Pypsmith: A Pythonic data reduction pipeline framework for the ESO software stack

A declarative approach to writing pipelines – validation, documentation and EDPS classification are derived from the code.

Martin Baláž[1] Hugo Buddelmeijer[5,6] Chi-Hung Yan[2] Kieran Leschinski[1] Oliver Czoske[1]
Gilles Otten[2] Janus Brink[1] Jennifer Karr[2] Benjamin Eisele[1] Wolfgang Kausch[3]
Jessica Pilling[3] Patrick Aschenbrenner[3] Thomas Marquart[4]

[1]Institute for Astrophysics, University of Vienna, Austria
[2]Academia Sinica, Institute for Astronomy and Astrophysics, Taipei, Taiwan
[3]University of Innsbruck, Austria
[4]Uppsala University, Sweden
[5]Entropynaut
[6]University of Groningen

ESO software stack

Since 2023, use of Python in ESO pipelines is officially supported. This brings an opportunity to work at a higher level of abstraction and eliminate a lot of boilerplate.

METIS is the first ELT instrument to take advantage of this. Rather than deliver a standalone pipeline in Python, our team decided to

- abstract away all reusable core components (data flow, I/O, base classes)
- implement the METIS pipeline as a module
- keep a simple interface for future instruments

The framework needs to be fully compatible with the ESO software ecosystem, so that the pipelines run under `EDPS` [1] and `PyEsoRex` [2]. ESO expects the use of `PyCPL` and `PyHDRL` functions [3], [4] where possible.

Caveats

`PyCPL` is still under development. Workarounds need to be written before bug fixes appear. The framework maintains isolation: when a fix arrives, only the internal wrapper needs to change.

Best of both worlds

Python is well suited when high flexibility, conciseness and deeper abstraction outweigh the drawbacks of lower performance.

Managing the data flow, dealing with I/O and processing metadata benefit greatly, and intensive computations are offloaded to `NumPy` or C.

Any colour you like

Algorithms can be integrated at several levels:

- **Python functions**: for prototyping; performance is typically poor
- **Python + `NumPy`**: detector data vectorise naturally, 10–100× faster
- **`PyCPL` / `PyHDRL`**: mandatory in ESO pipelines where the functionality exists
- **C functions**: where `NumPy` is still too slow, or existing C code can be reused

Design priorities

Use the abstractions

A naive rewrite of C code to Python does more harm than good. Abstractions of a high-level, object-oriented language save a lot of effort when used correctly.

Keep cognitive load low

Recipe authors need not care about the data flow. Defining the inputs, outputs and the transformation between them is sufficient. All pixel manipulation is then performed in one method.

Declarations first

All objects and relationships are defined beforehand and can be inspected without loading actual data. Abort early if any critical part is missing.

Boilerplate only written once

Registration, parameter handling, header propagation, error handling and cleanup are performed by the engine, not by the recipe.

With *Pypsmith*, a typical 800-line C recipe shrinks to about 60 lines of Python.

Variants by subclassing or parameters

Recipes with similar functionality share one definition and a base test suite. Differences are handled by targeted overrides or parameters.

Declare, extend, parameterise

Many recipes and data items only differ in a small set of parameters, such as spectral band, detector, or target. Parameterisable parts are either specialised explicitly at definition time with mixins, or can be determined from supplied data at runtime. Every data item has a mandatory schema and is validated against it at creation time.

