BioPipelines User Manual

BioPipelines User Manual

Index

BioPipelines

Tool Reference

Architecture Overview

What is BioPipelines?

BioPipelines is a Python framework that generates bash scripts for bioinformatics workflows. It does not execute computations directly - instead, it predicts the filesystem structure and creates scripts that will be executed on SLURM clusters.

Core Concepts

Pipeline: The main coordinator that orchestrates tools, manages the folder structure, and switches environments. The following code generates a unique folder /shares/*USER*/MyPipeline/JobName*_NNN* where all the output will be. **WARNING**: Don't include blank spaces in the pipeline and job names.

```
from PipelineScripts.pipeline import Pipeline
pipeline = Pipeline("MyPipeline", "JobName", "Description")
```

Tools: Individual bioinformatics operations like running models (RFdiffusion, LigandMPNN, Boltz2, …) or analyzing results (Filter, SelectBest, …) that generate bash scripts and predict their outputs. They are added to the pipeline by calling `pipeline.add(<Tool>(<Parameters>))`, which returns an object containing predictions of the filesystem after SLURM execution. The prediction can be used as input in subsequent tools. One can access the prediction with the default `print(<prediction>)` method.

```
from PipelineScripts.pipeline import Pipeline
from PipelineScripts.rfdiffusion import RFdiffusion
pipeline = Pipeline("Pipeline","Test","Some test")
rfd = pipeline.add(RFdiffusion(contigs="50-100", num_designs=5))
print(rfd)
"""
structures:
    - '<output_folder>/Test_1.pdb'
    - ...
    - '<output_folder>/Test_5.pdb'
structure_ids:
    - Test_1
    - ...
    - Test_5
datasheets:
    $structures (id, source_id, pdb, fixed, designed, contigs, time, status):
        - '<output_folder>/rfdiffusion_results.csv'
output_folder:
    - 'path/to/001_RFdiffusion'
```

```
main:
    - '<output_folder>/rfdiffusion_results.csv'
#Aliases
structures=pdbs
"""
```

The predicted output can then be used by other tools as input, thus enabling further prediction prior to the actual generation of the files:

```
from PipelineScripts.pipeline import Pipeline
from PipelineScripts.rfdiffusion import RFdiffusion
from PipelineScripts.protein_mpnn import ProteinMPNN
pipeline = Pipeline("Pipeline","Test","Some test")
rfd = pipeline.add(RFdiffusion(contigs="50-100", num_designs=5))
pmd = pipeline.add(ProteinMPNN(structures=rfd,num_sequences=2))
print(pmd)
"""
sequences:
    - '<output_folder>/Test_queries.csv'
sequence_ids:
    - Test_1_1
    - Test_1_2
    - ...
    - Test_5_1
    - Test_5_2
datasheets:
    $sequences (id, source_id, source_pdb, sequence, score, seq_recovery, rmsd):
        - '<output_folder>/Test_queries.csv'
output_folder:
    - 'path/to/002_ProteinMPNN'
fa_files:
    - '<output_folder>/seqs/Test_sequences.fa'
main:
    - '<output_folder>/proteinmpnn_results.csv'
queries_fasta:
    - '<output_folder>/Test_queries.fasta'
seqs_folder:
    - '<output_folder>/seqs'
#Aliases
sequences=queries_csv
"""
```

All the outputs are standardized such that the identity of the previous tool is not needed for the prediction to be used as input, and in general tools have to be developed agnostic of previous tool identities. Furthermore, this systems allows:

1. Verification of the success or failure of a given tool. At slurm runtime, after running a tool, the pipeline coordinator check for the presence of the predicted output and creates a file `<NNN>_<Tool>_COMPLETED` or `<NNN>_<Tool>_FAILED`. One can resubmit the same slurm bash script(for example, if the time ran out) and the completed steps will be skipped.
2. Standardized saving and loading of tool outputs. All the predictions are saved in the folder `ToolOutputs` within the job folder, and can be loaded with the tool LoadOutput. In the following example, rfd behaves identically to the one in the previous snippet:

```
from PipelineScripts.pipeline import Pipeline
from PipelineScripts.load_output import LoadOutput
```

```python
from PipelineScripts.protein_mpnn import ProteinMPNN
pipeline = Pipeline("Pipeline","Test","Some test")
rfd = pipeline.add(LoadOutput('/path/to/ToolOutputs/001_RFdiffusion.json'))
pmd = pipeline.add(ProteinMPNN(structures=rfd,num_sequences=2))
```

