SP-108 “Onboard PST Code” Report

The purpose of this document is to demonstrate that we have met the acceptence criteria for feature SP-108. The first part illustrates the way DSPSR integrates into the SKA Gitlab CI infrastructure, and the second part indicates where DSPSR currently stands with respect to SKA C++ coding recommendations.

DSPSR and SKA Gitlab CI

The first part of the acceptance criteria for SP-108 states that “DSPSR [...] can build, test, and deploy built docker images in the SKA Gitlab CI environment. Moreover, a CUDA-enabled DSPSR can build and deploy in the SKA Gitlab CI.” We have set up DSPSR, and an accompanying test project `dspsr-test`, to do these tasks in an automated fashion. Figure 1 shows a screenshot of the entire DSPSR CI pipeline.

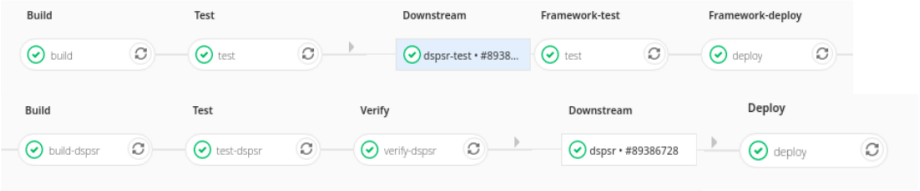

Figure 1: Screenshot of DSPSR Gitlab page showing successful CI pipeline

The main DSPSR repository is hosted on Sourceforge, and mirrored on the SKA Gitlab. By mirroring the repository, we allow existing DSPSR community development to continue unhindered. The Gitlab mirror periodically checks for new commits in the main repository, and it also supports manually triggering updates. DSPSR uses the Gitlab CI infrastructure to build, test and deploy in an automated fashion. The `build` and `test` stages use a Docker image that supplies a build environment that contains all of DSPSR's dependencies. This Docker image, `dspsr-build` is hosted on the SKA Nexus Docker registry. For more information on this build layer, see the section on Docker images. The `deploy` stage uses docker in docker to push Docker images of newest commits to the SKA Nexus Docker repository. Figure 2 shows the Docker image built in this pipeline, as hosted on the Nexus Docker registry.

The `build` stage of the DSPSR CI pipeline uses nvidia's CUDA Docker image supplied headers and libraries to compile any CUDA code in the DSPSR code base. Given that no GPU-enabled runners are active on the SKA Gitlab, the `test` stage does not run tests that require the GPU. In the event that such a runner becomes available, running tests that use the GPU simply requires modifying command line arguments passed to the DSPSR testing executable.

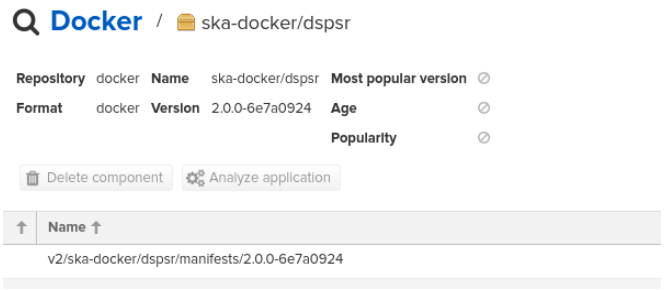

Figure 2: Screenshot of a DSPSR Docker image on the SKA Nexus Docker registry

At the moment, the main DSPSR repository does not run an extensive suite of tests. Instead, it triggers another repository, `dspsr-test`, to run a more extensive set of integration tests and a series of regression tests after the `test` stage. These regression tests attempt to validate new DSPSR commits by comparing the output of the main `dspsr` executable against some reference DSPSR commit. Importantly, `dspsr-test` uses DSPSR test data files. See gDMCP Test Data for more information. Given that these regression tests run in the CI environment, they are not setup to run on a GPU. Once all the stages in `dspsr-test` have successfully completed, `dspsr-test` triggers the final stage of the DSPSR CI pipeline. The `deploy` stage of the DSPSR project builds a CUDA-enabled DSPSR image. This CUDA-enabled image has been tested in an environment with access to a GPU.

