


DMTN-023: Pipeline Command-Line Drivers

Jim Bosch

Latest Revision: 2016-06-21

Introduction

This document provides a brief tutorial for using the LSST Software Stack's main command-line drivers. These form a sequence of high-level pipelines that can be used to process data from raw images to multi-band catalogs derived from coadds. The pipeline is currently very much at a prototype stage; the final LSST pipelines will be significantly more complex, and there will be no need to manually execute several steps in order to run it. But even in its current form, the pipeline is quite sophisticated, and it is already being used as the official pipeline of the Hyper Suprime-Cam (HSC) survey on Subaru.

Using these pipelines requires an *obs* package that has been specialized for the instrument that produced the data. These packages provide information about camera geometry, detrending, filters, file formats, directory structure, and everything else that makes a camera unique. Working *obs* packages already exist for HSC and Suprime-Cam (*obs_subaru*), SDSS (*obs_sdss*), CFHT's Megacam (*obs_cfht*), CTIO's DECam (*obs_decam*), and LSST simulations (*obs_lsstSim*). Creating a new *obs* package is a fair amount of work  and it's well beyond the scope of this tutorial. We'll be using *obs_subaru* in this tutorial, as it's currently the most rigorously tested of the *obs* packages, and we'll specifically refer to data that is available in the *ci_hsc* package, though the same commands (with different image IDs) should work on any dataset.

This document is intended to be read as a tutorial, not a reference – some features relevant to all command-line scripts are described in only one of the sections below, as it's expected that the reader will be going through all of them.

Data Repository Setup

We will assume that the raw data and master calibration frames (e.g. flats) are both already available. Most *obs* packages provide a way to build master calibration frames, but those haven't yet been standardized, and *ci_hsc* already includes everything we'll need for later processing. We'll assume the location of *ci_hsc* is in the environment variable `$CI_HSC_DIR`.



We'll start by creating a `DATA` directory, which will be the root of what we call a *data repository*:



```
$ mkdir DATA
```

The files within this directory will be managed by an object called the *butler* (`lsst.daf.persistence.Butler`), which abstracts all of our I/O; under normal circumstances, files and directories in a data repository should only be accessed or modified using the butler. The structure of the data repository is defined by another class called a *mapper*. Most mappers are defined in an *obs* package, which lets us use the native organization for each instrument (at least for raw data). To tell the butler which mapper a data repository uses, we create a `_mapper` file in the root of the data repository:

```
$ echo "lsst.obs.hsc.HscMapper" > DATA/_mapper
```



We can then *ingest* the raw images into the data repository, using the `ingestImages.py` script (implemented in `lsst.pipe.tasks.IngestImagesTask`):

```
$ ingestImages.py DATA $CI_HSC_DIR/raw/*.fits --mode=link
```



This adds symlinks for every file in the raw directory to the appropriate (butler-managed) location in the DATA directory; you can also use other `--mode` options to move, copy, or do nothing (if the files are already in the right place). In addition, this creates a *registry*: a database of all the raw images in the repository.



Calibration frames are typically stored in a separate data repository, and `ci_hsc` already contains a complete `ci_hsc` directory. We could just use this as-is, by passing `--calib=$CI_HSC_DIR/CALIB` to all of the downstream pipelines, but it will be easier to just create a symlink from this directory into our data repository:

```
$ cd DATA
```

```
$ ln -s $CI_HSC_DIR/CALIB .
```



This location will be automatically searched by pipelines when looking for calibration data. You can ignore all of the warnings you may see in the logs about failures to find calibration registries in other locations.

Some of our processing steps require an external reference catalog, which is currently provided by an `astrometry_net_data` package that must be set up using `EUPS` (the same system used to set up and declare LSST software versions). `ci_hsc` includes such a package. Before



first use, it must be declared:

```
$ eups declare astrometry_net_data sdss-dr9-fink-v5b+ci_hsc \  
-m none -r $SCI_HSC_DIR/sdss-dr9-fink-v5b
```



and then (like any EUPS product) it must set up every time you open a new shell:

```
$ setup astrometry_net_data sdss-dr9-fink-v5b+ci_hsc
```

When we run pipelines, the outputs will go into a new data repository we call a *rerun*. By default, reruns are created in a `rerun/<rerun-name>` subdirectory of the original data repository. Reruns can be chained – a rerun from an early stage of processing may be used as the input data repository for another stage.