Job submission

Call to generate the validate the pipeline, generate the full scripts, and generate a slurm file that can be executed on the cluster. Importantly, this will not result in the submission to slurm. For this you have to either: 1. Run from console the sbatch printed at the end when running the pipeline python script; 2. Copy-paste the job name and slurm content in the job composer form (https://apps.s3it.uzh.ch/pun/sys/myjobs); 3. Instead of running the pipeline with `python /path/to/<pipeline>.py`, use `./submit /path/to/<pipeline.py>` to both run and submit. However, this requires pandas to be executable.

```
pipeline.slurm()
"""
==============================Job============================== # Suggested job name
Pipeline: Test (002)
============================Slurm Script============================ # Slurm script for manual subm
#!/usr/bin/bash
#SBATCH --mem=16GB
#SBATCH --time=24:00:00
#SBATCH --output=job.out
#SBATCH --begin=now+0hour
# Make all files group-writable by default
umask 002
module load mamba singularityce
# Execute pipeline
path/to/RunTime/pipeline.sh    # Do not run this: it is part of the slurm script, it will most likely f
==============================SBATCH============================ # Simple submission using sbatch com
sbatch --job-name=Test --output path/to/RunTime/slurm.out path/to/RunTime/slurm.sh
"""
```

Filesystem Structure

After the execution, the filesystem will look somewhat like this:

```
/shares/<user>/BioPipelines/<pipeline>/<job>_<NNN>/
  RunTime/                    # Execution scripts only
      pipeline.sh             # Main orchestration script
      001_<tool 1>.sh         # Tool-specific script
      002_<tool 2>.sh
      ...
  ToolOutputs/                # Tool output predictions
      001_<tool 1>.json
      002_<tool 2>.json
      ...
  001_<Tool 1>/               # Tool outputs only
      <Some temporary folder>
      structure_1.pdb
      structure_2.pdb
      ...
      <results>.csv
  002_LigandMPNN/
      <Some temporary folder>
      <sequence queries>.csv
```

```
    <sequence queries>.fasta
    ...
_001_rfdiffusion.log        # Execution logs
_002_ligandmpnn.log
```

Environment Management

Most tools require a conda environment to run. To specify something different from the default, one has to add the parameter `env` to the `pipeline.add(...)` method:

```
# Use default environment from tool definition
tool1 = pipeline.add(Tool1(...))
# Override with custom environment
tool2 = pipeline.add(Tool2(...), env="CustomEnv")
```

Tool I/O Reference Guide

Overview

All tools return outputs through a unified `StandardizedOutput` object with general structure:

```
structures        # List[str] - File paths e.g. /path/to/structure_1.pdb, ...
structure_ids     # List[str] - Structure identifiers e.g. structure_1, ...
compounds         # [str]     - Path to csv file with columns id, format, smiles, ccd
compound_ids      # List[str] - Compound identifiers
sequences         # [str]     - Path to csv file with columns id, sequence, ...
sequence_ids      # List[str] - Sequence identifiers
msas              # [str]     - Path to csv file with columns id, sequence, msa, ...
msa_ids           # List[str] - MSA identifiers
datasheets        # List[DI]  - Contains objects of class DatasheetInfo
tool_folder       # [str]     - Path to output directory
```

Datasheet Organization

Datasheet names are tool-dependent. Accessing a datasheet with dot notation will return the path to the csv file. For accessing the metadata, use the info function.

```
# Simple path
path = <tool output>.datasheets.analysis  # path/to/analysis.csv

# Rich metadata access
info = <tool output>.datasheets.info("analysis")
print(info.path)         # path/to/analysis.csv
print(info.columns)      # ["id", ...]
print(info.description)  # "Results from ..."
print(info.count)        # Number of entries
```

Datasheet Column References

Tools can explicitly reference specific columns from upstream datasheets using dot notation .datasheets.., which returns the tuple (DatasheetInfo,str).

```
lmpnn = LigandMPNN(
    structures=rfdaa, #output from RFdiffusionAllAtom
    ligand='LIG',
    redesigned=rfdaa.datasheets.structures.designed     #structures has columns id, fixed, designed, ..
)
```

Troubleshooting

Common Issues

Path errors: Ensure running from BioPipelines root directory.

Environment issues: Check conda environment availability and tool compatibility.

Resource limits: Adjust GPU/memory requirements in `pipeline.resources()`.

Missing files: Check logs in `_<N>_<tool>.log` files.

Debug Mode

You can test the filesystem prediction locally by instantiating the pipeline object with the debug flag set as True:

```
pipeline = Pipeline(..., debug=True)
```

Complete BioPipelines Tool Reference

This document contains the complete, verified tool reference with all parameters, default environments, and output specifications.

