Contents

BioPipelines User Manual

Table of Contents

Architecture Overview

BioPipelines is a system that **generates bash scripts** for bioinformatics workflows. The pipeline itself does not execute computations directly - instead, it predicts and prepares the filesystem structure that will exist at SLURM runtime, then generates bash scripts to be executed when the SLURM job runs.

Execution Model

- **Pipeline Role**: Generates bash scripts and predicts filesystem structure
- **Execution Time**: Scripts are executed only at SLURM runtime
- **HelpScripts**: Utility scripts designed to be executed during SLURM job execution

- **Filesystem Prediction**: Pipeline pre-determines output paths and directory structure before job submission

Key Directories

- `PipelineScripts/`: Tool classes that generate pipeline configurations
- `HelpScripts/`: Runtime utility scripts executed during SLURM jobs
- **Output Structure**: Each tool creates numbered directories (`1_RFdiffusion/`, `2_ProteinMPNN/`, etc.)

Core Concepts

Datasheets

All tools output standardized CSV datasheets that serve as the primary interface between pipeline steps:

- **Structure Generation**: `id, structure_file, [tool-specific columns]`
- **Sequence Design**: `id, sequence, source_id, [metrics]`
- **Compound Generation**: `id, format, smiles, ccd`
- **Analysis Results**: `id, source_structure, [metric_columns]`

Tool Chaining

Tools are connected through their standardized outputs:

```
rfd = RFdiffusion(...)
pmpnn = ProteinMPNN(input=rfd.output)
alphafold = AlphaFold(input=pmpnn.output)
```

Environment Management

Each tool specifies its conda environment requirements for SLURM execution.

Tool Abstract

Tool Structure

Every BioPipelines tool follows this abstract pattern:

```
class ToolName(BaseConfig):
    TOOL_NAME = "ToolName"

    def __init__(self, input=None, **parameters):
        # Tool-specific initialization

    def get_output_files(self) -> Dict[str, List[str]]:
        # Returns predicted output file paths
        return {
            "structures": [...],      # PDB/CIF files
            "sequences": [...],       # FASTA files
            "compounds": [...],       # Compound files
            "datasheets": {...},      # CSV datasheets
            "output_folder": self.output_folder
        }

    @property
    def output(self) -> StandardizedOutput:
        # Standardized output interface
```

Input/Output Pattern

- **Input**: `input=previous_tool.output` or direct parameters
- **Output**: Standardized datasheets + files accessible via `tool.output`
- **Chaining**: Tools consume the `output` property of upstream tools

Datasheet Standardization

- **ID Column**: Always present, unique identifier for each item
- **Source Tracking**: `source_id`, `source_structure` columns link to upstream tools
- **File References**: Absolute paths to generated files
- **Metrics**: Tool-specific analysis columns

Structure Generation Tools

RFdiffusion

Purpose: Generate novel protein structures using diffusion models

Usage

```
rfd = RFdiffusion(
    name="my_designs",
    pdb="path/to/template.pdb",            # Optional template
    contigs="A1-100/50-80",  # Chain definitions
    length="200-300",             # Target length range
    num_designs=10,               # Number of structures
    environment="rfdiffusion"
)
```

Parameters

- `name`: Job identifier
- `pdb`: Template PDB file (optional)
- `contigs`: Chain length specifications
- `length`: Total length or range
- `num_designs`: Number of structures to generate
- `environment`: Conda environment name

Input

- Optional template PDB file
- Contig and length specifications

Output **Datasheet**: `rfdiffusion_results.csv` | Column | Description | |———|—————-| | `id` | Structure identifier | | `structure_file` | Path to generated PDB | | `fixed` | Fixed regions (PyMOL selection) | | `designed` | Designed regions (PyMOL selection) | | `exists` | Boolean, file existence check |

Files: PDB structures in output directory

AlphaFold

Purpose: Predict protein structures from sequences using AlphaFold

Usage

```
af = AlphaFold(
    input=pmpnn.output,           # From ProteinMPNN
    name="predictions",
    rank=True,                    # Rank by confidence
    num_relax=3,                  # Number to relax
    environment="alphafold"
)
```

Parameters

- `input`: Sequences from upstream tool
- `name`: Job identifier
- `num_relax`: Number of best models to relax
- `num_recycle`: Recycling iterations (default 3)
- `environment`: Conda environment

Input

- Sequence datasheet from ProteinMPNN or similar
- FASTA files with protein sequences

Output **Datasheet**: Inherits from input sequences | Column | Description | |———|—————-| | `id` | Sequence identifier | | `source_id` | Original sequence ID | | `sequence` | Protein sequence | | `structure_file` | Path to predicted PDB |