Exposure Processing

The main command-line driver for processing individual exposure images is `singleFrameDriver.py`, and like all of our command-line scripts, it's implemented in a *Task* class of the same name: `lsst.pipe.drivers.SingleFrameDriverTask`. We can run it on a single visit with the following command:

```
$ singleFrameDriver.py DATA --rerun example1a --id visit=903334 --cores=4
```

As the `--cores=4` argument implies, this will parallelize the work over four cores (on the same node). By setting the `--batch-type` argument to “pbs” or “slurm”, `singleFrameDriver.py` can also submit to a batch queue instead of running locally (you'll have to pass some other options as well, typically, to identify yourself to the queue). This sort of parallelization functionality is shared by all of our very highest-level tasks: those that inherit from `lsst.ctrl.pool.BatchParallel`. These usually live in the `pipe_drivers` package and have names that end with “Driver”.

The other arguments here are common to all command-line tasks:

- The first argument (`DATA` above) is the path to the root data repository (the one that contains raw data).
- We use the `--rerun` argument to give the rerun a name. The example above will put the outputs in `DATA/rerun/example1a`.
- We use the `--id` argument to pass *data IDs* that indicate which data to process. There's a fairly complex syntax for specifying multiple data IDs in one `--id` argument that we'll touch on later, but you can always also just use the `--id` option multiple times. Different instruments also have different data IDs for specifying raw data. HSC and CFHT use `{visit=}`, for instance, while LSST uses `{visit,raft,sensor}`.

`singleFrameDriver.py` always processes full `visit=`, which is why we've left off the CCD part of the data ID (actually, it processes as many of the CCDs in a visit that it can find in the registry – you'll note that `ci_hsc` doesn't include them all).

Most of the work in `singleFrameDriver.py` is delegated to `lsst.pipe.tasks.ProcessCcdTask`, which has its own command-line script, `processCcd.py`. You can call this directly if you just want to process a CCD or two:

```
$ processCcd.py DATA --rerun example1b --id visit=903334 ccd=16^100 -j2
```

You'll note that we've included the CCD part of the data ID here, and we've passed two CCD IDs, separated by a `^`. We've also replaced the `--cores=4` argument with `-j2`.

`lsst.pipe.tasks.ProcessCcdTask` doesn't inherit from `lsst.ctrl.pool.BatchParallelTask`, so it doesn't have the more sophisticated parallelization and batch submission features. But you can still parallelize over multiple local cores by specifying the number with `-j`.

Exposure-level processing includes doing basic detrending (ISR), PSF determination, cosmic ray detection and interpolation, WCS and magnitude zeropoint fitting, and basic detection, deblending, and measurement. It produces two main data products:

`calexp`

The calibrated exposure image for each CCD, including its PSF, WCS, and zeropoint in addition to the image, mask, and variance pixels. This is an instance of

```
lsst.afw.image.ExposureF.
```

`src`

The catalog of single-epoch sources for each CCD. This is an instance of

```
lsst.afw.table.SourceCatalog.
```

We'll cover how to read these datasets in [Using the Butler](#). They'll also be used by later pipelines.

In order to move on to the next steps, we'll want to first process data from multiple exposures. To process all of the visits in the `ci_hsc` dataset, do:

```
$ singleFrameDriver.py DATA --rerun example1 --cores=4 \  
  --id visit=903334..903338:2 --id visit=903342..903346:2 \  
  --id visit=903986..903990:2 --id visit=904010^904014
```

We've used a few more forms of `--id` syntax here:

- `X..Y:2` means "all IDs between X and Y (inclusive), incrementing by 2" (HSC visit numbers are always even).
- We've used `^` to join two visits we want to process, just as we used it with CCD IDs previously.
- We've passed `--id` multiple times, which just results in processing everything listed in all `--id` options.

Since we're only passing visit IDs here, using `--id` multiple times is the same as using `^`.

Note that this isn't true in general; `--id visit=X^Y ccd=A^B` processes both CCD A and CCD B for each of visit X and visit Y.