Structure Generation

RFdiffusion

Environment: `ProteinEnv`

Parameters: - `pdb`: Union[str, ToolOutput, StandardizedOutput] = "" - Input PDB template (optional) - `contigs`: str (required) - Contig specification (e.g., "A1-100", "10-20,A6-140") - `inpaint`: str = "" - Inpainting specification - `num_designs`: int = 1 - Number of backbone designs to generate - `active_site`: bool = False - Use active site model for small motifs (<30 residues) - `steps`: int = 50 - Number of diffusion steps - `partial_steps`: int = 0 - Partial diffusion steps - `reproducible`: bool = False - Use deterministic sampling - `design_startnum`: int = 1 - Starting number for design IDs

Outputs: - `structures`: List of generated PDB files - `datasheets.structures`:

id	source_id	pdb	fixed	designed	contigs	time	status

Example:

```
rfd = pipeline.add(RFdiffusion(
    contigs="50-100",
    num_designs=10
))
```

RFdiffusionAllAtom

Environment: `ProteinEnv`

Parameters: - `ligand`: str (required) - Ligand identifier in PDB (e.g., 'LIG', 'ATP') - `pdb`: Union[str, ToolOutput, StandardizedOutput] = "" - Input PDB template - `contigs`: str (required) - Contig specification - `inpaint`: str = "" - Inpainting specification - `num_designs`: int = 1 - Number of designs - `active_site`: bool = False - Use active site model - `steps`: int = 200 - Diffusion steps (default higher for AllAtom) - `partial_steps`: int = 0 - Partial diffusion steps - `reproducible`: bool = False - Deterministic sampling -

`design_startnum`: int = 1 - Starting design number - `ppi_design`: bool = False - Enable protein-protein interface design - `ppi_hotspot_residues`: List[str] = None - Hotspot residues for PPI - `ppi_binder_length`: int = None - Length of PPI binder - `autogenerate_contigs`: bool = False - Auto-infer fixed contig segments - `model_only_neighbors`: bool = False - Only remodel residues near ligand - `num_recycles`: int = 1 - Number of diffusion recycles - `scaffold_guided`: bool = False - Enforce strict adherence to scaffold - `align_motif`: bool = True - Pre-align functional motif - `deterministic`: bool = False - Fixed RNG seeds - `inpaint_str`: str = None - Secondary structure pattern for inpainting - `inpaint_seq`: str = None - Sequence pattern for inpainting - `inpaint_length`: int = None - Target length for inpainted regions - `guiding_potentials`: str = None - Custom external potentials

Outputs: - `structures`: List of generated PDB files - `datasheets.structures`:

id	source_id	pdb	fixed	designed	contigs	time	status

Example:

```
rfdaa = pipeline.add(RFdiffusionAllAtom(
    pdb=template,
    ligand='LIG',
    contigs='10-20,A6-140',
    num_designs=5
))
```

Sequence Design

ProteinMPNN

Environment: `ProteinEnv`

Parameters: - `structures`: Union[str, List[str], ToolOutput] (required) - Input structures - `datasheets`: Optional[List[str]] = None - Input datasheet files - `num_sequences`: int = 1 - Number of sequences per structure - `fixed`: str = "" - Fixed positions (PyMOL selection or datasheet reference) - `redesigned`: str = "" - Redesigned positions (PyMOL selection or datasheet reference) - `fixed_chain`: str = "A" - Chain to apply fixed positions - `plddt_threshold`: float = 100.0 - pLDDT threshold for automatic fixing (residues above threshold are fixed) - `sampling_temp`: float = 0.1 - Sampling temperature - `model_name`: str = "v_48_020" - ProteinMPNN model variant - `soluble_model`: bool = True - Use soluble protein model

Outputs: - `sequences`: CSV file with generated sequences - `datasheets.sequences`:

id	source_id	source_pdb	sequence	score	seq_recovery	rmsd

Note: Sample 0 is the original/template sequence, samples 1+ are designs.

Example:

```
pmpnn = pipeline.add(ProteinMPNN(
    structures=rfd,
    num_sequences=10,
    fixed="1-10+50-60",
    redesigned="20-40"
))
```

LigandMPNN

Environment: `ligandmpnn_env`

Parameters: - `structures`: Union[str, List[str], ToolOutput] (required) - Input structures - `ligand`: str (required) - Ligand identifier for binding site focus - `datasheets`: Optional[List[str]] = None - Input datasheet files - `name`: str = "" - Job name for output files - `num_sequences`: int = 1 - Number of sequences to generate per structure - `fixed`: str = "" - Fixed positions (LigandMPNN format "A3 A4 A5" or datasheet reference) - `redesigned`: str = "" - Designed positions (LigandMPNN format or datasheet reference) - `design_within`: float = 5.0 - Distance in Angstroms from ligand for post-generation analysis only (does not control design). For actually designing residues within a distance, use DistanceSelector to select positions first. - `model`: str = "v_32_010" - LigandMPNN model version (v_32_005, v_32_010, v_32_020, v_32_025) - `batch_size`: int = 1 - Batch size for processing