Files: PDB structures, confidence scores, MSAs

Boltz2

Purpose: Predict protein-ligand complex structures

Usage

```
boltz = Boltz2(
    proteins=pmpnn.output,          # Protein sequences
    ligands=compound_lib.output,    # Compound library
    name="complexes",
    num_samples=5,                  # Sampling iterations
    environment="boltz2"
)
```

Parameters

- `proteins`: Protein sequences (FASTA or upstream tool)
- `ligands`: Compound library or SMILES
- `config`: Configuration files
- `msas`: Multiple sequence alignments
- `name`: Job identifier
- `num_samples`: Number of samples per prediction
- `environment`: Conda environment

Input

- Protein sequences (FASTA files)
- Ligand library (compound datasheets)
- Optional MSAs and configuration files

Output **Multiple Datasheets**:

1. **Confidence**: `boltz_confidence_scores.csv` | Column | Description | |———|—————-| | `id` | Complex identifier | | `input_file` | Input configuration name | | `confidence_score` | Overall confidence | | `ptm` | Predicted Template Modeling score | | `iptm` | Interface PTM score |
2. **Sequences**: `boltz_sequences.csv` | Column | Description | |———|—————-| | `id` | Sequence identifier | | `sequence` | Protein sequence |
3. **Compounds**: `boltz_compounds.csv` | Column | Description | |———|—————-| | `id` | Compound identifier | | `smiles` | SMILES string | | `format` | Format type (SMILES/CCD) | | `ccd` | CCD identifier if applicable |

Files: CIF structure files, JSON confidence scores

RFdiffusionAllAtom

Purpose: Generate ligand-aware protein structures with all-atom diffusion

Usage

```
rfd_aa = RFdiffusionAllAtom(
    ligand="ZIT",                       # Ligand identifier
    pdb="template.pdb",               # Template structure
    contigs="A1-100,10-20",           # Contig specification
    num_designs=5,                    # Number of designs
    active_site=True,                 # Use active site model
    environment="rfdiffusion"
)
```

Parameters

- `ligand`: Ligand identifier (e.g., ‘ZIT’, ‘RFP’)
- `pdb`: Input PDB template (optional)
- `contigs`: Contig specification for design regions
- `inpaint`: Inpainting regions specification
- `num_designs`: Number of structures to generate
- `active_site`: Use active site model for small motifs
- `steps`: Diffusion steps (default 200)
- `ppi_design`: Enable protein-protein interaction design
- `ppi_hotspot_residues`: List of hotspot residues for PPI
- `environment`: Conda environment

Input

- Template PDB file with ligand context
- Ligand specifications and design regions
- Optional hotspot residue definitions

Output Datasheet: `rfdiffusion_allatom_results.csv`

Column	Description
`id`	Structure identifier
`source_id`	Template source ID
`structure_file`	Path to generated PDB
`fixed`	Fixed regions (PyMOL selection)
`designed`	Designed regions (PyMOL selection)
`contigs`	Contig specification used
`time`	Generation time
`status`	Completion status

Files: PDB structures with ligand contexts

Sequence Design Tools

ProteinMPNN

Purpose: Design protein sequences for given structures

Usage

```
pmpnn = ProteinMPNN(
    input=rfd.output,             # Structure input
    name="sequences",
    num_sequences=50,             # Sequences per structure
    temperature=0.1,              # Sampling temperature
    environment="proteinmpnn"
)
```

Parameters

- `input`: Structures from upstream tool
- `name`: Job identifier
- `num_sequences`: Number of sequences per structure
- `temperature`: Sampling temperature
- `batch_size`: Processing batch size
- `environment`: Conda environment

Input

- Structure datasheet from RFdiffusion or similar
- PDB files with protein structures

Output

Datasheet: `pmpnn_results.csv` | Column | Description | |———|—————-| | `id` | Sequence identifier | | `source_id` | Source structure ID | | `source_structure` | Path to input PDB | | `sequence` | Designed protein sequence | | `score` | ProteinMPNN score | | `seq_recovery` | Sequence recovery rate |

Files: FASTA files with designed sequences

LigandMPNN

Purpose: Design protein sequences considering ligand interactions

Usage

```
lmpnn = LigandMPNN(
    input=rfd.output,              # Structure input
    ligand_params="ligand.json",   # Ligand parameters
    name="ligand_designs",
    num_sequences=30,
    environment="ligandmpnn"
)
```

Parameters

- `input`: Structures with ligands
- `ligand_params`: Ligand parameter file
- `name`: Job identifier
- `num_sequences`: Sequences per structure
- `temperature`: Sampling temperature
- `environment`: Conda environment
- `fixed`: Selection of positions to keep fixed
- `designed`: Selection of positions to redesign