Joint Calibration

After processing individual exposures, we'd ideally do a joint fit of their catalogs to generate improved astrometric and photometric solutions. We call this procedure Joint Calibration. Unfortunately, this stage isn't quite up and running in the latest version of the LSST software stack. We have two packages for joint calibration:

- `meas_mosaic` was developed on a fork of the LSST software stack customized for HSC processing and has not yet been fully reintegrated into the LSST mainline. We expect this to happen very soon, but even when it is released `meas_mosaic` may only be capable of processing HSC data.
- `jointcal` is an in-development replacement for `meas_mosaic` that uses considerably more efficient algorithms. It will eventually support all (or nearly all) cameras with an `obs` package, but is not yet fully ready for production use. It already runs reliably on CFHT data and has been run successfully on data from a few other cameras, but its outputs have not yet been integrated into later stages of the pipeline, so the improved calibrations it generates are simply lost.




Coaddition

Image coaddition requires two different kinds of data IDs to be specified, because it concerns both the input images (the same exposure-level IDs that we saw in [Exposure Processing](#)) and the output coadds, which are organized into *tracts* and *patches* on the sky (as well as their filter). A tract is a large region containing many patches, and all patches within a tract share the same WCS with only integer offsets between them.


A particular tract and patch definition is called a `skymap`, and these are implemented by subclasses of `lsst.skymap.BaseSkyMap`. Full-sky and other large-area skymaps are created by the `makeSkyMap.py` script, which can be passed a configuration file to set up the desired skymap (most `obs` packages define a default skymap). Here, we'll instead use what we call a *discrete* skymap (`lsst.skymap.DiscreteSkyMap`), which is simply a single tract (with ID 0) at a particular pointing. We can use the `makeDiscreteSkyMap.py` script to create one that automatically encloses a collection of exposure-level images, by inspecting the bounding boxes and WCSs of the `calexp` data products produced by exposure processing:

```
$ makeDiscreteSkyMap.py DATA --rerun example1:example2 \
  --id visit=903334..903338:2 --id visit=903342..903346:2 \
  --id visit=903986..903990:2 --id visit=904010^904014 \
  --config skyMap.projection="TAN"
```

We've used the exact same data IDs here that we used when running `singleFrameDriver.py`, to ensure all of the images we've processed are included in the tract. There are two other new features of command-line processing demonstrated here:




- We've passed "example1:example2" as to the `--rerun` option. This *chains* the reruns, using "example1" as the input and "example2" as the new output. It's often a good idea to create a new rerun when you move on to a new stage of processing, so you can easily reprocess just that stage or remove just that stage's outputs. The last rerun in a chain has access to all of the data products in other data repositories in its chain (this is  of the big conveniences provided by the butler), so there's essentially no downside to creating a new rerun.
- We've used the `--config` (`-c`) option to customize the behavior of the task. All tasks have a tree of configuration options (usually an enormous one), and you can dump the full list to stdout by passing the `--show=config` command-line option to any script. Like `--help`, `--show=config` doesn't actually run the task, but you still need to provide the first (root data repository) argument, because that determines the *obs* package used and hence the values of some configuration options. You can also provide a file of configuration overrides in the same format by using the `--configfile` (`-C`) option. Config files are actually just Python files  that are exec'd in a special context .

`makeDiscreteSkyMap.py` doesn't have to do much work, so there's no point in parallelizing it. It will report the position of the skymap it creates and the number of patches in its logs; for the *ci_hsc* dataset, that should be `3 x 3`.


Now that we've defined the skymap (formally the `deepCoadd_skyMap` data product), we can use the `coaddDriver.py` script (`lsst.pipe.drivers.CoaddDriverTask`) to build a coadd. Coadds are built patch-by-patch, and we can build a single patch (the middle ) for both of the filters in the *ci_hsc* dataset with the following commands:

```
$ coaddDriver.py DATA --rerun example2 \
  --selectId visit=903334..903338:2 --selectId visit=903342..903346:2 \
  --id tract=0 patch=1,1 filter=HSC-R --cores=4

$ coaddDriver.py DATA --rerun example2 \
  --selectId visit=903986..903990:2 --selectId visit=904010^904014 \
  --id tract=0 patch=1,1 filter=HSC-I --cores=4
```

Unfortunately, `coaddDriver.py` isn't clever enough to realize that a coadd in a particular filter should only use visit images from that filter , so we have to manually split up the visits by filter and run the command twice . We've used the `--selectId` options to specify the input data IDs, and `--id` to specify the output data IDs. It's okay to provide more input data IDs than actually overlap the output patch; the task will automatically filter out non-overlapping CCDs. Like `singleFrameDriver.py`, `coaddDriver.py` is based on `lsst.ctrl.pool.BatchPileTask` , so we're using `--cores` to specify the number of (local) cores to parallelize over. We've also just used

`--rerun example2` to specify the rerun; this is now equivalent to `--rerun example1:example2` because we've already created the "example2" rerun and declared "example1" as its input (once a data repository is created in a chain, it cannot be disassociated from that chain).

