A Decade of LSST Middleware

Jim Bosch, LSST Data Release Science Lead

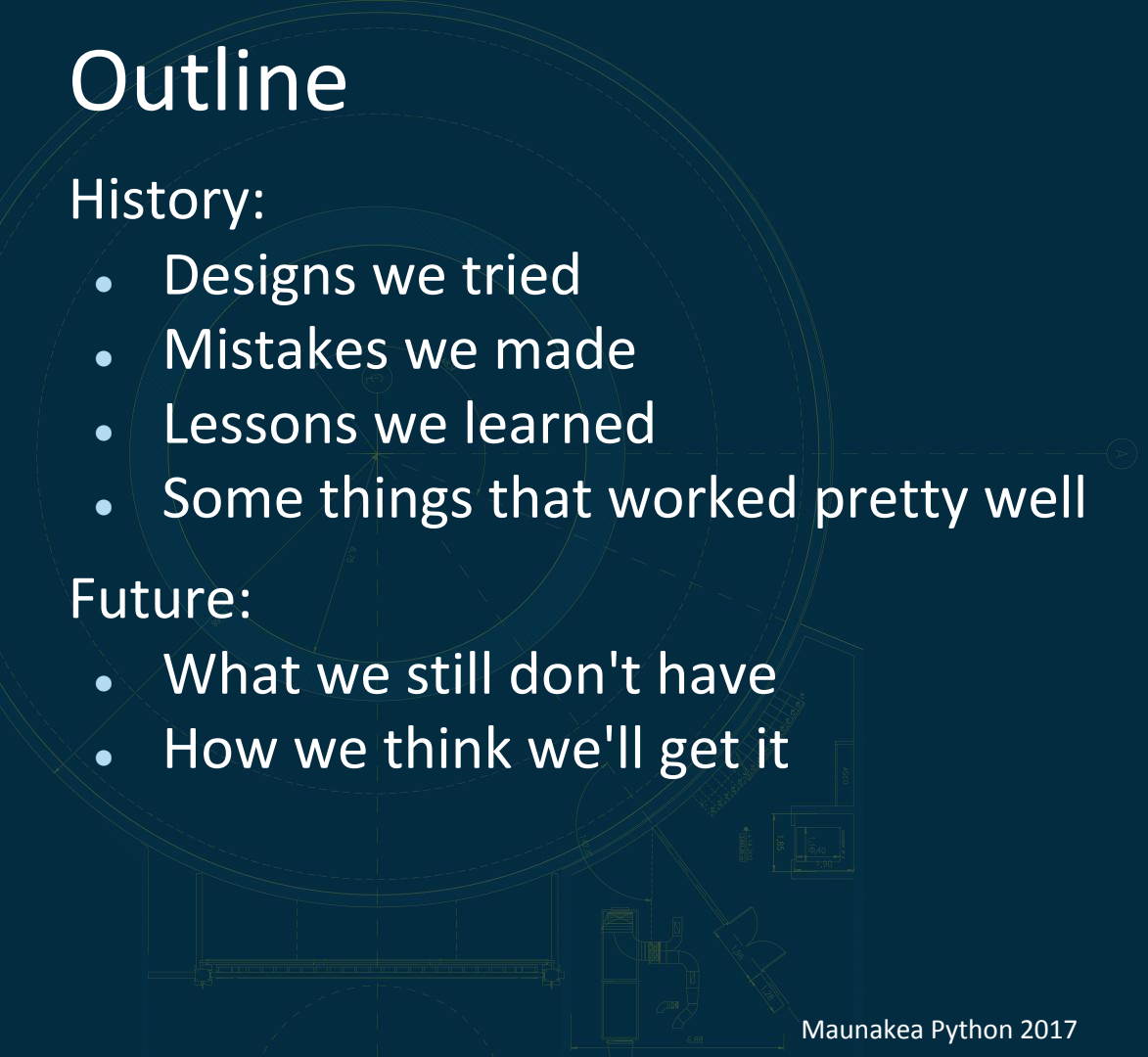
Outline

History:

- Designs we tried
- Mistakes we made
- Lessons we learned
- Some things that worked pretty well

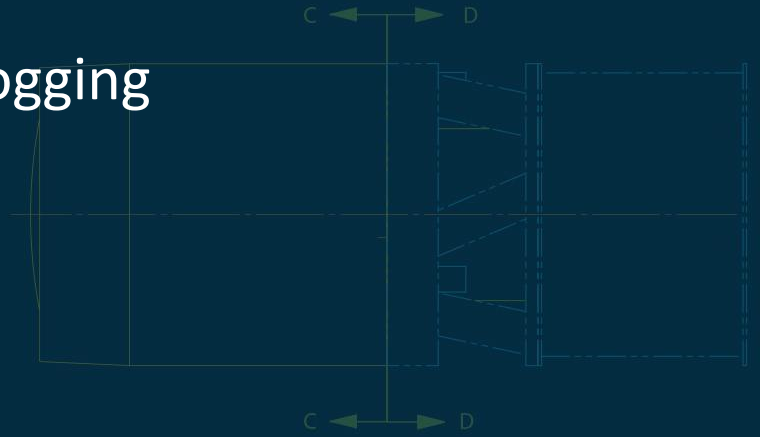
Future:

- What we still don't have
- How we think we'll get it



Early R&D: 2008-2011

pex.harness, pex.policy, and pex.logging



pex.harness

Pipeline code is split up into *Stages*:

- A Stage's process method takes a single dictionary, called a *Clipboard*, that provides all inputs and is passed all outputs.
- The keys used for Clipboard lookup are provided as part of the Stage's configuration file.

Stages are combined into pipelines by configuration files.

Stages represent small steps (e.g. source detection, PSF estimation).

What pex.harness got right

- Stages can be reordered, swapped in and out, etc. with declarative configuration.
- I/O totally abstracted away from algorithmic code.
- Large-scale execution totally up to control code.

Focused on the things pipeline operators care about.

What pex.harness got wrong

- Lots of boilerplate around algorithmic steps.
- Declarative step was too low:
  - a set of ordered Stages is itself an algorithm
  - combining them at the configuration level obscures the logic
  - the reordering/swapping flexibility was mostly a lie
- Abstractions for control code are only as good as their implementations. And our implementations were poor.
- Configuration and code defined far away from each other.

The end of pex.harness

Frustrated with trying to build a usable pipeline for Hyper Suprime-Cam using pex.harness, the HSC team (a significant fraction of the LSST team at this time) just combined the low-level algorithmic code by writing higher-level Python code that called it directly. The resulting "mini-pipeline" was called *Pipette*.

The main LSST stack never used Pipette directly (I think), but the next generation of LSST middleware would use it as a template.

pex.logging and pex.policy

More or less typical logging and text-file configuration libraries.

