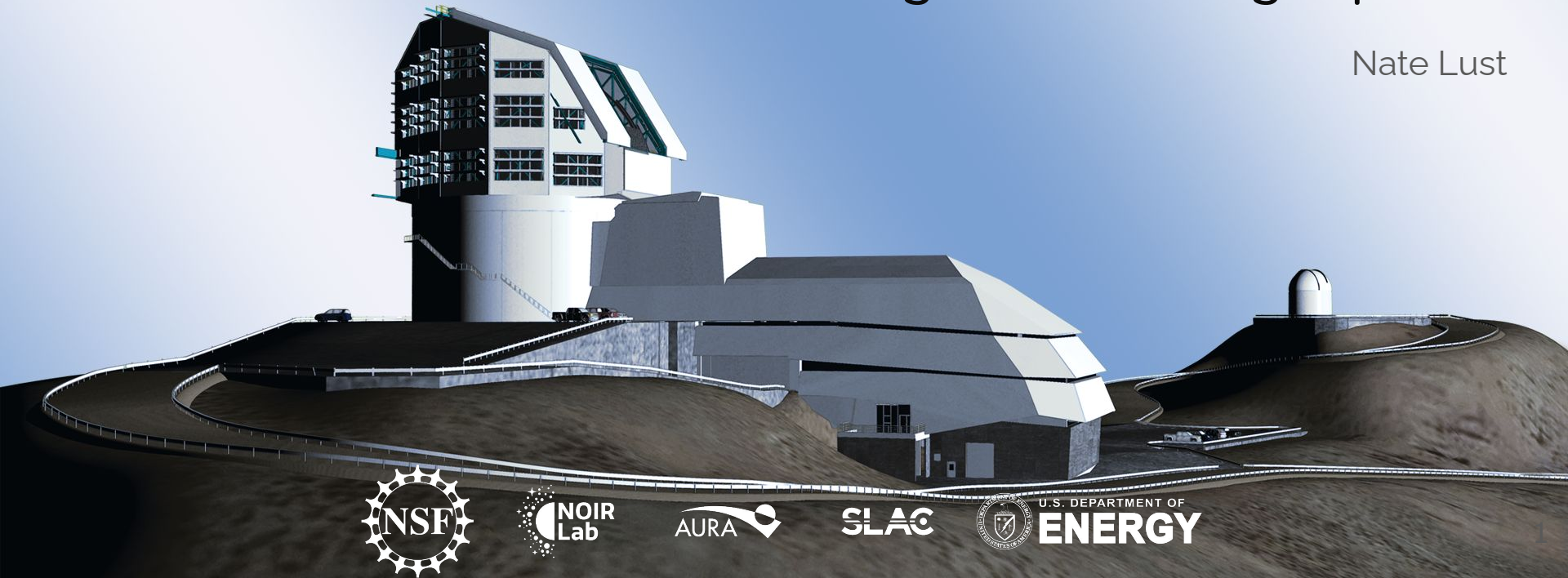
Rubin Observatory

Part 2: Science Pipelines, Writing and Running a Processing Pipeline

Nate Lust



What is a Pipeline?

Rubin Observatory

- Individual processing steps are declared in units of code called PipelineTasks
- Multiple PipelineTasks are grouped together into a processing workflow called a Pipeline
- Pipelines are documents declared in yaml format and have a python API for programmatic creation and manipulation
- Pipelines documents need not be created in an ordered fashion, execution ordering is determined by the dependency relations of the PipelineTasks that are to be run
- Pipelines create a very high level interface for controlling data processing and enable quick turnarounds with little programming or interacting with the software stack

What's in a Pipeline

Rubin Observatory

- A set of PipelineTasks to run with a unique label associated with each task
- Per label configuration (value or file based specification is supported)
- Instrument specification (optional) from which any specific configs are loaded
- Pipelines can inherit and extend other Pipelines
- Supports cross-PipelineTask configuration validation

An example Pipeline

```
description: cp_pipe DARK calibration construction
tasks:
  isr:
    class: lsst.ip.isr.isrTask.IsrTask
    config:
      connections.ccdExposure: 'raw'
      connections.outputExposure: 'cpDarkIsr'
      doBias: True
      doVariance: True
      doLinearize: True
      doCrosstalk: True
      doDefect: True
      doNanMasking: True
      doInterpolate: True
      doBrighterFatter: False
      doDark: False
      doFlat: False
      doApplyGains: False
      doFringe: False
  cpDark:
    class: lsst.cp.pipe.cpDarkTask.CpDarkTask
    config:
      connections.inputExp: 'cpDarkIsr'
      connections.outputExp: 'cpDarkProc'
  cpCombine:
    class: lsst.cp.pipe.cpCombine.CalibCombineTask
    config:
      connections.inputExps: 'cpDarkProc'
      connections.outputData: 'dark'
      calibrationType: 'dark'
      exposureScaling: "DarkTime"
      python: config.mask.append("CR")
contracts:
  - isr.doDark == False
  - cpCombine.calibrationType == "dark"
  - cpCombine.exposureScaling == "DarkTime"
```

