# Generation 3 Butler Design

Jim Bosch, for the Butler WG (especially Pim Schellart)

Large Synoptic Survey Telescope

## Design Philosophy

- Draw from and relate:
  - Generation 1-2 Butler interface/usage concepts
  - DESDM / Data Backbone implementation/usage concepts
- Fill in/support the missing pieces of SuperTask design.
- A single common interface, with clear customization points for implementations with different contexts and requirements:
  - production (scalability, rigorous integrity/provenance)
  - development/science (flexibility, portability, ease-of-use)

## What's Missing in Gen. 1-2 Butler

- Gen. 1-2 Butler didn't have the metadata/relationship query functionality needed to support SuperTask (or any other approach to executing more complex pipelines).
- Customization points were never successfully used (e.g. Mapper customization became Camera customization).
- A history of ad-hoc feature additions and workarounds made the divide between public and private interfaces unclear.
- Minimal repository-level provenance only.

# What's Changed from Gen. 1-2 Butler

- Multiple "data repositories" (Collections) in a single SQL database.  No more chained repositories.
- Vastly expanded schema in the SQL database, including relationships between observations and skymaps.
- SQL database is used to look up locations of stored Datasets, instead of searching the filesystem with a path template.

## What's Changed from Gen. 1-2 Butler

- Clear customization/extension points for specific environments.
- Dataset-level provenance, including input/output tracing.
- *Less* flexibility for camera-level specialization; we now conform cameras to butler, instead of customizing mappers for cameras.

## What's Missing in DESDM

DESDM is a complex production system; it's not designed for laptops, free-form rapid development, or science users.

- It *would* meet production requirements with only small modifications, and is hence is considered by Operations to be the baseline/fallback option.
- It *would not* meet Science Pipelines requirements as-is for pipeline development work.
- If a different system is developed to meet development needs, we all have concerns about transitioning science pipelines to production.