But we wrote them ourselves:

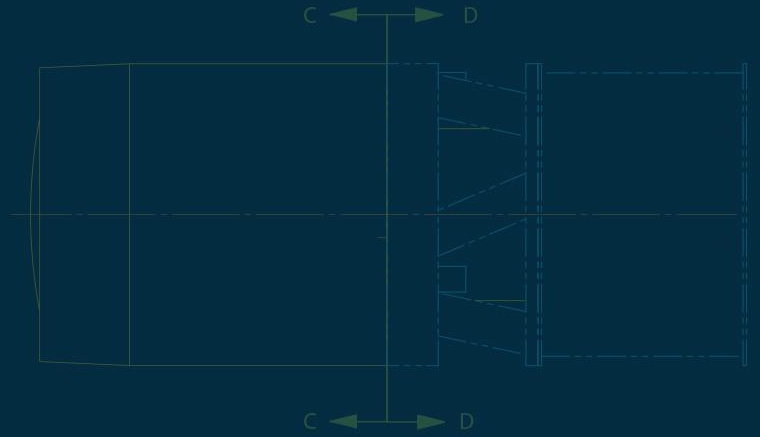
- our own file formats and parsers
- our own logger destinations and formatters

And then they became zombies:

- the original developers left the project
- we decided we'd be better off using third-party tools
- no one "owned" the work involved in replacing them

Transition to Construction: 2011-2015

pex.config, pipe.base, and Butler



pex.config

Define, document, and validate configuration options by creating Python classes.

Write configuration as Python files:

- No need to write a parser of our own.
- Unlimited flexibility in setting options programmatically.
- Use `import` in configuration files to load external plug-ins.

Defining Configuration Options

```
from lsst.pex.config import Config, Field, ConfigurableField

class DetectionConfig(Config):
    threshold = Field(
        dtype=float, default=5.0,
        doc="point-source S/N threshold"
    )
    background = ConfigurableField(
        dtype=BackgroundConfig,
        doc="options for backround subtraction during detection"
    )
```

Defining Configuration Options

A `ConfigurableField` is special `Field` subclass that holds another `Config` instance, usually for some lower-level algorithm it delegates to.

`Configs` for high-level algorithms thus naturally form a tree of configuration options.

Setting Configuration Options

We can set the configuration options for detection by `exec`'ing a file like the one below:

```
config.threshold = 4.5
config.background.binSize = 256
```

with the special variable `config` initialized to an instance of `DetectionConfig`.

How pex.config works

1. `Field` is a data descriptor (like the `property` built-in)
2. `Config` has a custom metaclass.
3. …ugly, complex metaprogramming…
4. Profit!

Great for users, woe to the maintainers.

pipe.base and Tasks

Tasks are:

- Callable - but with arbitrary signatures.
- Configurable - each `Task` class has an associated `Config`, and an instance of that `Config` is always passed to the `Task` constructor.
- Nestable - a `Task` can have a subtask: an attribute that is another `Task`.

The nested `Config` hierarchy parallels the `Task` hierarchy.

pipe.base and Tasks

Tasks are:

- Callable - but with arbitrary signatures.
- Configurable - each `Task` class has an associated `Config`, and an instance of that `Config` is always passed to the `Task` constructor.
- Nestable - a `Task` can have a subtask, an attribute that is another `Task`.

The nested `Config` hierarchy parallels the `Task` hierarchy.

Task Example

```
from lsst.pipe.base import Task

class DetectionTask(Task):
    ConfigClass = DetectionConfig

    def __init__(self, config, **kwds):
        self.makeSubtask("background")
        ...

    def run(self, exposure):
        self.background.run(exposure)
        ...
```

Retargeting

A `ConfigurableField` for a subtask can be *retargeted* to point to a `Config` instance for a different subtask:

```
config.background.retarget(SplineBackgroundTask)
```

This lets us swap out algorithm `Tasks` via configuration, but only if they have the same signatures - no *false* flexibility.

The Butler

Datasets (~files) are labeled by a dataset type name (e.g. `"src"`) and a dictionary data ID (*e.g.* `{"visit": 310, "sensor": 32}`).

Butler organizes these with a data repositories (~directories), and handles all the low-level I/O:

```
butler = Butler("/path/to/data")
catalog = butler.get("src", visit=310, sensor=32)
butler.put(catalog, "src", visit=310, sensor=32)
```

CmdLineTask

A `CmdLineTask` is a `Task` that uses a `Butler` for its inputs and outputs, and hence lives at the top of a `Config`/`Task` tree.

It also provides a specialized command-line argument parser that lets users provide configuration overrides and the data IDs to be processed.

`CmdLineTasks` can also be run with simple data-parallel multiprocessing.

Butler and CmdLineTask

A Data Repository (typically just the root one) also contains a SQLite database file that relates observational metadata (date, filter, etc.) to the data IDs.

This lets us include that metadata in our data ID expressions:

```
processCcd.py /path/to/data --id visit=350..400 ccd=0..103
```

or

```
processCcd.py /path/to/data --id filter=HSC-I field=COSMOS
```

Running CmdLineTasks

The pipe.base framework isn't great for pipeline *operators:*

- We can't change parallelization axes within a `CmdLineTask` (limiting how much you can put in a single `CmdLineTask`).
- There's no good way to run multiple dependent `CmdLineTasks` in series
- There's no good way to run even a single `CmdLineTask` in parallel beyond a single node.

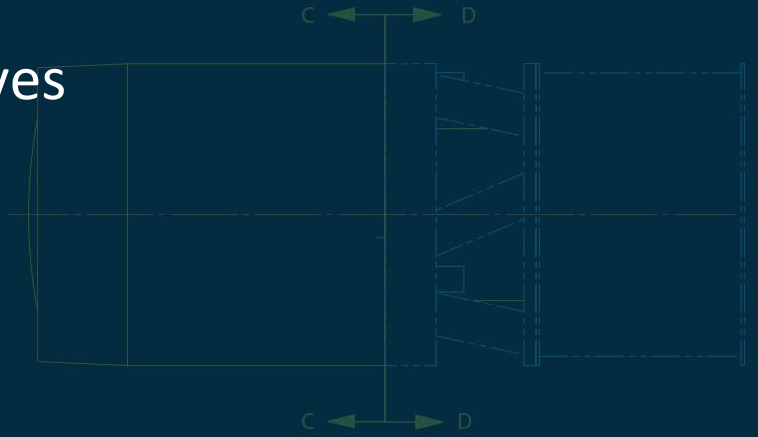
LSST and HSC developed two distinct bad ways to do these things

Evaluating Tasks, Configs, and Butler

- `Task` and `Config` have been a huge success - but maybe *too* successful.
  - We've abused `Config` trees to store non-configuration data.
  - Subtask retargeting is a hammer that can make a lot of problems look like nails.
- `Butler` and `CmdLineTask` succeeded at what they were designed for - but we quickly found ourselves wanting more from them.

Growing Pains: 2014-2017

New Butler, Log, and Nocturnal Elves



Replacing pex.logging

We needed:

- better scalability in the backend
- multithreading
- more powerful configuration system
- to not maintain of this ourselves

So we built a new logging package, delegating almost everything to *log4cxx*.