Rubin Observatory

Parts of a Pipeline: Tasks

Rubin Observatory

- PipelineTasks to run are specified under a tasks heading, each with a unique label. This allows specifying the same Task to run multiple times with different configurations.
- Tasks are specified with a full namespace qualifier

```
tasks:
  isr: lsst.ip.isr.IsrTask
  charImage: lsst.pipe.tasks.characterizeImage.CharacterizeImageTask
  calibrate: lsst.pipe.tasks.calibrate.CalibrateTask
```

Parts of a Pipeline: Configuration

Rubin Observatory

If a task needs configuration values that differ from default values, pipelines accept a “config” directive when declaring what task to run, and class is specified using a directive named “class”.

```
tasks:
  forcedPhotCcd:
    class: lsst.meas.base.forcedPhotCcd.ForcedPhotCcdTask
    config:
      doApplyExternalPhotoCalib: False
      doApplyExternalSkyWcs: False
      doApplySkyCorr: False
```

config supports key value, external config file, and string with valid python syntax

Parts of a Pipeline: Labeled subsets

Rubin Observatory

Pipelines can label subsets of tasks, with an optional description. This enables a standard subset of tasks to be easily executed. More on this later.

```
tasks:
  isr: lsst.ip.isr.IsrTask
  charImage: lsst.pipe.tasks.characterizeImage.CharacterizeImageTask
  calibrate: lsst.pipe.tasks.calibrate.CalibrateTask
  makeWarp: lsst.pipe.tasks.makeCoaddTempExp.MakeWarp
  assembleCoadd: lsst.pipe.tasks.assembleCoadd.CompareWarpAssembleCoaddTask
subsets:
  processCcd:
    subset:
      - isr
      - charImage
      - calibrate
    description: Used to process the data of a single ccd
```

Parts of a Pipeline: Contracts

Rubin Observatory

Pipelines may specify a series of statements, named contracts, that must evaluate to true for a pipeline to be considered sound. These contracts can be used to validate that configuration between tasks is self consistent, or that a configuration value has been set.

- Contracts are specified under a top level directive named ‘contracts’
- Support either a single contract string, or a mapping of contract: str, msg: str where msg is a string to display to a user if a contract is violated

```
contracts:
  - modA.biz == modB.baz
  - contract: modA.biz > 0
    msg: "the biz configuration associated with label modA must be positive"
```

Parts of a Pipeline: Instrument

Rubin Observatory

- An ‘instrument’ directive may be added to a pipeline from which configuration will be loaded.
- The value of the directive must be a fully qualified camera class.
- An instrument directive will most commonly be seen in Pipelines defined in an obs package.

```
Instrument: lsst.obs.subaru.HyperSuprimeCam
```

Parts of a Pipeline: Inheritance

Rubin Observatory

To support composition, Pipelines are inheritable. When a Pipeline inherits from another pipeline it includes all the tasks and configuration declarations, named subsets, and any contracts defined.

Inherited pipelines can then either declare new tasks, or extend/replace existing tasks

Parts of a Pipeline: Inheritance (cont.)

Rubin Observatory

```
inherits:
  -  $ENV_VAR1/SamplePipeline1.yaml
  -  location: $ENV_VAR2/SamplePipeline
     importContracts: Fas
  -  location: $ENV_VAR
     include:
       - modA
       - modB
  -  location: $ENV_VAR
     exclude: modC
```

Pipelines to inherit are placed under a top level inherits directive

l as a single

paths to

ts

ariable

Importing of contracts can be turned off

Imports optionally support including only specific labels OR excluding specific labels

Example Pipeline Inheritance

```
description: CI HSC
instrument: lsst.obs.subaru.ge
inherits:
  - location: $PIPE_TASKS_DIR/
tasks:
  makeWarpTask:
    class: lsst.pipe.tasks.
    config:
```

```
description: ProcessCcd
tasks:
  isr: lsst.ip.isr.IsrTask
  charImage: lsst.pipe.tasks.characterizeImage.CharacterizeImageTask
  calibrate: lsst.pipe.tasks.calibrate.CalibrateTask
```

```
description: DRP
inherits:
  - location: $PIPE_TASKS_DIR/pipelines/ProcessCcd.yaml
  - location: $PIPE_TASKS_DIR/pipelines/Coaddition.yaml
  - location: $PIPE_TASKS_DIR/pipelines/Multiband.yaml
  - location: $PIPE_TASKS_DIR/pipelines/Forced.yaml
```

```
nel['AL'].alardSigGauss = [1.0, 2.0, 4.5]
  mergeDetections:
    class: lsst.pipe.tasks.mergeDetections.MergeDetectionsTask
    config:
      priorityList: ["i", "r
  mergeMeasurements:
    class: lsst.pipe.tasks.m
    config:
      priorityList: ["i", "r"]
```

```
description: Coaddition
tasks:
  makeWarpTask: lsst.pipe.tasks.makeCoaddTempExp.MakeWarpTask
  assembleCoadd: lsst.pipe.tasks.assembleCoadd.CompareWarpAssembleCoaddTask
```