A minimal example of a recipe is shown in Figure 1. Input validation, `EDPS` classification, the recipe card, the flowchart, the manual page and the association maps are all derived from these declarations.

```python
class MetisLmImgBasicReduceImpl(BandLmMixin, Detector2rgMixin, DarkImageProcessor, MetisRecipeImpl):
    class InputSet(DarkImageProcessor.InputSet):  # inherits master dark automatically
        class RawInput(RawInput):
            Item = ImageRaw                       # LM_IMAGE_{target}_RAW
        class MasterFlatInput(MasterFlatInput):
            Item = MasterImgFlat                  # MASTER_IMG_FLAT_{source}_LM
        raw: RawInput
        master_flat: MasterFlatInput

    class ProductSet(PipelineProductSet):
        BasicReduced = products.BasicReduced      # LM_{target}_BASIC_REDUCED

    class Qc(QcParameterSet):
        class Median(QcParameter):
            _name_template = "QC LM IMG MEDIAN"
            _type, _unit = float, "counts"

    def process(self) -> set[DataItem]:
        images = self.inputset.raw.load_data('DET1.DATA')
        dark = self.inputset.master_dark.load_data('DET1.SCI') # extension must match the schema
        images.subtract_image(dark)
        images.divide_image(self.inputset.master_flat.load_data('DET1.SCI'))
        return {
            self.ProductSet.BasicReduced(header, Hdu(qc, image))
            for image in images
        }

class MetisLmImgBasicReduce(Recipe):              # the pyesorex plugin
    _name = "metis_lm_img_basic_reduce"
    _synopsis = "Basic science image data processing"
    _matched_keywords = frozenset({'DET.DIT', 'DET.NDIT', 'DRS.FILTER'})
    parameters = ParameterList([
        ParameterEnum(f"{_name}.stacking.method", default="add",
                      alternatives=("add", "average", "median"))])
    Impl = MetisLmImgBasicReduceImpl
```

Figure 1: Every recipe is declaration-first: all inputs, products and quality control parameters are classes. The entire imperative part is defined inside `process` and will be called automatically.

Introspection

Documentation is generated directly from the pipeline. It cannot drift from the code and inconsistencies are easy to spot. Metadata are rendered with `Jinja2` templates and `TikZ` into data cards, flowcharts and association maps, such as Figures 2–4. Longer text, e.g. algorithm descriptions, is stored externally. The whole is composed into a LaTeX or `typst` document.

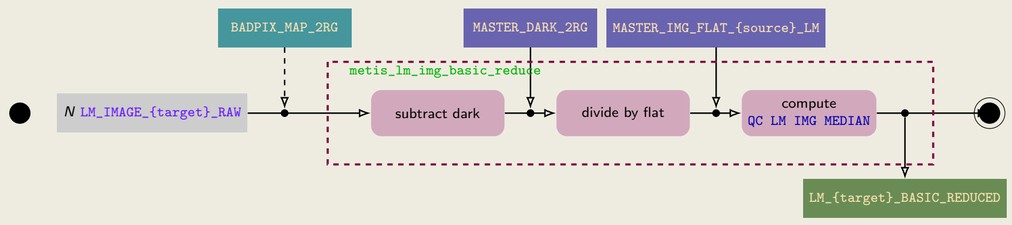


Figure 2: The processing flow of the recipe in Figure 1, as drawn from the code.

Name:	metis_lm_img_basic_reduce
Purpose:	Basic science image data processing
Input data:	LM_IMAGE_{target}_RAW [N] BADPIX_MAP_2RG [1, optional] MASTER_DARK_2RG [1] MASTER_IMG_FLAT_{source}_LM [1]
Matched keywords:	DET.DIT DET.NDIT DRS.FILTER
Parameters:	metis_lm_img_basic_reduce . stacking . method: Method used to combine the input images (add, average, median), default add
Algorithm:	Subtract dark, divide by flat
Output data:	LM_{target}_BASIC_REDUCED
QC1 parameters:	QC LM IMG MEDIAN

Figure 3: The recipe card, rendered from the declarations in Figure 1 by a generator.