The New Butler Saga

Everyone came up with their own wish list for `Butler` features:

- relate data IDs ("what sky patches overlap this observation?")
- dynamic definition of dataset types
- subset and transfer data repositories
- composite datasets (override an image's WCS, or load the WCS without loading the image)
- record/query provenance

The New Butler Saga

Everyone imagined a New Butler that would do all of these things.

Everyone agreed it would be a good idea.

Everyone assumed it would also do anything else they thought of (even if they didn't tell anyone about those ideas).

But only the nocturnal elves were actually working on New Butler.

The New Butler Saga

While waiting for the nocturnal elves responsible for New Butler, we started hacking the features we wanted into the existing code:

- we added special-case code for the instruments whose data we processed most
- we relied heavily on undocumented private interfaces
- we put workarounds in the algorithm code
- we added lots of features to Butler itself without understanding how they fit in architecturally

The New Butler Saga

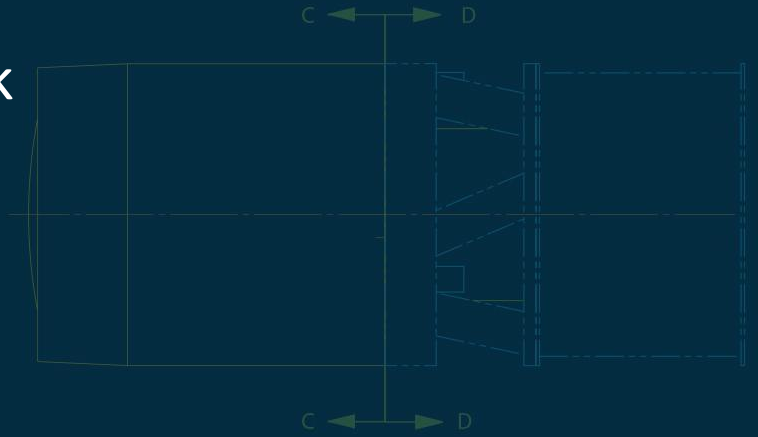
Finally, we replaced the elves with an actual developer, and tried to evolve the current Butler incrementally into New Butler.

This turned out to be impossible:

- hard to remove dependence on old implementation
- lots of functionality never captured in tests
- backwards-compatibility in on-disk data repositories was a constant problem

The Future: 2017-

Generation 3 Butler and SuperTask



Where We Are Now

- Logging and `Config`: in good shape
- `Task`: good, but for low-level code only
- `CmdLineTask`:
  - can't combine into full pipelines
  - can't run even small pipelines at scale.
- `Butler`:
  - *great* concept
  - doesn't do everything we need
  - implementation has become unmanageable

New Designs

A pair of cross-team "Working Groups" have come up with new designs to solve these problems:

- *SuperTask:* a replacement for CmdLineTask and a framework for running then.
- the *Generation 3 Butler:* a reimplementation of the Butler with a totally different architecture and lots of new functionality.

Butler Concepts

Dataset: a discrete data structure that has been stored. Usually (but not always) a single file.

Collection: a group of related Datasets. Used to label the inputs and outputs of processing runs. A Dataset can be associated with multiple Collections.

DataUnit: a unit of data that can be used to label a Dataset, such as "Visit 780" or "Sensor 10". Includes metadata.

Butler Components

Registry: ~ Database Client

Connects Dataset metadata and relationships to URIs.

Records provenance.

Defines Collections.

Datastore: ~ Filesystem Client

Reads, writes, stores, and transfers Datasets.

Creates URIs.

Finds Datasets from URIs.

Manages file formats.

Butler: Convenience Layer

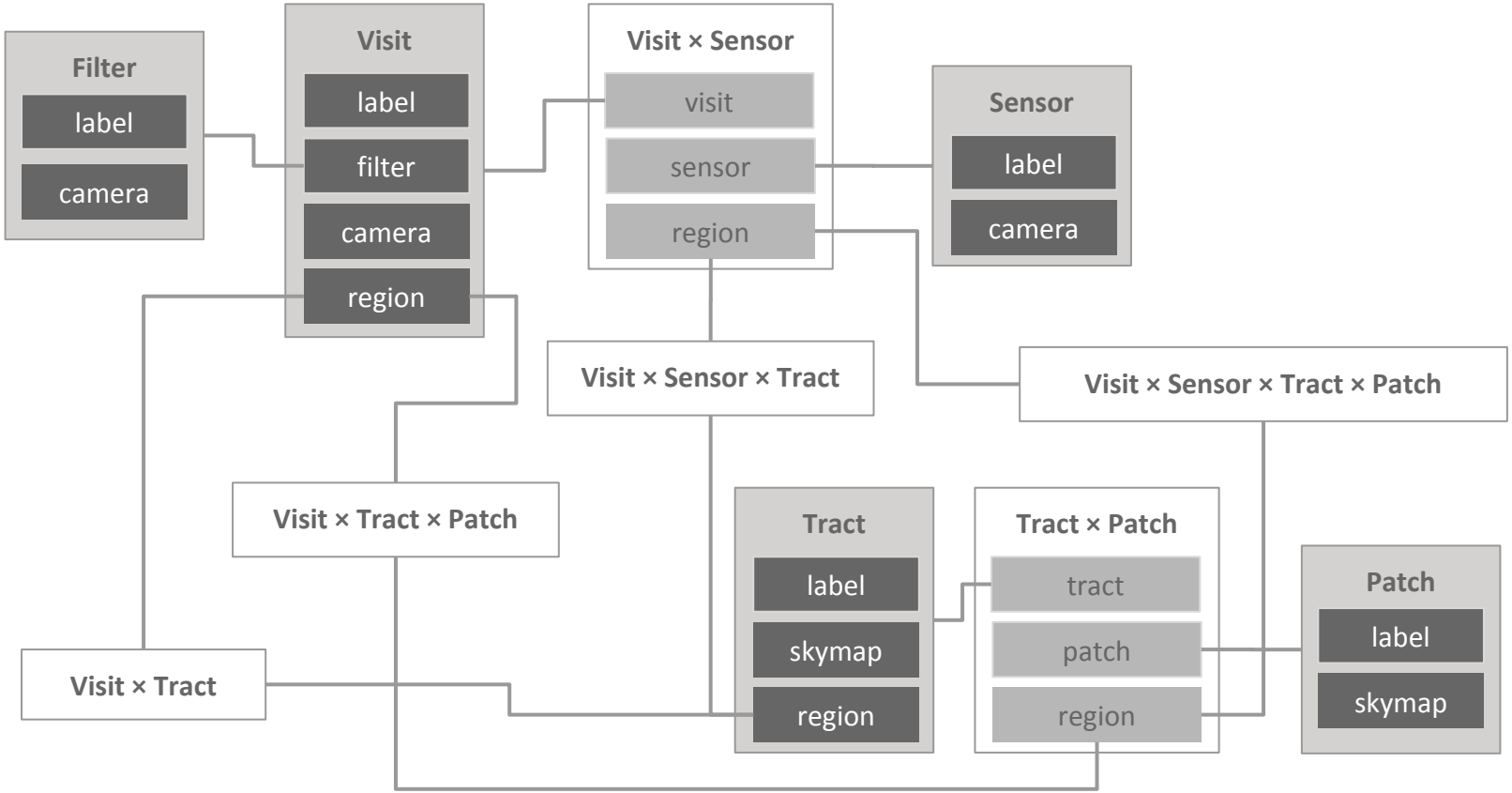
Holds and delegates to a Registry and Datastore.