Outputs: - `sequences`: CSV file with generated sequences - `datasheets.sequences`:

id	sequence	sample	T	seed	overall_confidence	ligand_confidence	seq_rec

Example:

```
lmpnn = pipeline.add(LigandMPNN(
    structures=rfdaa,
    ligand="LIG",
    num_sequences=5,
    redesigned=rfdaa.datasheets.structures.designed
))
```

MutationComposer

Environment: `MutationEnv`

Parameters: - `frequencies`: Union[List, ToolOutput, StandardizedOutput, DatasheetInfo, str] (required) - Mutation frequency datasheet(s) from MutationProfiler - `num_sequences`: int = 10 - Number of sequences to generate - `mode`: str = "single_point" - Generation strategy: - "single_point": One mutation per sequence - "weighted_random": Random mutations weighted by frequency - "hotspot_focused": Focus on high-frequency positions - "top_mutations": Use only top N mutations - `min_frequency`: float = 0.01 - Minimum frequency threshold for mutations - `max_mutations`: int = None - Maximum mutations per sequence - `random_seed`: int = None - Random seed for reproducibility - `prefix`: str = "" - Prefix for sequence IDs - `hotspot_count`: int = 10 - Number of top hotspot positions (for hotspot_focused mode) - `combination_strategy`: str = "average" - Strategy for combining multiple datasheets (average, maximum, stack, round_robin)

Outputs: - `sequences`: CSV file with composed sequences - `datasheets.sequences`:

id	sequence	mutations	mutation_positions

Example:

```
profiler = pipeline.add(MutationProfiler(original=ref, mutants=variants))
composer = pipeline.add(MutationComposer(
    frequencies=profiler.datasheets.relative_frequencies,
    num_sequences=50,
    mode="weighted_random",
```

```
    max_mutations=5
))
```

SDM (SiteDirectedMutagenesis)

Environment: `ProteinEnv`

Parameters: - `original`: Union[str, ToolOutput, StandardizedOutput] (required) - Input structure/sequence - `position`: int (required) - Target position for mutagenesis (1-indexed) - `mode`: str = "saturation" - Mutagenesis strategy: - "saturation": All 20 amino acids - "hydrophobic": Hydrophobic residues only - "hydrophilic": Hydrophilic residues only - "charged": Charged residues only - "polar": Polar residues only - "nonpolar": Nonpolar residues only - "aromatic": Aromatic residues only - "aliphatic": Aliphatic residues only - "positive": Positively charged residues only - "negative": Negatively charged residues only - `include_original`: bool = False - Include original amino acid in output - `exclude`: str = "" - Amino acids to exclude (single letter codes as string, e.g., "CP") - `prefix`: str = "" - Prefix for sequence IDs

Outputs: - `sequences`: CSV file with mutant sequences - `datasheets.sequences`:

id	sequence	mutation	position	original_aa	new_aa

- `datasheets.missing_sequences`:

id	sequence	reason

Example:

```
sdm = pipeline.add(SDM(
    original=template,
    position=42,
    mode="saturation",
    exclude="CP"
))
```

Fuse

Environment: `ProteinEnv`

Parameters: - `proteins`: Union[List[str], str] (required) - List of protein sequences or PDB file paths - `sequences`: Union[List[str], str] = None - Alias for proteins - `name`: str = "" - Job name for output files - `linker`: str = "GGGGSGGGGSGGGGSGGGGS" - Linker sequence that will be cut based on `linker_lengths` if specified - `linker_lengths`: List[str] = None - List of length ranges for each junction to generate multiple variants by cutting the linker (e.g., ["1-6", "1-6"])

Outputs: - `sequences`: CSV file with fused sequences - `datasheets.sequences`:

id	sequence	lengths

Example:

```
fused = pipeline.add(Fuse(
    proteins=[domain1, domain2, domain3],
    linker="GGGGS"
))
```

StitchSequences

Environment: `ProteinEnv`

Parameters: - `sequences`: List[Union[ToolOutput, StandardizedOutput]] (required) - List of sequence outputs to stitch - `selections`: Union[List[Union[str, ToolOutput]], str] = None - Position specifications for each sequence (e.g., ["1-50", "51-100"]) - `id_map`: Dict[str, str] = None - ID mapping pattern (default: {" ": "_N"})

Outputs: - `sequences`: CSV file with stitched sequences - `datasheets.sequences`:

id	sequence

Example:

```
stitched = pipeline.add(StitchSequences(
    sequences=[lmpnn1, lmpnn2],
    selections=["1-50", "51-100"]
))
```