Pipeline conventions

Rubin Observatory

- The name of Pipeline should follow class naming conventions (camel case with first letter capital).
- Preface a Pipeline name with an underscore if it is not intended to be inherited and or run directly (it is part of a larger pipeline).
- Use inheritance to avoid really long documents, using ‘private’ Pipelines named as specified above.
- Pipelines should contain a useful description of what the Pipeline is intended to do.
- Pipelines should be placed in a directory called ‘pipelines’ at the top level of a package.
- Instrument should packages provide Pipelines that are specifically configured for that instrument (if applicable).

Running a Pipeline

Rubin Observatory

Pipelines are run using some kind of activator. The default activator in the software stack is named pipetask

```
pipetask run -j <number_of_cores> -b <repo> --input
<input_collection(s)> --output <output_collection> -p
<package>/pipeline/Pipeline.yaml
```

- The input argument takes a name, or a list of names, of collections to search for inputs the Pipeline needs
- Output takes a name that will be used when creating a collection that will store all the outputs produced by the pipeline

Running a subset of a Pipeline

Rubin Observatory

Pipelines also support running a subset of an already defined pipeline, useful for testing/re-creating outputs. This can be a list of labels, or a start..stop range (only valid for linear pipelines)

List of labels:

```
pipetask run -j <number_of_cores> -b <repo> --input <input_collection(s)>
--output <output_collection> -p
<package>/pipelines/Pipeline.yaml:label1,label2,label3
```

Label range;

```
pipetask run -j <number_of_cores> -b <repo> --input <input_collection(s)>
--output <output_collection> -p <package>/pipelines/Pipeline.yaml:label1..label3
```

Running a subset of a Pipeline

Rubin Observatory

As mentioned Pipelines can contain one or more labeled subsets. Supply one or more names in the same way as specifying a task label to run the defined tasks. These may be mixed with task labels in a list style specification.

```
pipetask run -j <number_of_cores> -b <repo> --input <input_collection> --output
<output_collection> -p <repo>/pipelines/Pipeline.yaml:processCcd,makeWarp
```

Configuration at runtime

Rubin Observatory

Pipelines support overriding configuration at runtime with either individual values and/or a valid pex_config formated file

```
pipetask run -j <number_of_cores> -b <repo> --input <input_collection(s)> --output
<out_collection> -p <package>/pipelines/Pipeline.yaml -c modA:biz=6 -C
modB:<path>/config
```

Running an individual PipelineTask

If there are a small number of tasks to run, and that pipeline is only intended to be run one or two times, it is sometimes easier to directly specify the PipelineTasks to run. This can be done using the -t option and complete namespace of a PipelineTask along with a :<label>

```
pipetask run -j <number_of_cores> -b <repo> --input <input_collection(s)>
--output <out_collection> -t lsst.demo.TaskA:modA -c modA:biz=6
```

This method is intended for debugging or testing purposes. It is highly recommended to create a Pipeline document, or run part of all of an existing Pipeline. Running a single task will in general have a different configuration than the same task defined within a pipeline.

Examining a Pipeline to be executed

Rubin Observatory

Due to inheritance and dynamic configuration, it is sometimes hard to get a complete picture of the Pipeline that will be executed. In this case use the --show pipeline option of pipetask. The output will be the fully inherited pipeline with all command line configs applied.

pipetask build -p <package>/pipelines/Pipeline.yaml --show pipeline

Note the sub command above is build and not run, as we only want to view the pipeline.

Specifying Data

Rubin Observatory

The data that a Pipeline will process is determined by the tasks that will be run, what input collections are specified, and what constraints are given in the pipetask command.

PipelineTasks declare what datasets they require as inputs and what datasets they will produce. This information is used to order the pipeline execution. The dataset types of the beginning tasks that are present in the input collections are then loaded and used to begin the execution chain.

Further constraints on what dataset types will be loaded can be given with the -d option to pipetask.

```
pipetask run -j <number_of_cores> -b <repo> --input <input_collection(s)>
--output <output_collection> -p <package>/pipelines/Pipeline.yaml -d "band
= 'i'"
```