Operates on a single Collection.

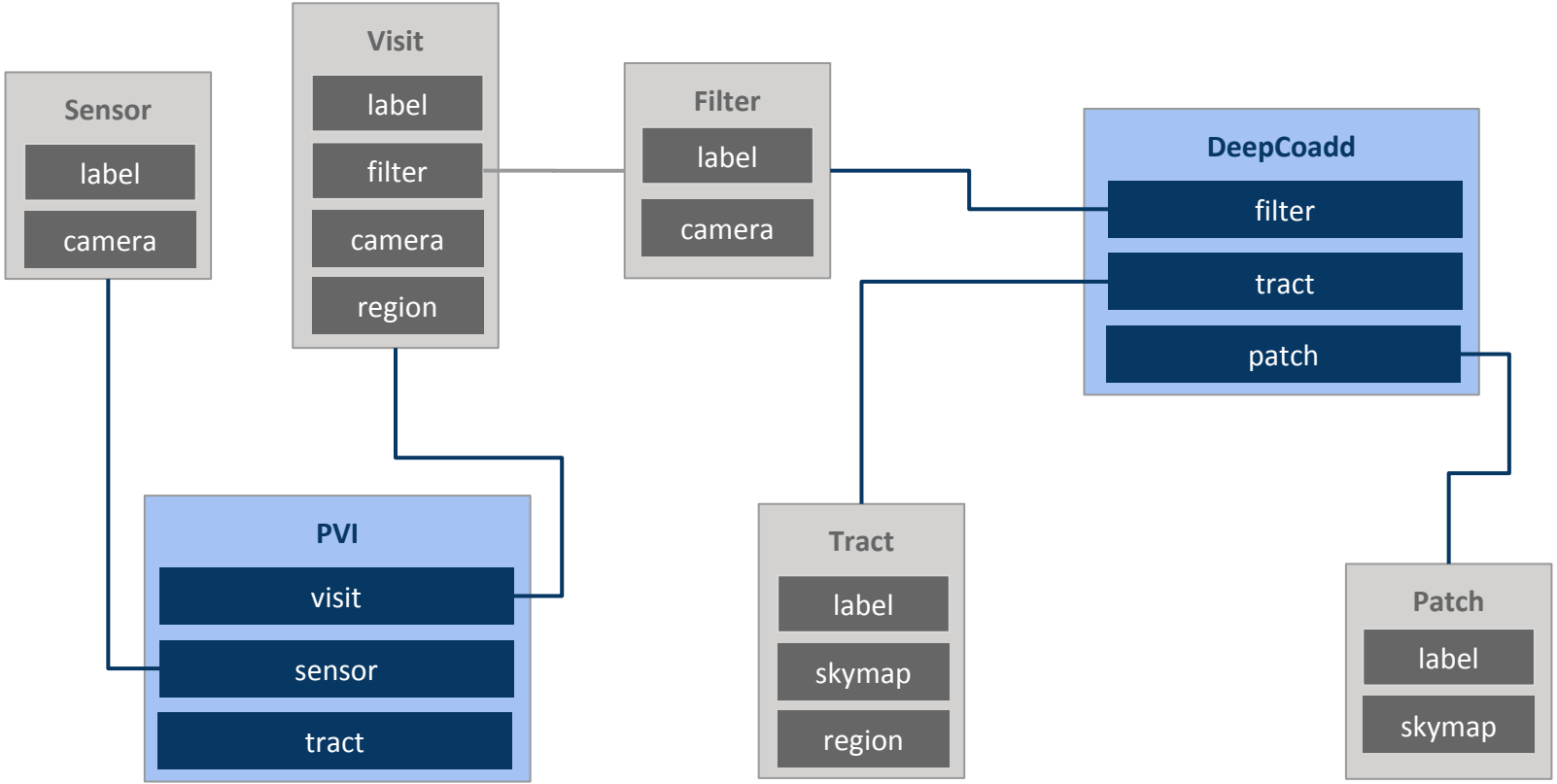
Registry

- A Python client to a SQL database that stores metadata, relationships, and URIs for Datasets.
- Exposes a SQL schema common to *all* Registries.
  - Might be implemented with views.
  - Provides a direct SQL interface for SELECT queries.
  - All inserts/updates go through (virtual) Python methods.
- Also holds provenance information.
- Can hold multiple Collections.
- *Abstract: several implementations expected.*

