray/01
/workflow/batch:batch_imaging:0.1.0
/workflow/batch:batch_imaging:0.1.1
...
```

Which shows that the configuration contains the state of the Tango devices and the workflow definitions.

Details about the existing commands of the `ska-sdp` utility can be found in the CLI to interact with SDP section in the SDP Configuration Library documentation.

5.3.2 Starting a workflow

Assuming the configuration is prepared as explained in the previous section, we can now add a processing block to the configuration:

```
# ska-sdp create pb <workflow_type>:<workflow_id>:<workflow_version>
```

For example

```
# ska-sdp create pb batch:test_dask:0.2.6
Processing block created with pb_id: pb-sdpcli-20210802-00000
```

The processing block is created with the `/pb` prefix in the configuration:

```
# ska-sdp list -v pb
Keys with prefix /pb:
/pb/pb-sdpcli-20210802-00000 = {
  "dependencies": [],
  "id": "pb-sdpcli-20210802-00000",
  "parameters": {},
  "sbi_id": null,
  "workflow": {
    "id": "test_dask",
    "type": "batch",
    "version": "0.2.6"
  }
}
/pb/pb-sdpcli-20210802-00000/owner = {
  "command": [
    "test_dask.py",
    "pb-sdpcli-20210802-00000"
  ],
  "hostname": "proc-pb-sdpcli-20210802-00000-workflow-mvdnk",
  "pid": 1
}
/pb/pb-sdpcli-20210802-00000/state = {
  "deployments": {
    "proc-pb-sdpcli-20210802-00000-dask-1": "RUNNING",
    "proc-pb-sdpcli-20210802-00000-dask-2": "RUNNING"
  },
```

(continues on next page)

(continued from previous page)

```
  "resources_available": true,
  "status": "RUNNING"
}
```

The processing block is detected by the processing controller which deploys the workflow. The workflow in turn deploys the execution engines (in this case, Dask). The deployments are requested by creating entries with `/deploy` prefix in the configuration, where they are detected by the Helm deployer which actually makes the deployments:

```
# ska-sdp list -v deployment
Keys with prefix /deploy:
/deploy/proc-pb-sdpcli-20210802-00000-dask-1 = {
  "args": {
    "chart": "dask",
    "values": {
      "image": "artefact.skao.int/ska-sdp-wflow-test-dask:0.2.6",
      "worker.replicas": 2
    }
  },
  "id": "proc-pb-sdpcli-20210802-00000-dask-1",
  "type": "helm"
}
/deploy/proc-pb-sdpcli-20210802-00000-dask-2 = {
  "args": {
    "chart": "dask",
    "values": {
      "image": "artefact.skao.int/ska-sdp-wflow-test-dask:0.2.6",
      "worker.replicas": 2
    }
  },
  "id": "proc-pb-sdpcli-20210802-00000-dask-2",
  "type": "helm"
}
/deploy/proc-pb-sdpcli-20210802-00000-workflow = {
  "args": {
    "chart": "workflow",
    "values": {
      "env": {
        "SDP_CONFIG_HOST": "sdp-etcd-client.default.svc.cluster.local",
        "SDP_HELM_NAMESPACE": "sdp"
      },
      "pb_id": "pb-sdpcli-20210802-00000",
      "wf_image": "artefact.skao.int/ska-sdp-wflow-test-dask:0.2.6"
    }
  },
  "id": "proc-pb-sdpcli-20210802-00000-workflow",
  "type": "helm"
}
```

The deployments associated with the processing block have been created in the `sdp` namespace, so to view the created pods we have to ask as follows (on the host):

```
$ kubectl get pod -n sdp
```

(continues on next page)

(continued from previous page)