Structure Prediction

AlphaFold

Environment: `ProteinEnv`

Parameters: - `sequences`: Union[str, List[str], ToolOutput, Dict[str, Any]] (required) - Input sequences or dict with sequences - `datasheets`: Optional[List[str]] = None - Input datasheet files - `name`: str = "" - Job name - `num_relax`: int = 0 - Number of best models to relax with AMBER - `num_recycle`: int = 3 - Number of recycling iterations - `rand_seed`: int = 0 - Random seed (0 = random)

Outputs: - `structures`: List of predicted PDB files - `datasheets.structures`:

id	source_id	sequence

Example:

```
af = pipeline.add(AlphaFold(
    sequences=lmpnn,
    num_relax=1,
    num_recycle=5
))
```

Boltz2

Environment: `Boltz2Env`

Parameters: - `config`: Optional[str] = None - Direct YAML configuration string - `proteins`: Union[str, List[str], ToolOutput] (required) - Protein sequences - `ligands`: Union[str, ToolOutput, StandardizedOutput, None] = None - Ligand SMILES string or compound library ToolOutput - `msas`: Optional[Union[str, ToolOutput]] = None - Pre-computed MSA files for recycling (pass entire ToolOutput, not .msas) - `sequences`: Union[str, List[str], ToolOutput] = None - Legacy parameter (use proteins instead) - `ligand_smiles`: Optional[str] = None - Legacy parameter (use ligands instead) - `ligand_library`: Optional[str] = None - Path to CSV file with ligand library (deprecated, use CompoundLibrary tool) - `primary_key`: Optional[str] = None - Key column in library to filter by - `library_repr`: str = "SMILES" - Ligand representation (SMILES, CCD) - `library_type`: str = "noncovalent" - Binding type (noncovalent, covalent) - `affinity`: bool = True - Calculate binding affinity predictions - `output_format`: str = "pdb" - Output format (pdb, mmcif) - `msa_server`: str = "public" - MSA generation (public, local) - `global_msas_cache`: bool = False - Enable global MSA caching across jobs - `recycling_steps`: Optional[int] = None - Number of recycling steps (default: model-specific) - `diffusion_samples`: Optional[int] = None - Number of diffusion samples (default: model-specific) - `use_potentials`: bool = False - Enable external potentials

Outputs: - `structures`: List of predicted complex PDB files - `datasheets.confidence`:

id	input_file	confidence_score	ptm	iptm	complex_plddt	complex_iplddt

- `datasheets.affinity`:

id	input_file	affinity_pred_value	affinity_probability_binary

Example:

```
boltz_apo = pipeline.add(Boltz2(proteins=lmpnn))
boltz_holo = pipeline.add(Boltz2(
    proteins=lmpnn,
    ligands="CC(=O)OC1=CC=CC=C1C(=O)O",  # Aspirin SMILES
    msas=boltz_apo,  # Pass entire ToolOutput
    affinity=True
), env="Boltz2Env")
```

Analysis

ResidueAtomDistance

Environment: `ProteinEnv`

Parameters: - `structures`: Union[ToolOutput, StandardizedOutput] (required) - Input structures - `atom`: str (required) - Atom selection (e.g., 'LIG.Cl', 'name CA', 'A10.CA') - `residue`: str (required) - Residue selection (e.g., 'D in IGDWG', '145', 'resn ALA') - `method`: str = "min" - Distance calculation method (min, max, mean, closest) - `metric_name`: str = None - Custom name for distance column in output (default: "distance")

Outputs: - `datasheets.analysis`:

id	source_structure	{metric_name}

Example:

```
distances = pipeline.add(ResidueAtomDistance(
    structures=boltz,
    atom="LIG.Cl",
    residue="D in IGDWG",
    method="min",
    metric_name="chlorine_distance"
))
```

PLIP (Protein-Ligand Interaction Profiler)

Environment: `ProteinEnv`

Note: This tool is not fully debugged yet and may require adjustments.