We can process multiple patches at once, but there's no nice `--id` syntax for specifying multiple adjacent patches; we have to use `^`  which is a bit verbose and hard to read. Here are the command-lines for processing the other 8 patches:

```
$ coaddDriver.py DATA --rerun example2 \  
  --selectId visit=903334..903338:2 --selectId visit=903342..903346:2 \  
  --id tract=0 patch=0,0^0,1^0,2^1,0^1,2^2,0^2,1^2,2 filter=HSC-R \  
  --cores=4
```

```
$ coaddDriver.py DATA --rerun example2 \  
  --selectId visit=903986..903990:2 --selectId visit=904010^904014 \  
  --id tract=0 patch=0,0^0,1^0,2^1,0^1,2^2,0^2,1^2,2 filter=HSC-I \  
  --cores=4
```

`coaddDriver.py` delegates most of its work to `lsst.pipe.tasks.MakeCoaddTempExpTask`, `lsst.pipe.tasks.SafeClipAssembleCoadd`, and `lsst.pipe.tasks.DetectCoaddSourcesTask`, which each have their own scripts (`makeCoaddTempExp.py`, `assembleCoadd.py`, and `detectCoaddSources.py`, respectively), and like `lsst.pipe.tasks.ProcessCcdTask`, only support simple `-j` parallelization. The first of these builds the `deepCoadd_tempExp` data product, which is a resampled image in the tract coordinate system for every patch/visit combination. The second combines these into the coadd images themselves. The third actually starts the process of detecting sources on the coadds; while this step fits better conceptually in [Multi-Band Coadd Processing](#), it actually modifies the coadd images themselves (by subtracting the background and setting a mask bit to indicate detections). So we do detection as part of coaddition to allow us to only write one set of coadd images, and to do so only once (though both sets of images are written by default).

There are a few features of our coadds that are worth pointing briefly here:

- Our coadds are not PSF-homogenized. Instead, we construct a PSF model on the coadd by interpolating, resampling, and combining the single-exposure PSF models with the appropriate weights. Eventually LSST will produce PSF-homogenized coadds as well, and there are already some configuration options to enable this, but they're currently broken (resampling and PSF homogenization are done in the wrong order, so the homogenization doesn't quite work).
- We do not do any direct outlier rejection when building our coadds, as this can do serious damage to coadd PSFs. Instead, we find artifacts (e.g. satellite trails) by comparing the difference between a coadd built with per-pixel outlier rejection and a coadd built with no rejection whatsoever to detections done on single visits. Masking artifacts found this way does much less damage to the PSFs (and it lets us flag objects whose PSFs have been damaged), and it frequently works better than pixel-level outlier rejection. It doesn't work perfectly, however, and it's not the approach we plan to eventually use in LSST operations (we'll instead find these artifacts on difference images).
- We ultimately plan to delay all background subtraction until after coaddition, while using a procedure called *background matching* to ensure backgrounds are consistently defined over groups of overlapping images. This isn't working yet, but there are still a lot of configuration options in the coaddition tasks for it.

The data products produced by coaddition are:

`deepCoadd_tempExp`

Resampled images for every patch/visit combination. These may be deleted after coadds are built to save space. This is one of the few operations where direct filesystem operations are necessary, however – there's no way to delete files with the butler yet.

`deepCoadd_calExp`

Background-subtracted coadds with detection masks. Includes the coadded PSF model.

`deepCoadd`

Original coadds without detection masks and only any background subtraction done on the individual images. Includes the coadded PSF model. These are not used by later pipelines, and writing them can be disabled by passing the config option

`assembleCoadd.dowrite=False` to `coaddDriver.py`.

`deepCoadd_det`

A catalog of detections, done separately on each patch/band combination. As there is no deblending or measurement of these detections, this catalog is not very useful directly, but it is an important input to the next stage of processing.