Registry Schema: DataUnit Joins



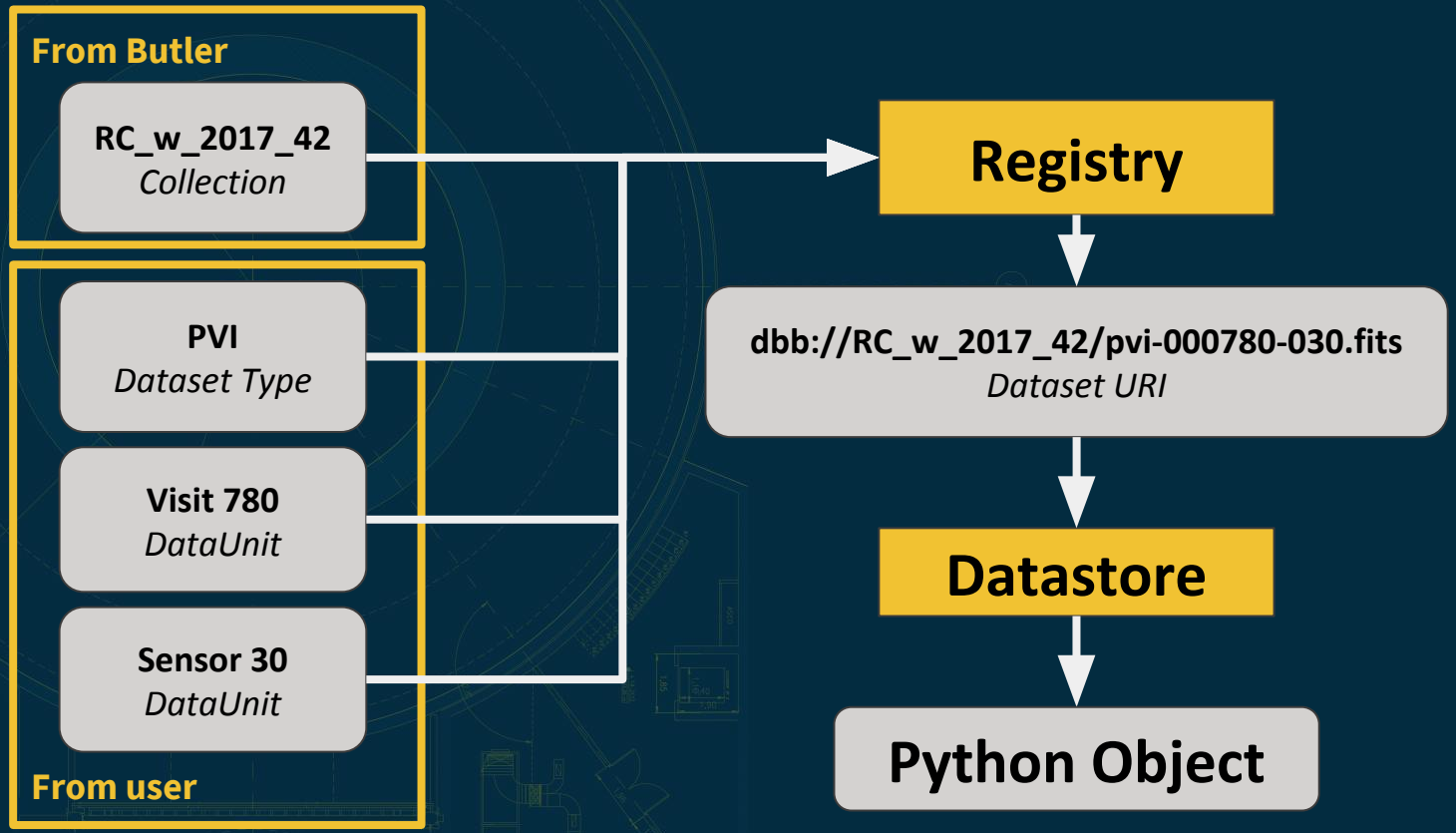
Registry Schema: Datasets



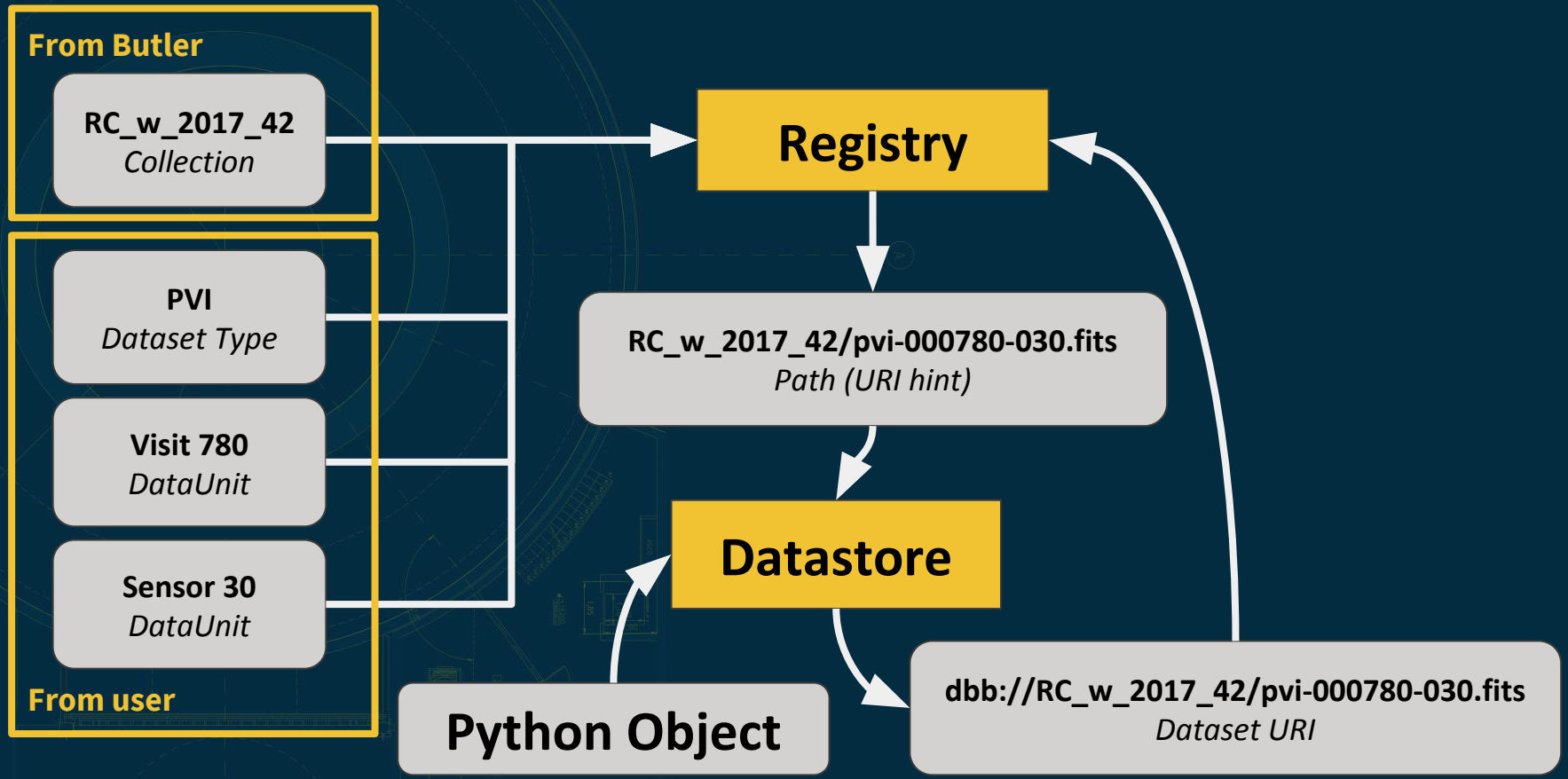
Datastore

- A Python object that can can read and write Datasets to/from URIs.
- May involve communication with a remote server (and any file transfer that involves).
- Responsible for all file format and file name (if applicable) configuration.
- Does not know about Collections or DataUnits.
- *Abstract: several implementations expected.*