Parameters: - `structures`: Union[str, List[str], ToolOutput, StandardizedOutput] (required) - Input structures - `ligand`: str = "" - Specific ligand identifier (if empty, analyzes all ligands) - `output_format`: List[str] = None - Output formats (default: ['xml', 'txt', 'pymol']) - `create_pymol`: bool = True - Generate PyMOL session files - `create_images`: bool = False - Generate ray-traced images - `analyze_peptides`: bool = False - Include protein-peptide interactions - `analyze_intra`: bool = False - Include intra-chain interactions - `analyze_dna`: bool = False - Include DNA/RNA interactions - `max_threads`: int = 4 - Maximum threads for parallel processing - `verbose`: bool = True - Enable verbose output

Outputs: - `datasheets.interactions`:

id	ligand_id	interaction_type	residue	distance	angle	energy

Example:

```
plip = pipeline.add(PLIP(
    structures=boltz,
    ligand="LIG",
    create_pymol=True
))
```

DistanceSelector

Environment: `ProteinEnv`

Parameters: - `structures`: Union[str, List[str], ToolOutput] (required) - Input structures - `ligand`: str (required) - Ligand identifier for distance reference - `distance`: float = 5.0 - Distance cutoff in Angstroms - `reference_type`: str = "ligand" - Type of reference (ligand, atoms, residues) - `reference_selection`: str = "" - Specific PyMOL selection if not using ligand

Outputs: - `datasheets.selections`:

id	pdb	within	beyond	distance_cutoff	reference_ligand

Example:

```
selector = pipeline.add(DistanceSelector(
    structures=boltz,
    ligand="ATP",
    distance=8.0
))
```

ConformationalChange

Environment: `ProteinEnv`

Note: This tool is not fully debugged yet and may require adjustments.

Parameters: - `reference_structures`: Union[str, ToolOutput, StandardizedOutput] (required) - Reference structures - `target_structures`: Union[ToolOutput, StandardizedOutput] (required) - Target structures to compare - `selection`: Union[str, ToolOutput] (required) - Region specification (PyMOL selection or datasheet reference) - `alignment`: str = "align" - Alignment method (align, super, cealign)

Outputs: - `datasheets.conformational_analysis`:

id	reference_structure	target_structure	selection	num_residues	RMSD	max_distance	mean_distance	sum_over_square_root

Example:

```
conf_change = pipeline.add(ConformationalChange(
    reference_structures=apo_structures,
    target_structures=holo_structures,
    selection="resi 10-50",  # PyMOL selection
    alignment="super"
))
```

MutationProfiler

Environment: `MutationEnv`

Parameters: - `original`: Union[ToolOutput, StandardizedOutput] (required) - Original/reference sequences - `mutants`: Union[ToolOutput, StandardizedOutput] (required) - Mutant sequences to analyze - `include_original`: bool = True - Include original sequence in frequency analysis

Outputs: - `datasheets.profile`:

position	original	count	frequency

- `datasheets.mutations`:

position	original	A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y

- `datasheets.absolute_frequencies`:

position	original	A	C	…	Y

- `datasheets.relative_frequencies`:

position	original	A	C	…	Y

Example:

```
profiler = pipeline.add(MutationProfiler(
    original=template,
    mutants=lmpnn
))
```

ProteinLigandContacts

Environment: `ProteinEnv`

Note: This tool is not fully debugged yet and may require adjustments.

Parameters: - `structures`: Union[str, List[str], ToolOutput, StandardizedOutput] (required) - Input structures - `ligand`: str (required) - Ligand identifier - `contact_distance`: float = 4.5 - Distance cutoff for defining contacts (Angstroms) - `contact_types`: List[str] = None - Types of contacts to analyze (default: all)

Outputs: - `datasheets.contacts`:

id	ligand_id	residue	contact_type	distance

Example:

```
contacts = pipeline.add(ProteinLigandContacts(
    structures=boltz,
    ligand="LIG",
    contact_distance=4.0
))
```

Data Management

Filter

Environment: `ProteinEnv`

Parameters: - `data`: Union[ToolOutput, StandardizedOutput] (required) - Datasheet input to filter - `pool`: Union[ToolOutput, StandardizedOutput] = None - Structure/sequence pool for copying filtered items - `expression`: str (required) - Pandas query-style filter expression (e.g., "distance < 3.5 and confidence > 0.8") - `max_items`: Optional[int] = None - Maximum items to keep after filtering - `sort_by`: Optional[str] = None - Column name to sort by before applying max_items - `sort_ascending`: bool = True - Sort order (True = ascending, False = descending)

Outputs: - Filtered pool with same structure as input - `datasheets.missing`:

id	structure	msa

Example:

```
filtered = pipeline.add(Filter(
    data=distances.datasheets.analysis,
    pool=boltz,
    expression="distance < 3.5 and confidence_score > 0.85",
    max_items=10,
    sort_by="distance"
))
```