Multi-Band Coadd Processing

LSST's coadd processing pipeline is designed to produce consistent cross-band catalogs, in terms of both deblending and measurement. After detecting separately in every band (which is included in [Coaddition](#)), there are four steps, each of which is associated with its own command-line task:

- We merge detections across bands in a patch using `lsst.pipe.tasks.MergeCoaddDetectionsTask` (`mergeCoaddDetections.py`). This produces a single catalog data product, `deepCoadd_mergeDet`. Like `deepCoadd_det`, this catalog isn't useful on its own.
- We deblend and measure objects independently in every band using `lsst.pipe.tasks.MeasureMergedCoaddSourcesTask` (`measureCoaddSources.py`). This produces the first generally-useful coadd catalog, `deepCoadd_meas`. Because the objects are defined consistently across all bands, the rows of all of the per-band `deepCoadd_meas` catalogs refer to the same objects, making them easy to compare.
- We compare measurements across bands, selecting a "reference" band for every object, using `lsst.pipe.tasks.MergeMeasurementsTask` (`mergeCoaddMeasurements.py`). This produces the `deepCoadd_ref` catalog (one for all bands), which just copies a row from the `deepCoadd_meas` corresponding to each object's reference band, while adding a flag to indicate which band was selected as the reference for that object. The rows of the per-band `deepCoadd_forced_src` catalogs also line up with each other and those of the `deepCoadd_meas` and `deepCoadd_ref` catalogs.
- We measure again in every band while holding the positions and shapes fixed at the values measured in each object's reference band, using `lsst.meas.base.ForcedPhotCoaddTask` (`forcedPhotCoadd.py`). This produces the `deepCoadd_forced_src` dataset, which provides the flux measurements that provide our best estimates of colors.

Because our coadds are not PSF-homogenized, the forced coadd fluxes don't produce consistent colors unless some other form of PSF correction is applied. In production settings, we use an external catalog of bright stars to set some masks when building coadds, fluxes and optional CModel fluxes (see [Enabling Extension Packages](#)) do provide this correction, while other fluxes do not (and the CModel correction is only approximate; it depends on how well the galaxy's morphology can be approximated by a simple model).

There is no need to run these tasks independently; the `multiBandDriver.py` script (`lsst.pipe.drivers.MultiBandDriverTask`) can be used to run them all in the appropriate order. This is a `lsst.ctrl.pool.BatchParallelTask`, so all of the more sophisticated parallelization

options are available. Before we we run it, however, we'll have to create a small configuration file.

In production settings, we use an external catalog of bright stars to set some masks when building coadds, and when we measure, we use those masks to set flags on the objects. Since we haven't used that external catalog here, we need to turn off the flag-setting, and that's a bit more complex than we can do on the command line. Here is the content of the file; save it as

`no-bright-object-mask.py`:

```
config.measureCoaddSources.measurement.plugins["base_PixelFlags"].masksFpCenter.remove("BRIGHT_OI")
config.measureCoaddSources.measurement.plugins["base_PixelFlags"].masksFpAnywhere.remove("BRIGHT_OI")
```

```
$ multiBandDriver.py DATA --rerun example2:example3 \
  --id tract=0 patch=1^1 filter=HSC-R^HSC-I \
  --cores=2 -C no-bright-object-mask.py
```

We've run only the middle patch here. Because there's so little data here, the outer patches have a lot of area with no valid pixels, and coadd processing will fail if there is too much missing area (unless you set some other configuration options we won't go into here). You'll also see a lot of warnings about failed measurements even on the middle patch for the same reason. Because we're only running one patch, we're also only using two cores, as that's the most the script will be able to make use of (because there are two filters).

Other Command-Line Tasks

The LSST includes a few more pipelines that aren't covered in detail here. None of these are

`lsst.ctrl.pool.BatchParallelTasks`s, so they don't support sophisticated parallelization. The most important ones are:

- Calibration product production, using the `construct[Bias,Dark,Flat,Fringe].py` scripts. These have only been rigorously tested on HSC data, but they should work on most other cameras as well.
- Forced photometry on exposure images with the coadd reference catalog, using `forcedPhotCcd.py` (`lsst.meas.base.ForcedPhotCcdTask`). This works, but we don't have a way to deblend sources in this mode of processing yet, so the results are suspect for blended objects.
- Difference imaging and transient source detection and characterization, using `imageDifference.py` (`lsst.pipe.tasks.ImageDifferenceTask`). This has been run quite successfully on several datasets by experts, but may require some configuration-tuning to get high-quality results in general.

Enabling Extension Packages

Some of the most useful measurement algorithms are included in the LSST stack as optional extension packages, and may not be enabled by default for a particular *obs* package (and even if they are, a [EUPS](#) product may need to be explicitly setup).