Figure 3 shows the CI scheme for DSPSR and `dspsr-test`. All the stages use the `dspsr-build` docker image, with the exception of the "framework-deploy" and "deploy" stages of the `dspsr-test` and DSPSR pipelines, respectively. For a more detailed description of all the stages of the `dspsr-test` see `dspsr-test` CI stages

gDMCP Test Data

An important aspect of the tests run in the `dspsr-test` pipeline is the use of test data files. We have set up a public dataset for these files with gDMCP, an astronomical data hosting service. The `dspsr-test` Python codebase contains a wrapper for the CKAN API that gDMCP uses. Moreover, `dspsr-test` builds a Docker image, `dspsr-test` that allows for downloading gDMCP data without installing Python dependencies.

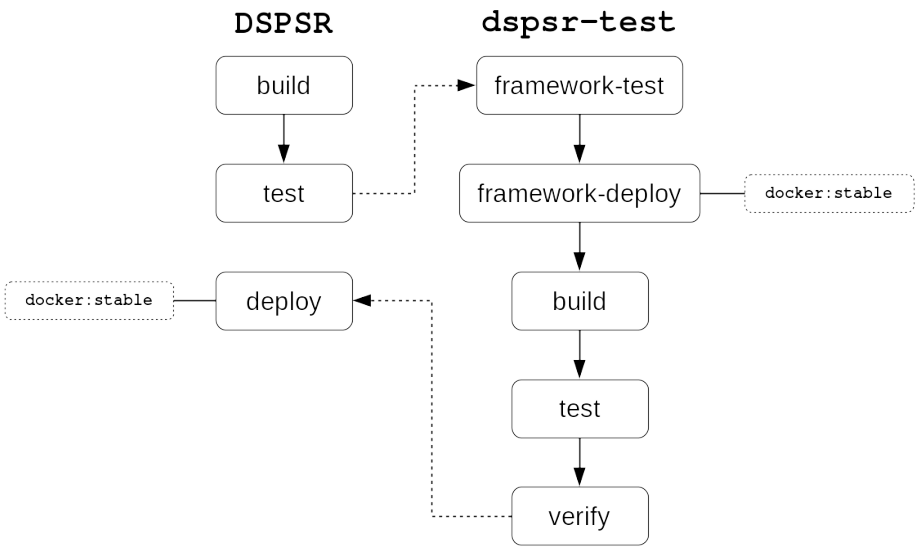

Figure 3: CI pipeline for DSPSR and `dspsr-test`. Boxes with solid lines represent individual pipeline stages. Solid arrows between boxes represent intra pipeline triggers. Dashed arrows represent inter pipeline triggers. The dashed boxes indicate that a given stage uses a docker image other that `dspsr-build`.

Docker images

DSPSR and `dspsr-test` use the `dspsr-build` Docker image. `dspsr-build` is deployed in an automated fashion from the `dspsr-docker` repository. The `dspsr-build` Docker image uses a multi-stage build, drawing from the `psrchive` and `psrdada` images. Figure 4 shows the dependency tree of the `dspsr-build` Docker image.

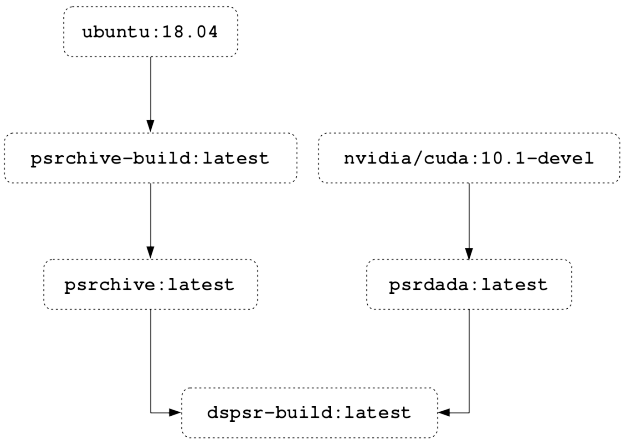

Figure 4: Dependency tree of `dspsr-build` Docker image

`dspsr-test` CI Stages

`dspsr-test` currently comprises five CI stages:

- `framework-test`: Unit tests of Python code used to run regression tests.
- `framework-deploy`: Builds a Docker image of `dspsr-test` repository and pushes to Docker Hub.
- `build`: Builds reference DSPSR commit, and newest commit on master branch of DSPSR. Saves installation locations of both builds, and the source tree of the newest commit.
- `test`: Downloads gDMCP test data. Runs C++ unit and integration tests using newest DSPSR build.
- `verify`: Runs Python regression tests using the two DSPSR builds from the `build` stage. If these tests pass, trigger `deploy` stage of main DSPSR CI pipeline.

DSPSR and SKA C++ coding recommendations

Part of the acceptance criteria for SP-108 is that "DSPSR adheres to SKA C++ coding practices to an agreed upon acceptable level". Given that DSPSR is has

long been a community run piece of software, our main goal for this feature was to determine where DSPSR stands with regards to SKA coding recommendations, and to indicate where any progress is being made in better aligning with these standards.

Build and Test Tools

The following list indicates how DSPSR is tracking with regards to each of the build and test tools used in the sample C++ project on the SKA developer portal.

- *Continuous integration (CI) setup using Gitlab*

  DSPSR currently builds, tests, and deploys using the Gitlab CI environment. Moreover, we have an additional Gitlab project `dspsr-test` devoted to performing commit level regression tests.

- *CMake as a build tool*

  DSPSR currently uses GNU Autotools as a configuration and build tool. We anticipate it would take around four story points to get CMake working with DSPSR. We estimate that half of this time would be spent verifying that both systems (CMake and Autotools) produce equivalent build products.

- *GoogleTest test framework*

  DSPSR currently uses Catch2 for testing. Given that the current testing codebase is rather small, we anticipate that it would take two story points to move tests to GoogleTest.

- *`gcov` to measure test coverage*

  DSPSR does not currently use any test coverage tool. Our estimates indicate that it would take roughly three story points to get `gcov` working with DSPSR.

Coding Practices

This section indicates the status of DSPSR with respect to each of the coding standards laid out by the sample projecty on the SKA developer portal.

- *Headers and namespaces*

  In the example C++ project on the SKA developer portal, namespaces mirror the directory structure of the project. Moreover, any SKA specific functionality is found in a corresponding `ska` directory and namespace. DSPSR decidedly does not following this paradigm. Implementing this sort of scheme would require introducing major changes to the public facing DSPSR API, in addition to refactoring most of the DSPSR source code.

- *Test code location*

  The example C++ project on the SKA developer portal bundles test code along with source code. DSPSR currently keeps test code in a separate `test` subdirectoy at the top level of the source tree.

- *Dependency management*

  The SKA developer portal page on C++ development proposes a scheme for listing dependencies in a `dependencies.txt` file. The format of this file is designed with the CMake build system in mind. DSPSR, along with other pulsar processing software use a custom dependency management layer to find and build any external dependencies. Aligning more closely to the proposed dependency management scheme would have to be part of adding CMake as a DSPSR build system. Given that DSPSR requires specific dependency configurations (such as a single precision FFTW build), adjusting the way DSPSR finds and builds external software could prove difficult.

- *cplusplus Core Guidelines*

  The cplusplus Core Guidelines are a set of style and coding conventions for C++. The clang C++ linter, Clang-Tidy, can be configured to use these guidelines for linting. Linting has never been part of the DSPSR development process, so adjusting existing code to fit within these guidelines would be a non trivial task. However, we could start using this tool for any new code that is introduced to the existing codebase.