SelectBest

Environment: `ProteinEnv`

Parameters: - `pool`: Union[ToolOutput, StandardizedOutput, List[Union[ToolOutput, StandardizedOutput]]] (required) - Single or list of tool outputs to select from - `datasheets`: Union[List[Union[ToolOutput, StandardizedOutput, DatasheetInfo, str]], List[str]] (required) - Datasheets to evaluate for selection - `metric`: str (required) - Primary metric to optimize - `mode`: str = "max" - Optimization direction ("max" or "min") - `weights`: Optional[Dict[str, float]] = None - Dictionary of {metric_name: weight} for multi-metric selection - `tie_breaker`: str = "first" - How to break ties ("first", "random", or metric name) - `composite_function`: str = "weighted_sum" - How to combine metrics (weighted_sum, product, min, max) - `name`: str = "best" - Name for output structure file

Outputs: - Single best structure/sequence with same format as input pool

Example:

```
best = pipeline.add(SelectBest(
    pool=boltz,
    datasheets=[distances.datasheets.analysis],
    metric="distance",
    mode="min"
))

# Multi-objective selection
best_multi = pipeline.add(SelectBest(
    pool=boltz,
    datasheets=[analysis.datasheets.merged],
    metric="composite_score",
    weights={"binding_affinity": 0.6, "pLDDT": 0.4},
    mode="max"
))
```

RemoveDuplicates

Environment: `ProteinEnv`

Parameters: - `pool`: Union[ToolOutput, StandardizedOutput] (required) - Items to deduplicate - `history`: Optional[Union[ToolOutput, StandardizedOutput, List]] = None - Previous datasheets for cross-cycle deduplication - `compare`: str = "sequence" - Comparison method (sequence, structure, id) - `similarity_threshold`: float = 1.0 - Similarity threshold for structure comparison (1.0 = exact match)

Outputs: - Deduplicated pool with same structure as input - `datasheets.removed`:

id	reason

Example:

```
unique = pipeline.add(RemoveDuplicates(
    pool=lmpnn,
    compare="sequence"
))
```

MergeDatasheets

Environment: `ProteinEnv`

Parameters: - `datasheets`: List[Union[ToolOutput, StandardizedOutput, DatasheetInfo, str]] (required) - List of datasheets to merge - `key`: str = "id" - Join column name - `prefixes`: Optional[List[str]] = None - Prefixes for columns from each datasheet - `suffixes`: Optional[List[str]] = None - Suffixes for columns from each datasheet - `how`: str = "inner" - Join type (inner, outer, left, right) - `calculate`: Optional[Dict[str, str]] = None - Derived column expressions {new_col: expression}

Outputs: - `datasheets.merged`: Combined datasheet with columns from all inputs

Example:

```
merged = pipeline.add(MergeDatasheets(
    datasheets=[distances.datasheets.analysis, plip.datasheets.interactions],
    prefixes=["dist_", "plip_"],
    key="id",
    calculate={"score": "dist_distance + plip_energy"}
))
```

ConcatenateDatasheets

Environment: `ProteinEnv`

Parameters: - `datasheets`: List[Union[ToolOutput, StandardizedOutput, DatasheetInfo, str]] (required) - List of datasheets to concatenate - `fill`: str = "N/A" - Value for missing columns - `ignore_index`: bool = True - Reset index in concatenated output

Outputs: - `datasheets.concatenated`: Row-wise concatenation of all input datasheets

Example:

```
concat = pipeline.add(ConcatenateDatasheets(
    datasheets=[cycle1_results, cycle2_results, cycle3_results],
    fill="N/A"
))
```

SliceDatasheet

Environment: `ProteinEnv`

Parameters: - `datasheet`: Union[ToolOutput, StandardizedOutput, DatasheetInfo, str] (required) - Input datasheet to slice - `start`: int = 0 - Starting row index - `end`: Optional[int] = None - Ending row index (None = to end) - `step`: int = 1 - Step size for slicing - `columns`: Optional[List[str]] = None - Specific columns to keep (None = all columns)

Outputs: - `datasheets.sliced`: Sliced datasheet

Example:

```
sliced = pipeline.add(SliceDatasheet(
    datasheet=results.datasheets.analysis,
    start=0,
    end=100,
    columns=["id", "distance", "confidence"]
))
```

ExtractMetrics

Environment: `ProteinEnv`

Parameters: - `datasheets`: List[Union[ToolOutput, StandardizedOutput, DatasheetInfo, str]] (required) - Input datasheets - `metrics`: List[str] (required) - Metric column names to extract - `group_by`: Optional[str] = None - Column to group by for aggregation - `aggregation`: str = "mean" - Aggregation function (mean, median, min, max, sum, std) - `pivot`: bool = False - Pivot metrics to columns

Outputs: - `datasheets.extracted`: Extracted metrics datasheet

Example:

```
metrics = pipeline.add(ExtractMetrics(
    datasheets=[boltz.datasheets.confidence],
    metrics=["complex_plddt", "ptm"],
    group_by="input_file",
    aggregation="mean"
))
```