```
NAME                                                        READY   STATUS   ␣
↪RESTARTS   AGE
proc-pb-sdpcli-20210802-00000-dask-1-scheduler-6d84584948-t5kzz   1/1     Running   0   ␣
↪       30s
proc-pb-sdpcli-20210802-00000-dask-1-worker-5b568bb45b-7jcsr      1/1     Running   0   ␣
↪       30s
proc-pb-sdpcli-20210802-00000-dask-1-worker-5b568bb45b-rfxvs      1/1     Running   0   ␣
↪       30s
proc-pb-sdpcli-20210802-00000-dask-2-scheduler-5f6dfc6d56-r4rt5   1/1     Running   0   ␣
↪       29s
proc-pb-sdpcli-20210802-00000-dask-2-worker-78cd65b78f-dqkp4      1/1     Running   0   ␣
↪       29s
proc-pb-sdpcli-20210802-00000-dask-2-worker-78cd65b78f-jxbjz      1/1     Running   0   ␣
↪       29s
proc-pb-sdpcli-20210802-00000-workflow-mvdnk                      1/1     Running   0   ␣
↪       33s
```

5.3.3 Cleaning up

Finally, let us remove the processing block from the configuration (in the SDP console shell):

```
# ska-sdp delete pb pb-sdpcli-20210802-00000
/pb/pb-sdpcli-20210802-00000
/pb/pb-sdpcli-20210802-00000/state
Deleted above keys with prefix /pb/pb-sdpcli-20210802-00000.
```

If you re-run the commands from the last section you will notice that this correctly causes all changes to the cluster configuration to be undone as well.

5.4 Accessing the Tango interface

By default, the `ska-sdp` chart does not deploy the iTango shell pod from the `ska-tango-base` chart. To enable it, you can upgrade the release with:

```
$ helm upgrade test ska/ska-sdp --set ska-tango-base.itango.enabled=true
```

Then you can start an iTango session with:

```
$ kubectl exec -it ska-tango-base-itango-console -- itango3
```

You should be able to list the Tango devices:

```
In [1]: lsdev
Device                                  Alias                   Server                  ␣
↪    Class
----------------------------------------- ------------------------ ----------------------
↪--- --------------------
test_sdp/elt/master                                              SDPMaster/0             ␣
↪    SDPMaster
test_sdp/elt/subarray_1                                          SDPSubarray/01          ␣
↪    SDPSubarray
```

(continues on next page)

(continued from previous page)

```
sys/access_control/1                                    TangoAccessControl/1 ␣
↪    TangoAccessControl
sys/database/2                                          DataBaseds/2         ␣
↪    DataBase
sys/rest/0                                              TangoRestServer/rest ␣
↪    TangoRestServer
sys/tg_test/1                                           TangoTest/test       ␣
↪    TangoTest
```

This allows direct interaction with the devices, such as querying and changing attributes and issuing commands:

```
In [2]: d = DeviceProxy('test_sdp/elt/subarray_1')

In [3]: d.state()
Out[3]: tango._tango.DevState.OFF

In [4]: d.On()

In [5]: d.state()
Out[5]: tango._tango.DevState.ON

In [6]: d.obsState
Out[6]: <obsState.EMPTY: 0>

In [7]: config_sbi = '''
   ...: {
   ...:   "id": "sbi-test-20210525-00000",
   ...:   "max_length": 21600.0,
   ...:   "scan_types": [
   ...:     {
   ...:       "id": "science",
   ...:       "channels": [
   ...:         {"count": 5, "start": 0, "stride": 2, "freq_min": 0.35e9, "freq_max": 0.
↪358e9, "link_map": [[0,0], [200,1]]}
   ...:       ]
   ...:     }
   ...:   ],
   ...:   "processing_blocks": [
   ...:     {
   ...:       "id": "pb-test-20210525-00000",
   ...:       "workflow": {"type": "realtime", "id": "test_realtime", "version": "0.2.4
↪"},
   ...:       "parameters": {}
   ...:     },
   ...:     {
   ...:       "id": "pb-test-20210525-00001",
   ...:       "workflow": {"type": "realtime", "id": "test_receive_addresses", "version
↪": "0.3.6"},
   ...:       "parameters": {}
   ...:     },
   ...:     {
   ...:       "id": "pb-test-20210525-00002",
```

(continues on next page)

(continued from previous page)