These include:



- Kron photometry, in the [meas_extensions_photometryKron](#) package.
- Shear estimation using the HSM algorithms, in the [meas_extensions_shapeHSM](#) package.
- CModel galaxy photometry, in the [meas_modelfit](#) package.

With the exception of CModel, simply setting up these [EUPS](#) products will enable them when processing HSC data (and CModel will be enabled in this way very soon). For other *obs* packages, we recommend inspecting the `config` directory of `obs_subaru` to find configuration files that can be used to enable these extensions (such a file exists for CModel as well, even though it isn't used by default).

Note that photometry extension algorithms should be enabled in both exposure processing and coadd processing, even if coadd fluxes are the only ones of interest; we need to run the algorithms on individual exposures to calculate their aperture corrections, which are then coadded along with the PSFs to calculate coadd-level aperture corrections.

Using the Butler

Data products produced by the pipelines described above are best accessed using the butler. Creating a butler in Python is easy; just pass the rerun directory to the

`lsst.daf.persistence.Butler` constructor:

```
from lsst.daf.persistence import Butler
butler = Butler("DATA/rerun/example3")
```

We can then use the `get` method to extract any of the data products we've produced; for example:

```
calexp = butler.get("calexp", visit=903334, ccd=16, immediate=True)
src = butler.get("src", visit=903334, ccd=16, immediate=True)
skyMap = butler.get("deepCoadd_skyMap", immediate=True)
coadd = butler.get("deepCoadd_calexp", tract=0, patch="1,1", filter="HSC-I", immediate=True)
meas = butler.get("deepCoadd_meas", tract=0, patch="1,1", filter="HSC-I", immediate=True)
```

Even though some of these are in the “example1” or “example2” rerun, we can access them all through a single butler initialized to the “example3” root.

We've passed `immediate=True` to all of these to tell the butler to read and return objects immediately; if we don't, it'll return a lazy-I/O proxy that mostly behaves like the object it points at, but can occasionally be a little confusing (especially in terms of introspection).

We can also use the butler to get the filename of a data product by appending “_filename” to the data product name, in case we actually do need to manipulate the filesystem directly:

```
filename = butler.get("deepCoadd_tempExp_filename", visit=903334, tract=0, patch="1,1")[0]
```


Note that getting a `*_filename` data product actually returns a single-element list (in the future, some data products may be split across multiple files, though none currently are).

Frequently Encountered Problems

Configuration and Software Version Changes

The first time a command-line task is run in a chain of data repositories, the versions of all of the software it uses and the full configuration tree are saved to the output repositories. The next time that task is run, the versions and configuration are compared against the saved versions, and the task will fail if they're not the same. This is usually desirable in production environments, where it's important that all data units be processed the same way. It would be desirable to make the comparison only happen within one rerun, not a full rerun chain – but this is not yet implemented.

In testing work, this behavior is frequently inconvenient, and the pipeline provides options to override it: `--clobber-config` and `--clobber-versions` will simply overwrite the existing configuration and version information (respectively), and `--no-versions` will prevent version information from being written or tested entirely.

These tests can also be dangerous in parallel execution, as they can be subject to race conditions (because one process can be testing for the existing of the file while another is writing it). The built-in parallelization provided by the various `lsst.ctrl.pool.BatchParallelTask` options and `-j` are safe in this respect; these do the writing and comparisons in a single process before starting the parallel processing. External wrappers that run the same task in multiple processes may not be safe, especially if the `--clobber-*` operations are being used; the default behavior is protected from race conditions by using a locking approach based on operations that are atomic on most filesystems, but the `--clobber-*` options are not. 

Clobbering and Skipping Outputs

Some command-line tasks (especially the `*Driver.py` tasks) test whether a data product exists in the current rerun chain, and skip any processing that would be replace it. This is exactly the behavior desired when a large job dies unexpected and you want to resume it. But it can be very confusing when you actually want to re-do the processing (especially the fact that processing is skipped if the output data product appears anywhere in the rerun *chain*, not just the last rerun in the chain – this is another behavior we plan to change in the future).

Tasks with this behavior have configuration parameters to disable it, usually with names with words like “overwrite”, “clobber”, or “skip”. Because these are configuration parameters (not normal command-line options), changing them and then restarting processing in the same rerun will trigger an error of the type described in the [previous section](#).