AverageByDatasheet

Environment: `ProteinEnv`

Parameters: - `datasheets`: List[Union[ToolOutput, StandardizedOutput, DatasheetInfo, str]] (required) - Input datasheets - `group_by`: str (required) - Column to group by - `metrics`: List[str] (required) - Metric columns to average - `weights`: Optional[Dict[str, float]] = None - Weights for each metric

Outputs: - `datasheets.averaged`: Averaged metrics by group

Example:

```
averaged = pipeline.add(AverageByDatasheet(
    datasheets=[cycle1.datasheets.analysis, cycle2.datasheets.analysis],
    group_by="structure_id",
    metrics=["distance", "confidence"]
))
```

Utilities

LoadOutput

Environment: `ProteinEnv`

Parameters: - `output_json`: str (required) - Path to tool output JSON file (in ToolOutputs folder) - `filter`: Optional[str] = None - Pandas query-style filter expression - `validate_files`: bool = True - Check file existence when loading

Outputs: - Same structure as original tool that created the output

Example:

```
previous_boltz = pipeline.add(LoadOutput(
    output_json="/path/to/job/ToolOutputs/003_Boltz2_output.json",
    filter="confidence_score > 0.8"
))
```

MMseqs2

Environment: `ProteinEnv`

Parameters: - `sequences`: Union[str, List[str], ToolOutput, StandardizedOutput] (required) - Input sequences - `output_format`: str = "csv" - Output format (csv, a3m) - `timeout`: int = 3600 - Timeout in seconds for server response

Outputs: - `datasheets.msas`:

id	sequence_id	sequence	msa_file

Example:

```
msas = pipeline.add(MMseqs2(
    sequences=lmpnn,
    timeout=7200
))
```

MMseqs2Server

Environment: None (doesn't require conda)

Parameters: - `port`: int = 8000 - Server port - `host`: str = "0.0.0.0" - Server host - `workers`: int = 4 - Number of worker processes

Note: This server tool must be run separately to provide MMseqs2 as a service. However, the MMseqs2 client automatically runs it when needed, so manual server setup is typically not required.

CompoundLibrary

Environment: `ProteinEnv`

Parameters: - `library`: Union[str, Dict[str, Union[str, List[str]]]] (required) - Dictionary with expansion keys or path to CSV file - `primary_key`: Optional[str] = None - Root key for expansion when library is dictionary - `covalent`: bool = False - Generate CCD/PKL files for covalent ligand binding - `validate_smiles`:

bool = True - Validate SMILES strings during expansion - `conformer_method`: str = "UFF" - Method for conformer generation (UFF, OpenFF, DFT)

Outputs: - `compounds`: CSV file with compound library - `datasheets.compounds`:

id	format	smiles	ccd	{branching_keys}

Examples:

```
# With primary_key for combinatorial expansion
library = pipeline.add(CompoundLibrary(
    library={
        "scaffold": "<linker><fluorophore>",
        "linker": ["CCOCC", "CCOCCOCC"],
        "fluorophore": ["c1ccc(N)cc1"]
    },
    primary_key="scaffold"
))

# Without primary_key - direct SMILES list
library = pipeline.add(CompoundLibrary(
    library={
        "compounds": ["CCO", "CCCO", "CCCCO", "c1ccccc1"]
    }
))
```

FetchStructure

Environment: `ProteinEnv`

Parameters: - `structure_ids`: List[str] (required) - List of structure IDs to fetch - `database`: str = "pdb" - Database source (pdb, alphafold) - `format`: str = "pdb" - File format (pdb, cif, mmcif)

Outputs: - `structures`: Downloaded structure files - `datasheets.structures`:

id	structure_file	source_database

Example:

```
fetched = pipeline.add(FetchStructure(
    structure_ids=["1ABC", "2XYZ"],
    database="pdb"
))
```

PyMOL

Environment: `ProteinEnv`

Note: This tool is not fully debugged yet and may require adjustments.

Parameters: - `structures`: Union[str, List[str], ToolOutput, StandardizedOutput] (required) - Structures to visualize - `reference_structure`: Optional[Union[str, ToolOutput]] = None - Reference for alignment

- `color_by`: str = “chain” - Coloring scheme (chain, b_factor, element, ss, spectrum) - `alignment`: str = “align” - Alignment method (align, super, cealign) - `ray_trace`: bool = False - Generate ray-traced images - `image_size`: Tuple[int, int] = (1920, 1080) - Image dimensions if ray tracing

Outputs: - PyMOL session files (.pse) - Images (if ray_trace=True)

Example:

```
pymol = pipeline.add(PyMOL(
    structures=best,
    reference_structure=template,
    color_by="b_factor",
    alignment="super"
))
```