```
    ...:       "workflow": {"type": "batch", "id": "test_batch", "version": "0.2.4"},
    ...:       "parameters": {},
    ...:       "dependencies": [
    ...:         {"pb_id": "pb-test-20210525-00000", "type": ["visibilities"]}
    ...:       ]
    ...:     },
    ...:     {
    ...:       "id": "pb-test-20210525-00003",
    ...:       "workflow": {"type": "batch", "id": "test_dask", "version": "0.2.5"},
    ...:       "parameters": {},
    ...:       "dependencies": [
    ...:         {"pb_id": "pb-test-20210525-00002", "type": ["calibration"]}
    ...:       ]
    ...:     }
    ...:   ]
    ...: }
    ...: '''

In [8]: d.AssignResources(config_sbi)

In [9]: d.obsState
Out[9]: <obsState.IDLE: 0>

In [10]: d.Configure('{"scan_type": "science"}')

In [11]: d.obsState
Out[11]: <obsState.READY: 2>

In [12]: d.Scan('{"id": 1}')

In [13]: d.obsState
Out[13]: <obsState.SCANNING: 3>

In [14]: d.EndScan()

In [15]: d.obsState
Out[15]: <obsState.READY: 2>

In [16]: d.End()

In [17]: d.obsState
Out[17]: <obsState.IDLE: 0>

In [18]: d.ReleaseResources()

In [19]: d.obsState
Out[19]: <obsState.EMPTY: 0>

In [20]: d.Off()

In [21]: d.state()
Out[21]: tango._tango.DevState.OFF
```

More details about each of the SDP Subarray commands can be found here

5.5 Removing the SDP

To remove the SDP deployment from the cluster, do:

```
$ helm uninstall test
```

5.6 Developing the SDP chart

If you want to install the chart from the source code in the SDP Integration repository, for instance if you are developing a new version, then you can do it like this:

```
$ helm install --dependency-update test charts/ska-sdp
```

The `--dependency-update` flag downloads the `ska-tango-base` chart on which the `ska-sdp` chart depends.

5.7 Developing SDP Workflows

Instructions on how to develop and test SDP workflows can be found in the Science Pipeline Workflows documentation.

CHAPTER

SIX

RUNNING THE SDP IN SKAMPI

The Science Data Processor (SDP) is integrated with the other telescope subsystems as part of the SKA evolutionary prototype, also known as the *Minimum Viable Product* (MVP). The integration is done in the SKA MVP Prototype Integration (SKAMPI) repository.

Instructions for installing and running the MVP can be found in the SKAMPI documentation. *TODO: this needs exact, better links (probably need to wait for skampi docs to be updated)*

The default namespace into which SKAMPI deploys (in a non-GitLab CI environment) is `integration`. You can change this by setting the `KUBE_NAMESPACE` and `KUBE_NAMESPACE_SDP` environment variables before deploying SKAMPI. E.g:

```
$ export KUBE_NAMESPACE=test-skampi-deployment
$ export KUBE_NAMESPACE_SDP=test-skampi-deployment-sdp
```

If you are deploying on a shared machine, make sure your namespace doesn't clash with existing SKAMPI deployments.

SDP is integrated into the SKAMPI charts for SKA-Mid and SKA-Low. All of the standard SDP pods will start up upon SKAMPI deployment.

6.1 Interacting with SDP within SKAMPI

6.1.1 Using the SDP console

Follow the "Testing it out" steps in *Running the SDP stand-alone*. Remember to specify the namespace in which the pod is running when you want to access one. E.g. to start the console pod, run:

```
$ kubectl -n <namespace> exec -it sdp-console-0 -- bash
```

6.1.2 Using the iTango pod

If your deployment doesn't have the `ska-tango-base-itango-console` running, add the following to the `pipeline.yml` file in the SKAMPI root directory:

```
tango-base:
  itango:
    enabled: true
```

And then upgrade your SKAMPI installation via its Makefile:

```
$ make upgrade-chart VALUES=pipeline.yml
```

Once you have a running iTango console, follow the steps to test it out in *Running the SDP stand-alone* at "Accessing the Tango interface" section. Again, you will have to specify the namespace you deployed in, when interacting with the iTango pod.

6.1.3 Using the Observation Execution Tool

Observation Execution Tool (OET), is an application, which provides on-demand Python script execution for the SKA. For interactive telescope control (and hence SDP control), and experimentation, one can use the OET Jupyter Notebooks and OET scripts:

- OET Jupyter Notebooks
- OET scripts

Using the OET Jupyter Notebooks

An OET Jupyter Notebook server is automatically deployed when SKAMPI is started. The pod running it has the name `oet-jupyter-test-...`, where `...` will be a multi-character string, specific to the deployment.

The webserver running in this pod can be accessed via a web link of the following structure:

```
http://<ingress_host>/<namespace>/jupyter
```

`<namespace>` is the `KUBE_NAMESPACE`, while the `<ingress_host>` is either an environment variable called `INGRESS_HOST`, or the default one can be found in the `Makefile` of SKAMPI under the same variable name. You can also find this value by running this command:

```
$ kubectl describe ingress <pod_name> -n <namespace>
```

Depending on how you access the website (i.e. with port forwarding or directly), you may need to replace the `<ingress_host>` with your localhost or similar. More information on how to access the Notebooks on SKAMPI (including the required password) can be found in the OET docs.

You can access the existing Notebooks in `scripts/notebooks`. Based on these examples, you may also create your own version of an SKA control Notebook. You may also use the OET scripts as a starting point for your own development. For example, the steps in the script at Control using static JSON can be copy-pasted into a Jupyter Notebook. You will need to specify a JSON file, which contains the necessary configuration string to control the subarray device and hence, SDP. Example JSON files can be found in the OET Scripts repository. Here is an example how you can update the lines which require JSON files:

```
# Allocate resources, provide a path to a file with allocation JSON
subarray.allocate_from_file('../data/example_allocate.json')

# Configure sub-array, provide a path to a file with configuration JSON
subarray.configure_from_file('../data/example_configure.json', scan_duration=10.0)
```

The above assumes that your Notebook is started from the `scripts/notebooks` directory.

Using the OET Rest Client

The OET Rest Client provides a command line interface to communicate with a backend, which allows one to run SKA control scripts.

The easiest to do this is through the terminal window of the Jupyter server deployed in SKAMPI. Please follow the above links to learn more about OET and how to use Python scripts to control an SKA telescope via this interface.

CHAPTER

SEVEN

HELM CHART

This is a summary of the Helm chart parameters that can be used the customise the SDP deployment. The current default values can be found in the chart's values file.

7.1 Configuration database

The configuration database is implemented on top of etcd.

Parameter	Description	Default
`etcd.image`	etcd container image	`quay.io/coreos/etcd`
`etcd.version`	etcd container version	`3.3.25`
`etcd.imagePullPolicy`	etcd container image pull policy	`IfNotPresent`
`etcd.useOperator`	Use etcd-operator to deploy the etcd cluster (deprecated)	`false`
`etcd.maxTxnOps`	Maximum number of operations per transaction (not supported by etcd-operator)	`1024`

7.2 Console

The console provides a command-line interface to monitor and control the SDP by interacting with the configuration database.

Parameter	Description	Default
`console.enabled`	Enable the console	`true`
`console.image`	Console container image	`artefact.skao.int/ska-sdp-console`
`console.version`	Console container version	See values file
`console.imagePullPolicy`	Console container image pull policy	`IfNotPresent`

7.3 Operator web interface

The operator web interface can be used to control and monitor the SDP by interacting with the configuration database.

Parameter	Description	Default
`opinterface.enabled`	Enable the operator web interface	`true`
`opinterface.image`	Operator web interface container image	`artefact.skao.int/ska-sdp-opinterface`
`opinterface.version`	Operator web interface container version	See values file
`opinterface.imagePullPolicy`	Operator web interface container image pull policy	`IfNotPresent`

7.4 Processing controller

Parameter	Description	Default
`proccontrol.image`	Processing controller container image	`artefact.skao.int/ska-sdp-proccontrol`
`proccontrol.version`	Processing controller container version	See values file
`proccontrol.imagePullPolicy`	Processing controller container image pull policy	`IfNotPresent`

7.5 Workflows

Workflow definitions to be used by SDP. These map the workflow type, ID and version to a container image. By default the definitions are read from the science pipeline workflows repository in GitLab. A different URL may be specified. Alternatively a list of workflow definitions can be passed to the chart.

Parameter	Description	Default
`workflows.url`	URL from which to read the workflow definitions	`https://gitlab.com/ska-telescope/sdp/ska-sdp-science-pipelines/-/raw/master/workflows.json`
`workflows.definitions`	List of workflow definitions. If present, used instead of the URL. See the example below	Not set

Example of workflow definitions in a values file:

```
workflows:
  definitions:
  - type: realtime
    id: test_realtime
    version: 0.2.2
    image: artefact.skao.int/ska-sdp-wflow-test-batch:0.2.2
  - type: batch
    id: test_realtime
    version: 0.2.2
    image: artefact.skao.int/ska-sdp-wflow-test-realtime:0.2.2
```

7.6 Helm deployer

Parameter	Description	Default
`helmdeploy.image`	Helm deployer container image	`artefact.skao.int/ska-sdp-helmdeploy`
`helmdeploy.version`	Helm deployer container version	See values file
`helmdeploy.imagePullPolicy`	Helm deployer container image pull policy	`IfNotPresent`
`helmdeploy.namespace`	Namespace for SDP dynamic deployments	`sdp`
`helmdeploy.prefix`	Prefix for Helm release names	`''`
`helmdeploy.createNamespace`	Create the namespace for dynamic deployments	`false`
`helmdeploy.createClusterRole`	Create a cluster role to allow dynamic deployments to create persistent volumes	`false`
`helmdeploy.chart_repo.url`	Chart repository URL	`https://gitlab.com/ska-telescope/sdp/ska-sdp-helmdeploy-charts/-/raw/master/chart-repo/`
`helmdeploy.chart_repo.refresh`	Chart repository refresh interval (in seconds)	`300`

7.7 LMC (Tango devices)

Parameter	Description	Default
`lmc.enabled`	Enable the LMC. If set to `false`, the SDP will run in headless mode	`true`
`lmc.image`	LMC container image	`artefact.skao.int/ska-sdp-lmc`
`lmc.version`	LMC container version	See values file
`lmc.imagePullPolicy`	LMC container image pull policy	`IfNotPresent`
`lmc.allCommandsHaveArgument`	Enable all Tango device commands to receive a transaction ID	`false`
`lmc.prefix`	Telescope prefix for Tango device names (e.g. `low` or `mid`)	`test`
`lmc.nsubarray`	Number of subarrays to deploy	1

7.8 Tango infrastructure

Parameters for the ska-tango-base subchart and Tango dsconfig. The ska-tango-base subchart must be enabled to support the Tango devices when running the SDP stand-alone.

Parameter	Description	Default
`ska-tango-base.enabled`	Enable the ska-tango-base subchart	`true`
`ska-tango-base.itango.enabled`	Enable the itango console in the ska-tango-base subchart	`false`
`dsconfig.image.*`	Tango dsconfig container image settings	See values file

7.9 Proxy settings

Proxy settings are applied to the components that retrive configuration data via HTTPS: the workflow definitions and the Helm charts.

Parameter	Description	Default
`proxy.server`	Address of proxy server	Not set
`proxy.noproxy`	List of addresses or subnets for which the proxy should not be used	Not set

Example of proxy settings in a values file:

```
proxy:
  server: http://proxy.mydomain
  noproxy:
  - 192.168.0.1
  - 192.168.0.2
```

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search