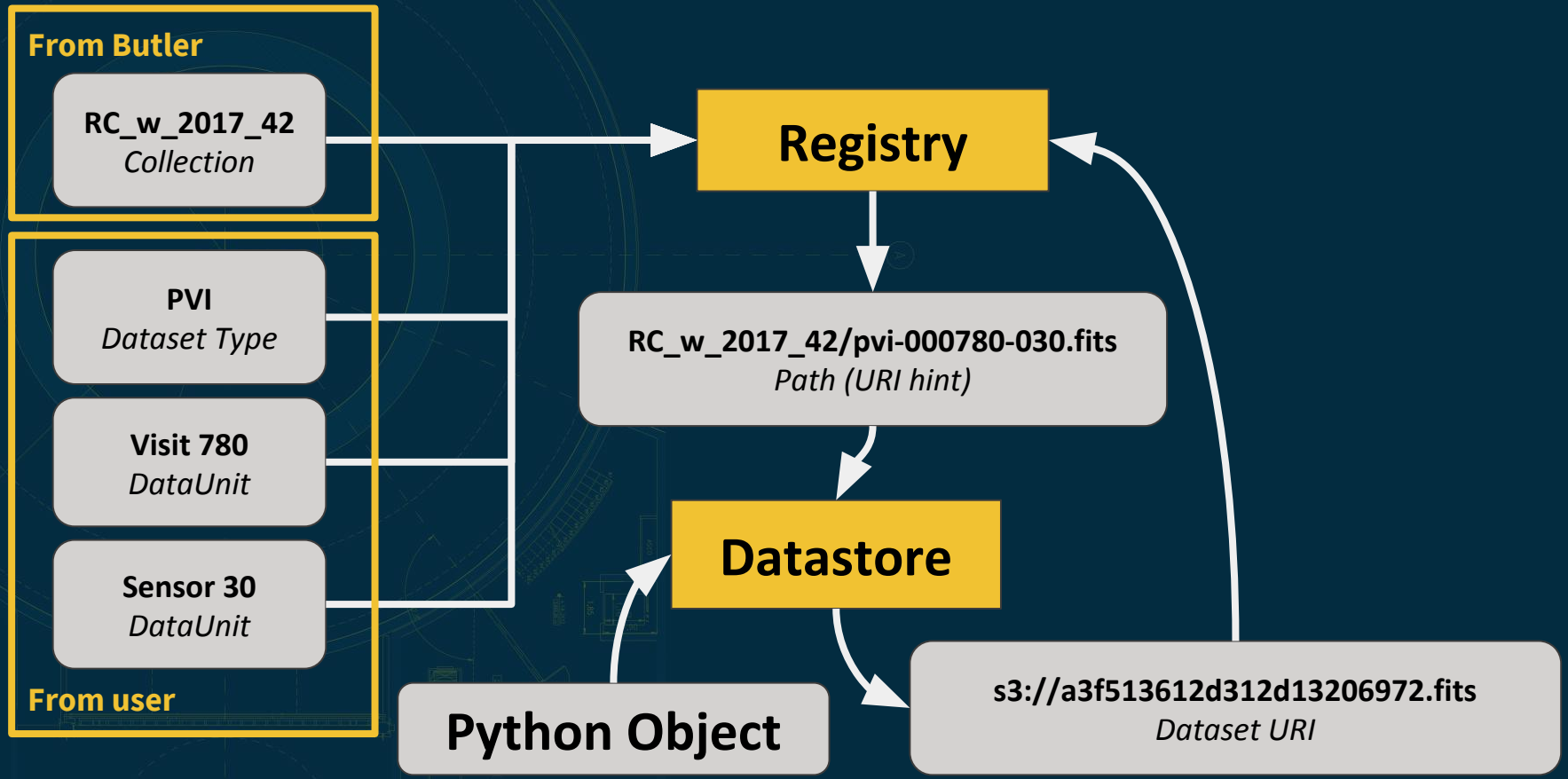
Butler.get


From Butler
RC_w_2017_42
Collection
PVI
Dataset Type
Visit 780
DataUnit
Sensor 30
DataUnit
From user
Registry
dbb://RC_w_2017_42/pvi-000780-030.fits
Dataset URI
Datastore
Python Object

Butler.put


From Butler
RC_w_2017_42
Collection
PVI
Dataset Type
Visit 780
DataUnit
Sensor 30
DataUnit
From user
Registry
RC_w_2017_42/pvi-000780-030.fits
Path (URI hint)
Datastore
Python Object
dbb://RC_w_2017_42/pvi-000780-030.fits
Dataset URI

Butler.put


From Butler
RC_w_2017_42
Collection
PVI
Dataset Type
Visit 780
DataUnit
Sensor 30
DataUnit
From user
Registry
RC_w_2017_42/pvi-000780-030.fits
Path (URI hint)
Datastore
Python Object
s3://a3f513612d312d13206972.fits
Dataset URI

SuperTask Concepts

Quantum: a discrete unit of work, containing a list of input Datasets and a list of output Datasets. The same object represents work to be done, and work already done (provenance).

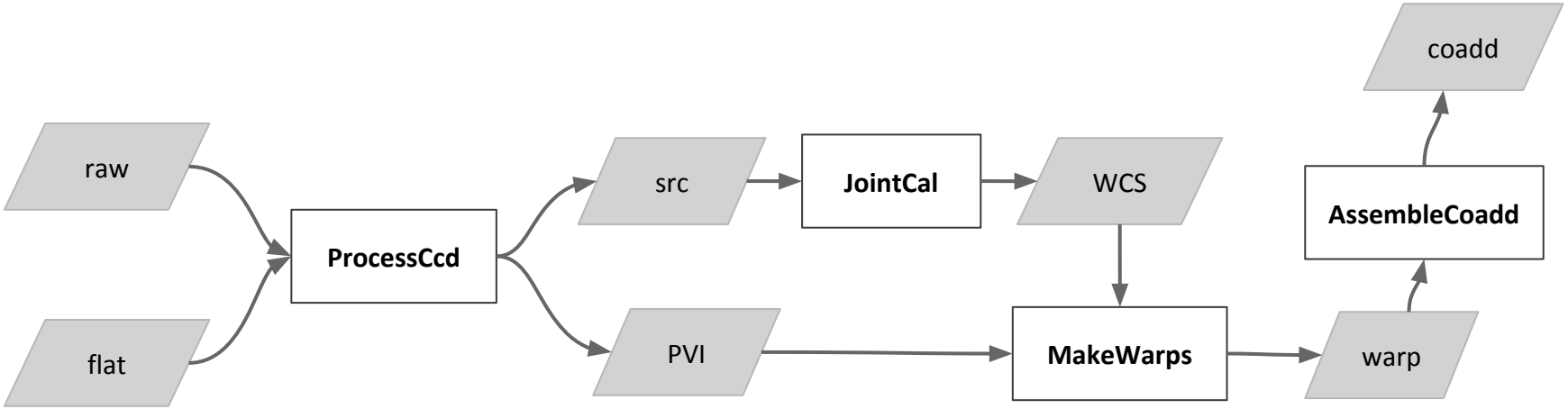
SuperTask: a Task that processes Quanta independently, using a Butler for input and output, *and knows how to define its own Quanta.*

Pipeline: a sequence of SuperTasks classes and their Configs.