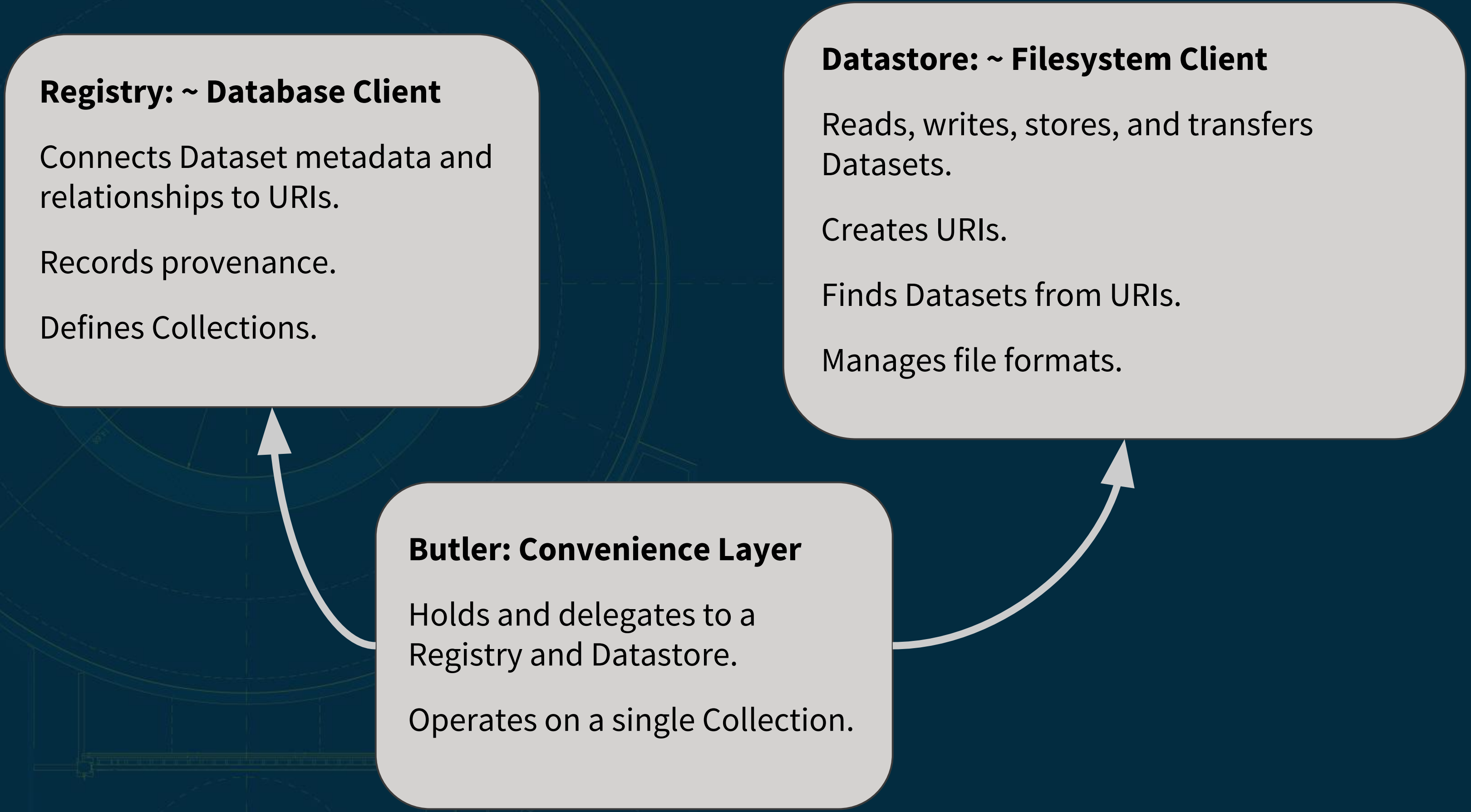
## Basic Concepts

**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. Like a Gen. 2 Data Repository. A Dataset can be associated with multiple Collections.

**DataUnit:** a unit of data that can be used to label a Dataset. Like a key-value pair in a data ID dictionary in the Gen. 2 Butler (e.g. "Visit 780" or "Sensor 10"), with associated metadata.