Many more operations on structured data are possible, such as

- automatic comparison to specification (external audit script)
- building `pyesorex` manual pages
- generating `EDPS` workflows from recipes (planned)

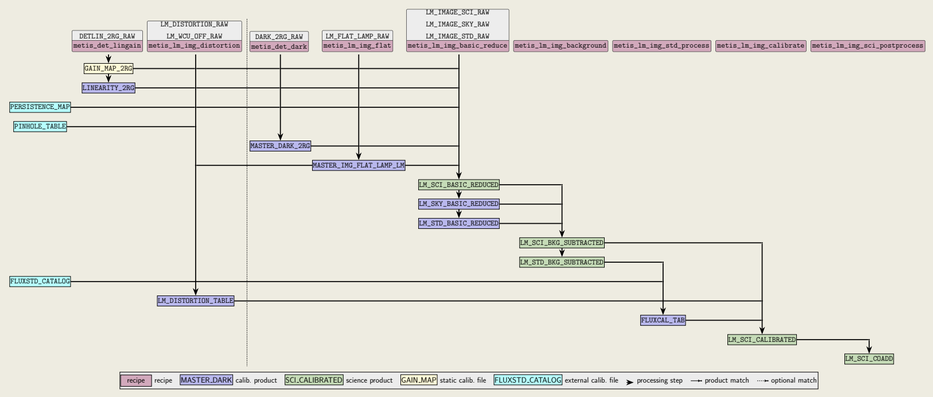


Figure 4: The association map for the LM imaging mode of METIS, generated with a `TikZ` template from the Python recipes. This map alone replaced 440 lines of hand-written LaTeX; 19 such figures are generated in the same way.

Modular design

Pypsmith provides two core modules:

- **Engine**, which manages I/O and data flow, and defines base classes for instruments. It also provides hooks for custom processing steps. Written purely in Python.
- **Data Reduction Library**, a collection of pure functions operating on pixel data. `CPL` and `HDRL` are used directly; new functions are mostly written in `NumPy`. Available C functions can be wrapped and reused.

Instruments

Each instrument is defined as a separate Python module. It needs to provide definitions of data items, processing steps and recipes as subclasses of core classes. Instrument properties (detector sizes, positions, etc.) and custom algorithms have to be defined there as well.

Testing and status

The class hierarchy allows for parameterising tests over recipes and data items for all possible specialisations. For METIS, full simulated data are provided by `ScopeSim` [5].

Tests cover a broad spectrum of conditions:

- static assertions, e.g. that recipe metadata are set
- simple dynamic tests, e.g. that each declared QC parameter is actually written
- `EDPS` classification on mocked FITS headers with no data
- end-to-end runs on simulated data (`ScopeSim` simulation + `EDPS`)

The METIS pipeline is past the skeleton stage and currently defines 43 recipes in 10 `EDPS` workflows. Experimental support for MICADO is underway, as a "shadow" pipeline in addition to the official delivery for ESO (in C).

We want you

Introduction of additional instruments to *Pypsmith*, including **yours**, is very welcome – and would help cover use cases not considered so far.

Release and licence

Pypsmith is released under GPLv3, along with all bundled pipelines, and currently published on GitHub as `METIS_Pipeline`. `PyCPL` and `PyHDRL` are available from ESO repositories.

References

[1] W. Freudling *et al.*, "Adaptive data reduction workflows for astronomy: The ESO Data Processing System (EDPS)," *Astronomy & Astrophysics*, vol. 681, p. A93, 2024, doi: 10.1051/0004-6361/202347651.

[2] European Southern Observatory, "PyEsoRex – ESO Recipe Execution Tool for PyCPL." [Online]. Available: https://www.eso.org/sci/software/pycpl/pycpl-site/

[3] M. S. Deshpande *et al.*, "PyCPL: The ESO Common Pipeline Library in Python v1.0." 2024.

[4] A. J. Horton *et al.*, "Python and pipelines: Python interfaces for ESO's CPL and HDRL data reduction libraries," in *Software and Cyberinfrastructure for Astronomy VIII*, in Proc. SPIE, vol. 13101. 2024, p. 131013Q. doi: 10.1117/12.3019678.

[5] K. Leschinski *et al.*, "ScopeSim: a flexible general purpose astronomical instrument data simulation framework in Python," in *Software and Cyberinfrastructure for Astronomy VI*, in Proc. SPIE, vol. 11452. 2020, p. 114521Z. doi: 10.1117/12.2559784.