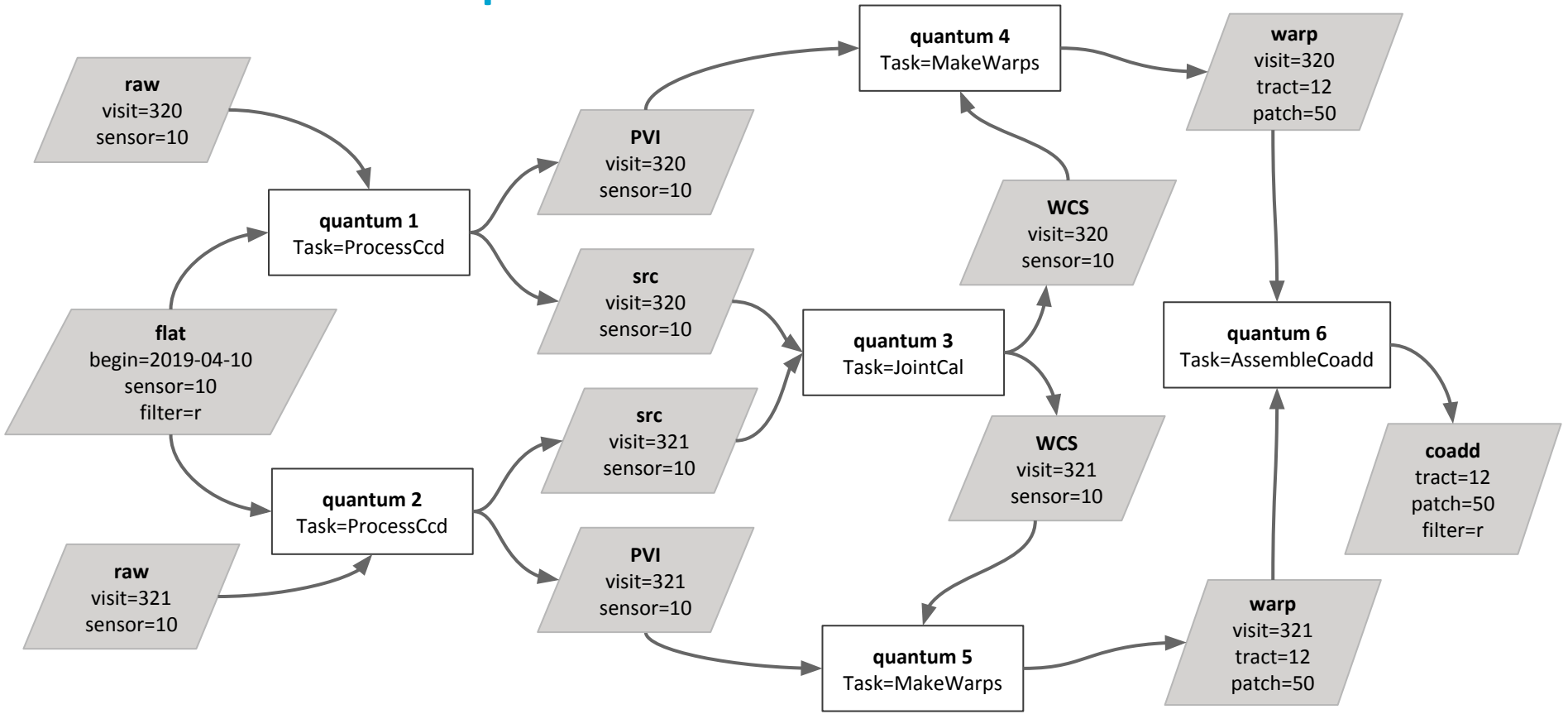
SuperTask Interface

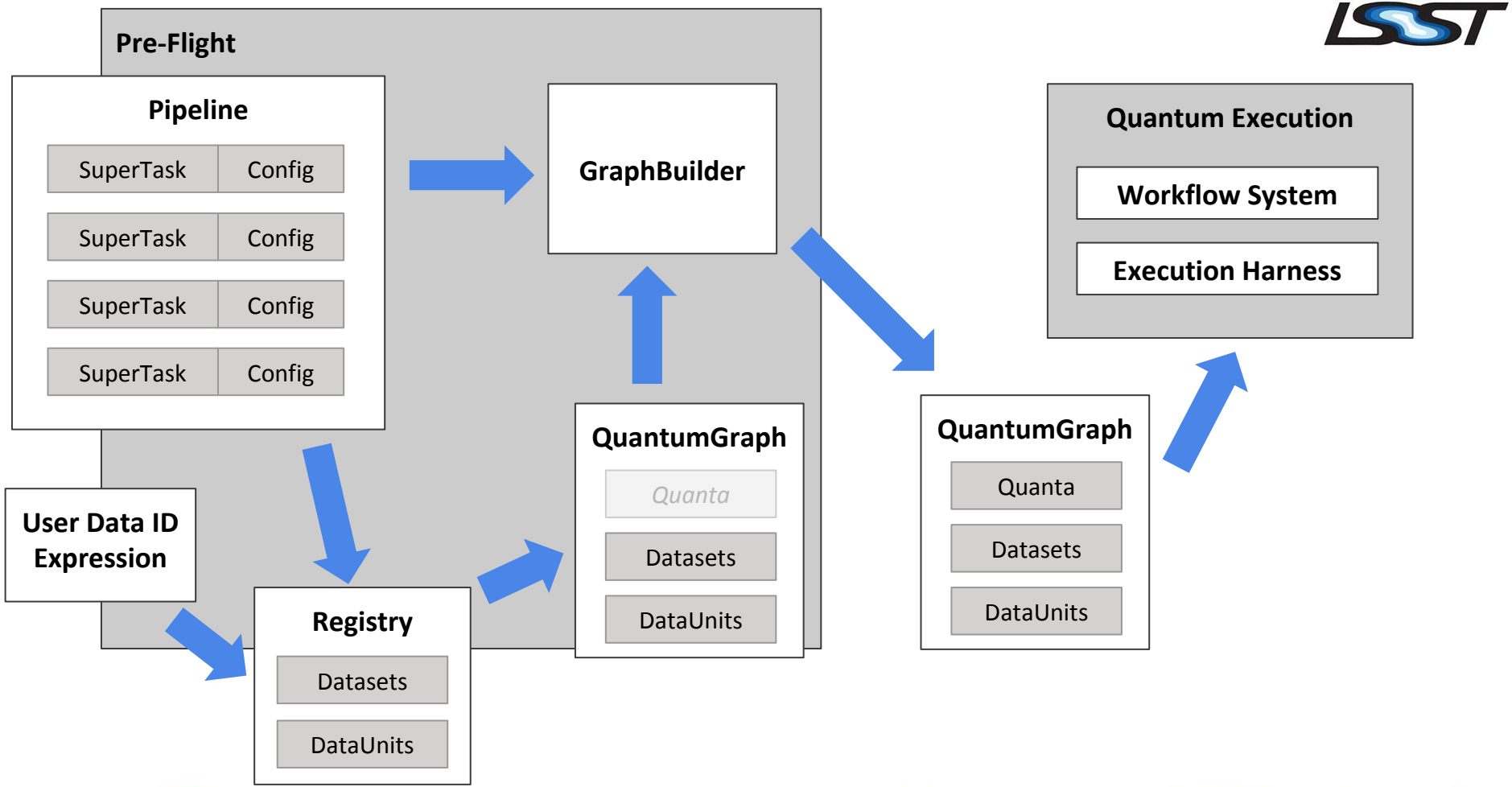
```
class SuperTask:

    def defineQuanta(self, graph):
        """Define Quanta from the Datasets in the given QuantumGraph and
        add them to it.
        """
        ...

    def runQuantum(self, quantum, butler):
        """Run the SuperTask on the inputs and outputs defined by the given
        Quantum, retrieving inputs and writing outputs using a Butler.
        """
        ...
```