# Major 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.*

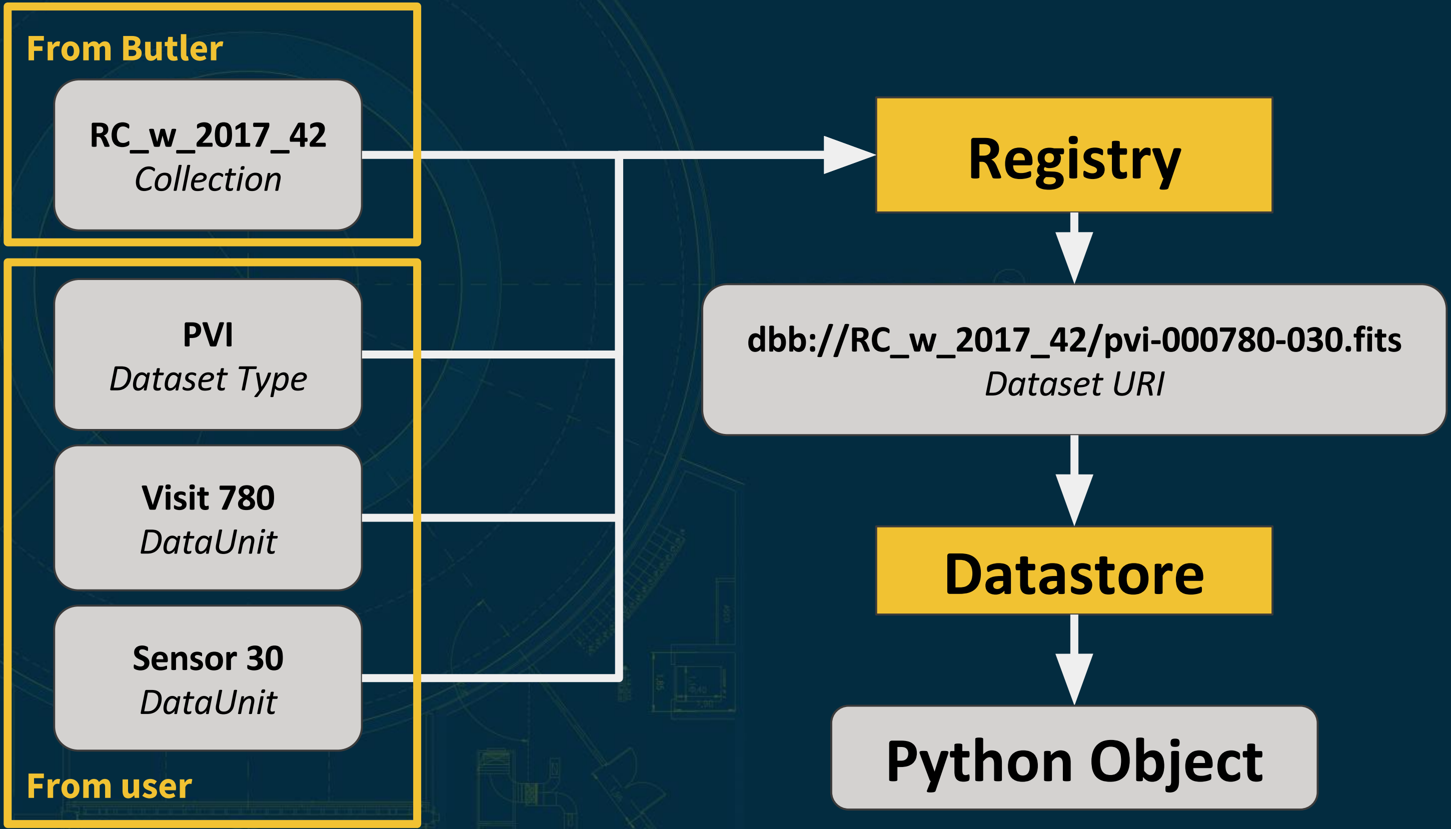
## Datastore

- 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

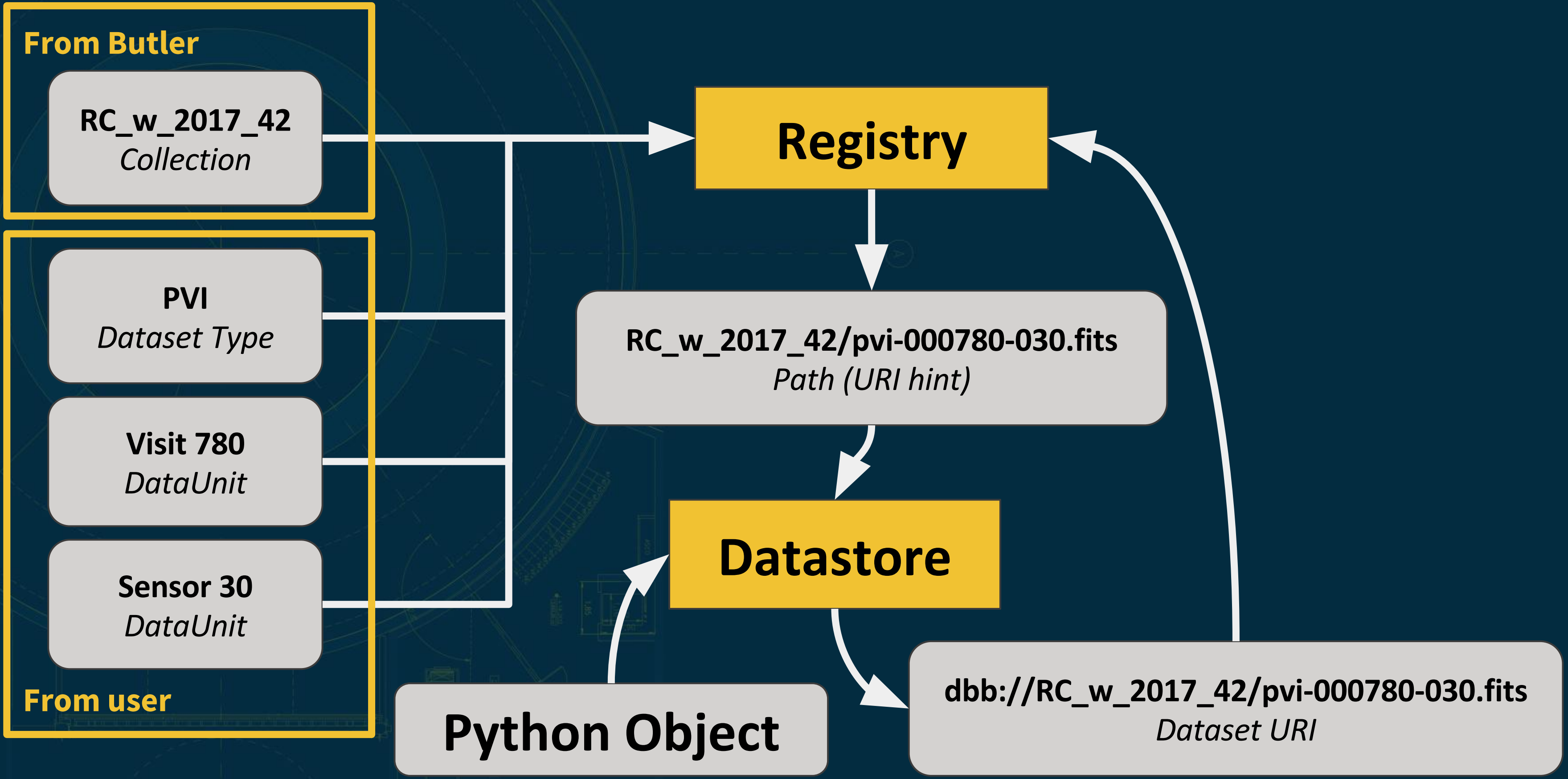
- Combines:
  - a single Collection
  - a Registry instance
  - a Datastore instance
- Can get/put Datasets using DataUnits as labels.
  - Map DataUnits to URIs using Registry.
  - get/put Dataset from Datastore using URIs.
- *Concrete: only one implementation will exist.*

# Butler.get

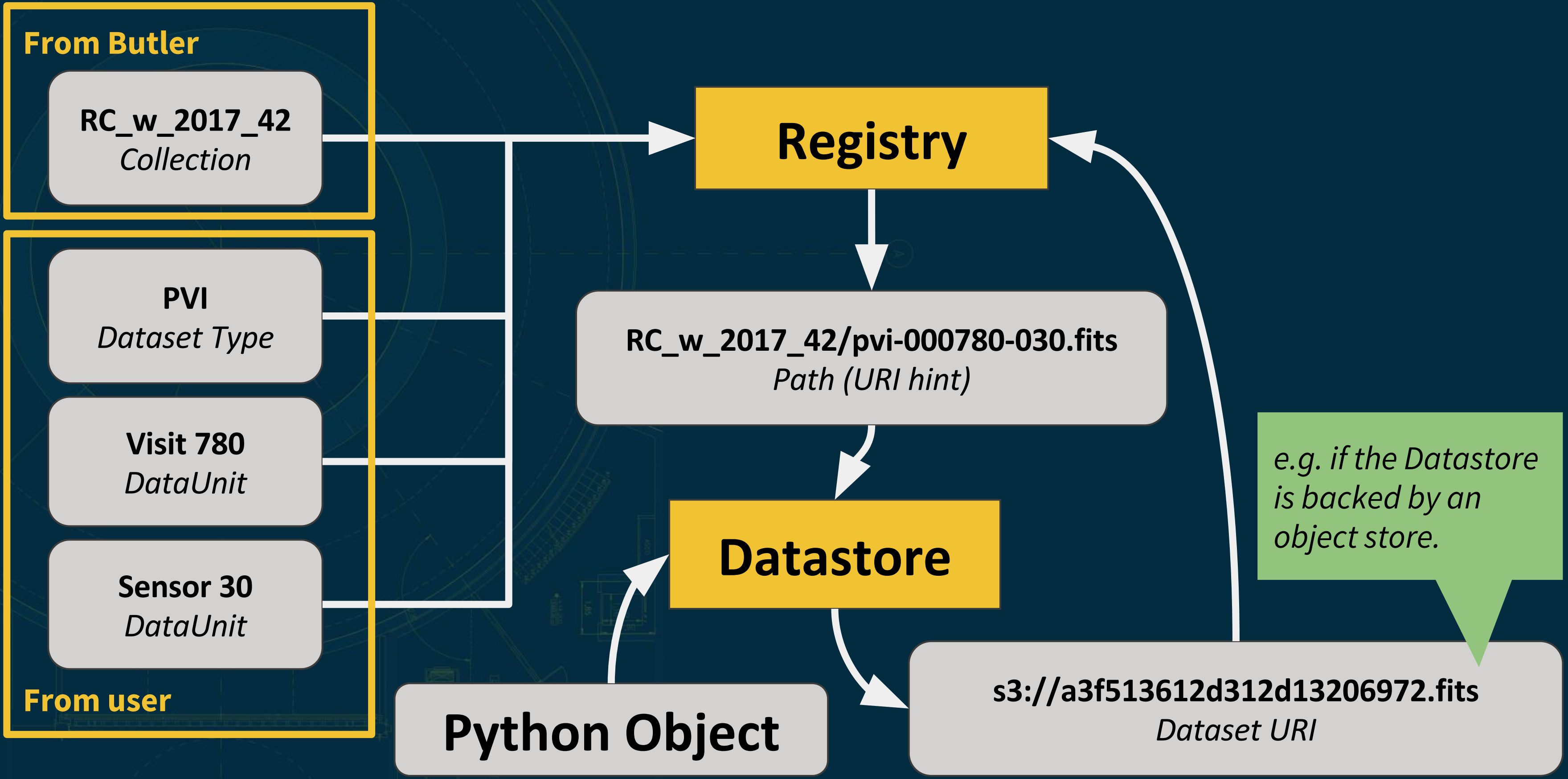


From Butler
RC_w_2017_42
*Collection*
PVI
*Dataset Type*
Visit 780
*DataUnit*
Sensor 30
*DataUnit*
From user
Registry
dbb://RC_w_2017_42/pvi-000780-030.fits
*Dataset URI*
Datastore
Python Object

## 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
e.g. if the Datastore is backed by an object store.

## Proxy Datastores

A Datastore instance can hold instances of other Datastores and delegate work to them.

- **Local caching of remote data:** cache Datasets in a dict keyed by URI, delegate to remote Datastore when requested URI isn't in the dict (n.b. Registry is not updated with local URIs).
- **Layer personal storage and Data Release products:**
  - Delegate writes to personal storage Datastore.
  - Dispatch reads to either Data Backbone Datastore or personal storage Datastore via URI prefixes.

## Proxy Registries

A proxy Registry needs to be implemented in the SQL database, not just the Python client, because a SQL query can involve joins between tables belonging to the proxy and its target.

- **Layer personal storage and Data Release products:**
  - Use most tables from Data Release Registry as-is.
  - Define Dataset "tables" as views that union the Data Release tables and "MyDB" tables.

## Limited Registries

- *Not* based on a SQL database.
- Just complex enough to support Butler get/put and provenance recording:
    - Sufficient for SuperTask execution
    - Cannot support SuperTask preflight
    - Cannot support provenance queries
- Primarily intended for use on batch worker nodes.
- May be used to support obs_file/processFile use cases (but a full SQLite Registry may be better).
- Concrete: only one implementation will exist.

# DataUnits

- Schema has a table for every DataUnit type (e.g. Visit, Patch)
- Some DataUnits have foreign keys to others (Patch → Tract)
- Some DataUnit types have many-to-many joins (Visit↔Patch)
- Some DataUnits have metadata (e.g. Visit exposure time)
- Each DataUnit table has a join table to the Dataset table
- Cameras and SkyMaps are themselves DataUnits (so a Registry can hold Visits from multiple Cameras)
- *Most* Dataset metadata is actually DataUnit metadata

## Risks

- Can SQL Schema be made performant at scale?
  - Our schema is unlikely to *already* be performant; optimizing at this point is premature, but we're hoping it won't require major changes.
  - Can probably trade off flexibility for performance if needed (i.e. we can bake in more assumptions about pipeline structure into the schema).
- Can we avoid and/or solve the Registry transfer problem?

## Implications for LDF Architecture

This design strongly encourages systems with a centralized, tightly-managed database that has personal, less-managed database areas *on the same server*.

- Data Release database with personal space for science users.
- Production databases with personal space for developers, staff scientists, and CI software agents.

We need SQL joins between personal and global metadata tables to avoid transferring content between DB servers.