Pipeline as a Graph



QuantumGraph




Pre-Flight
Pipeline
SuperTask
Config
SuperTask
Config
SuperTask
Config
SuperTask
Config
GraphBuilder
QuantumGraph
Quanta
Datasets
DataUnits
User Data ID Expression
Registry
Datasets
DataUnits
QuantumGraph
Quanta
Datasets
DataUnits
Quantum Execution
Workflow System
Execution Harness

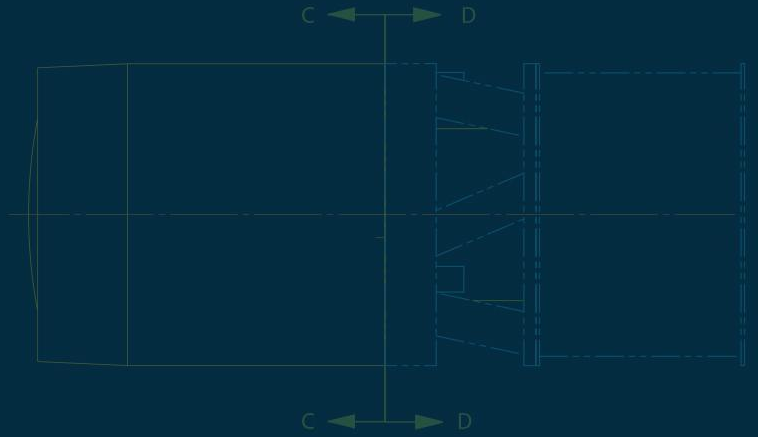
Status

High-level designs for the SuperTask Framework and the Gen. 3 Butler are mostly complete.

Minimal implementations are tentatively expected to be complete around the middle of 2018.

We'll be processing data from the auxiliary telescope in late 2018!

Lessons Learned



Closing the Loop

Stages and SuperTasks have some big similarities:

- they have generic signatures (clipboard vs. butler+quantum)
- they are combined into pipelines via declarative configuration
- they can be introspected to build an executable DAG

Did we just end up back where we started?

Closing the Loop

The boundary has moved: SuperTasks are larger than Stages.

- We can do this because the tree of *non-super* `Tasks` carries configurability, reusability, and production hooks (e.g. logging) down to "regular" code.

SuperTasks put everything in one place.

- `Configs` are just Python; they live in the same modules as the code they configure.
- SuperTasks define their Quanta as well as run them.

Using Third-Party Code

We've moved away from doing everything ourselves.

- We use Python itself to parse configuration files.
- Custom front-ends with third-party backends for logging and batch processing.
- We'll be leaning more on direct SQL queries for managing dataset metadata.

Writing custom solutions has rarely been a problem for us - but maintaining them has been.

Using Third-Party Code

We never really gave any complete third-party frameworks a shot.

- If I was starting over today, I'd think hard about Spark/Hadoop.
  - Those weren't really options in 2008.
  - It's too late for LSST to switch to something that intrusive now.
- Everyone's data is different; don't expect anything to solve all of your problems.
- It's really hard to guess the right technologies at the start of a multi-decade project.

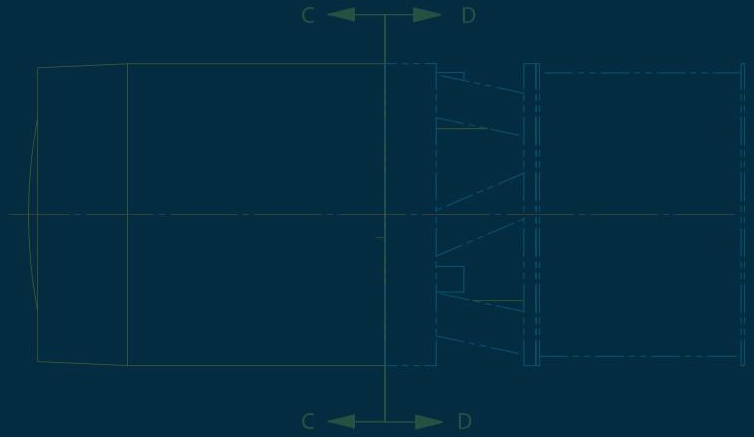
Beware Success

Good code doesn't attract users; *useful* code does.

Users make it easier to make code a little better - they submit patches, find bugs, etc.

Users make it harder to make code a *lot* better - backwards compatibility makes architectural changes difficult.

Backup Slides



Why not Python's built-in logging?

We would have needed a custom backend; this is just an interface question. We explicitly considered and debated doing this.

In the end, we decided:

- it was more important to maximize similarities between logging interfaces in C++ and Python (which are still not identical);
- using our own interface made logger configuration in hybrid C++/Python code easier.

The Butler

Data repositories can be *chained* (originally via symlink), defining a multi-repository search path:

```
/path/to/data
/path/to/data/raw/<...>

/path/to/data/rerun/dr12/_parent -> ../..
/path/to/data/rerun/dr12/src/<...>
```

A butler pointed at `/path/to/data/rerun/dr12/` will search both repositories.

More on Tasks

- Regular `Tasks` don't do I/O; only `CmdLineTasks` do.
- Configuration for a `Task` is considered frozen after it's constructed.
- `Tasks` are constructed with loggers and metadata objects that "know" their location in the hierarchy.

Nifty pex.config features

There can be many layers of configuration overrides applied to a pipeline.

So when a configuration option is assigned to, we record the file and line number where it happened in the tree of `Config` objects.

Later, when a configuration option isn't set to the value we expect, we can easily find out why.

Running CmdLineTasks

HSC wrote a MPI-based system (now `ctrl.pool`), using a higher level of `BatchPoolTasks` that call multiple `CmdLineTasks`.

- Hard-codes changes in parallelization between `CmdLineTasks`.
- Creates submission scripts for PBS/Torque and Slurm.
- Assumes (and sometimes thrashes) a big shared filesystem.
- Makes running the current pipeline 4 steps, instead of ~10.
- Still have to also split up "wide" jobs into multiple submissions: no automatic retries, load-balancing, etc.

Running CmdLineTasks

LSST wrote a HTCondor system (`ctrl.execute`) for running single `CmdLineTasks`.

- No support at all for connecting different steps.
- Has specialized configuration packages for different compute environments.
- Has better support for working on scratch space.
- Probably for running extremely "wide" jobs.