# Major Operations Concern

- Gen 3. Butler / SuperTask puts more control logic in science pipelines code rather than high-level configuration or separately added queries.
    - Forces production behavior to be more similar to development behavior (possibly good).
    - Limits flexibility for Operations (esp. bad during pre-campaign testing).
    - Need a detailed look at use cases to understand whether this is a concrete problem or just philosophical differences.
- Lack of concreteness in how things are configured makes it impossible to evaluate that part of the design.

## Major To-Do Items

- Identify concrete Registry and Datastore implementations.
- Design back-end interface for Registry and Datastore.
- Design common components below Datastore interface.
- Design Butler and Datastore configuration interfaces.
- Flesh out Registry SQL schema:
    - Add metadata columns.
    - Add/adjust DataUnits to work with Calibration Products Production.

## Major To-Do Items

- Some high-level interfaces aren't appropriate for Butler, but still require coordination between a Registry and a Datastore:
    - Managing Collections (may involve deleting Datasets)
    - Transferring both Datasets and their their metadata
- Handling composite/multi-file Datasets is divided between Registry and Datastore in a way that is at best complex and confusing.  Should try to shift responsibilities to simplify.
- **Further review by Butler WG and broader DM Team:** this design is complex, and not everyone has digested it well enough to sign off.  Going through use cases in detail helps!

# Reuse vs. Replace

**Registries and small common components:**

- totally new code
- rely heavily on SQLAlchemy

**Datastores:**

- reuse perhaps ~50% of Gen. 2 Butler code in persistence, config
- may be hard to identify the code we can reuse
- rewriting persistence layer is also desirable, but not urgent

**Butler:**

- mostly new code
- not much code overall

## For more information on:

- composite datasets (e.g. WCSs of Exposures)
- Dataset slicing (e.g. retrieving subimages)
- the full set of DataUnits in the common schema
- executing SuperTasks
- transfers between Registries
- recording and reporting provenance

…see DMTN-056; the design can handle all of these, even if we haven't worked out *all* of the details.

## For Discussion

The next few slides are about management, not design, and are here just to kick off the discussion and planning phase that needs happen next.

First: what's the procedure for closing the WG?

- Presumably RFCs to baseline use cases and requirements docs.
- Maybe a live review by DMLT or other non-WG members?
- Design doc *could* be baselined now, but would definitely benefit from some (regular, T/CAM-planned) additional work.

## Transition Plan A

On a long-lived branch, or a set of new packages:

- Develop minimal Gen. 3 Butler:
    - all concrete classes
    - SQLite Registry
    - Limited no-SQL Registry
    - Local POSIX filesystem Datastore
- Finish SuperTask framework.
- Convert all CmdLineTasks to SuperTasks.
- Implement minimal (ctrl_pool-replacement) activator.
- Deprecate and remove Gen. 2 Butler and CmdLineTasks.

# Transition Plan B

On a long-lived branch, or a set of new packages:

- Develop minimal Gen. 3 Butler:
  - all concrete classes
  - SQLite Registry
  - Limited no-SQL Registry
  - Local POSIX filesystem Datastore
- Convert all CmdLineTasks to use Gen. 3 Butler.
- Deprecate and remove Gen. 2 Butler.

Then, on a new branch:

- Finish SuperTask framework.
- Implement minimal (ctrl_pool-replacement) activator.
- Deprecate and remove CmdLineTasks.

## Proposed Work Packages

- Minimal Gen. 3 Butler developed by [PTJ]ims.
    - Jim & Pim wrap up design
    - Pim & Tim do initial implementation
- Gen. 2 Butler features frozen, Nate maintains until it is retired.
- After Minimal Gen. 3 Butler:
    - LDF team implements DBB Registry/Datastore
    - ??? implements layered Data Release + LSP user-space Registries/Datastores.
    - ??? implements developer-space Registries/Datastores.

# Proposed Maintenance/Ownership

- Cannot be a single person (or, IMO, even a single team).
- All teams need to develop expertise on core components.
  - Need to share bug triage/repair load beyond [PTJ]ims.
  - Teams need to support their own Registries/Datastores.
  - Major emergent work on core components assigned case-by-case.
- Still need a to maintain a consistent design/vision.
- Same questions/concerns for SuperTask framework.