Input

- Protein structures with bound ligands

- Ligand parameter configuration

Output Datasheet: `lmpnn_results.csv` | Column | Description | |———|————-| | `id` | Sequence identifier | | `source_id` | Source structure ID | | `sequence` | Designed sequence | | `ligand_binding_score` | Binding affinity score |

Files: FASTA files with ligand-aware sequences

Compound Generation Tools

CompoundLibrary

Purpose: Generate and process compound libraries for structure prediction

Usage

```
compounds = CompoundLibrary(
    library={"core": "C1CCCCC1", "R1": ["F", "Cl", "Br"]},
    name="my_library",
    primary_key="core",
    max_compounds=1000,
    environment="rdkit"
)
```

Parameters

- `library`: Dictionary or file paths with compound definitions
- `name`: Library identifier
- `primary_key`: Primary library component
- `max_compounds`: Maximum number to generate
- `covalent`: Generate covalent variants
- `environment`: Conda environment

Input

- **Dictionary**: `{"scaffold": "SMILES", "R1": ["substitutions"]}`
- **File Paths**: CSV/TXT files with SMILES
- **SMILES String**: Single compound

Output Datasheet: `{name}_compounds.csv` | Column | Description | |———|————-| | `id` | Compound identifier | | `smiles` | SMILES string | | `format` | Format type (SMILES) | | `ccd` | CCD code (if applicable) |

Optional Datasheet: `{name}_compound_properties.csv` | Column | Description | |———|————-| | `id` | Compound identifier | | `MW` | Molecular weight | | `LogP` | Lipophilicity | | `HBD` | Hydrogen bond donors | | `HBA` | Hydrogen bond acceptors |

Files: JSON library definitions, property calculations

Analysis Tools

ResidueAtomDistance

Purpose: Calculate distances between residue selections and atom coordinates

Usage

```
distance = ResidueAtomDistance(
    input=boltz.output,           # Structures to analyze
    residue="D in IGDWK",
    atom_coords="LIG.Cl",
    name="active_site_distance"   # Metric name
)
```

Parameters

- `input`: Structures from upstream tool
- `residue_selection`: PyMOL-style residue selection
- `atom_coords`: Target atom coordinates [x, y, z]
- `name`: Analysis identifier
- `distance_cutoff`: Maximum distance threshold

Input

- Structure datasheet with PDB files
- Selection criteria and target coordinates

Output **Datasheet**: `{name}_distances.csv` | Column | Description | |——–|—————-| | `id` | Structure identifier | | `source_structure` | Path to analyzed PDB | | `{metric_name}` | Distance measurement |

Confidence

Purpose: Extract confidence scores (pLDDT) from protein structures

Usage

```
confidence = Confidence(
    input=alphafold.output,       # Structures to analyze
    name="plddt"            # Custom metric name
)
```

Parameters

- `input`: Structure datasheet from prediction tools
- `name`: Analysis identifier
- `metric_name`: Custom name for confidence metric column
- `residue_range`: Specific residues to analyze (optional)

Input

- Structure datasheet with PDB files
- Protein structures with B-factor confidence scores

Output **Datasheet**: `{name}_confidence.csv` | Column | Description | |———|—————-| | `id` | Structure identifier | | `source_structure` | Path to analyzed PDB | | `<name>` | pLDDT value |

Confidence Score Interpretation

- **Very high confidence**: >90 (highly accurate)
- **High confidence**: 70-90 (generally accurate)

- **Low confidence**: 50-70 (potentially inaccurate)
- **Very low confidence**: <50 (likely inaccurate)

Filtering Tools

Filter

Purpose: Filter datasheets using pandas query expressions

Usage

```
filter_tool = Filter(
    input=analysis.output,        # Datasheet to filter
    expression="plddt_avg > 70 & contact_count > 50",
    name="high_quality",
    max_items=100                 # Limit results
)
```

Parameters

- `input`: Datasheet from analysis tools
- `expression`: Pandas query expression
- `name`: Filter identifier
- `combination`: "AND" or "OR" for multiple criteria
- `max_items`: Maximum items to keep
- `score_weights`: Weighting for scoring criteria

Input

- Analysis datasheet with metrics
- Filter expression criteria

Output **Datasheet**: Filtered version of input - Same columns as input - Only rows meeting filter criteria - Additional metadata about filtering

Expression Examples

```
# Simple filters
"confidence > 0.8"
"distance < 5.0"

# Complex filters
"(plddt_avg > 70) & (contact_count > 50)"
"score > 0.9 | (confidence > 0.8 & distance < 3.0)"

# String matching
"id.str.contains('design')"
```

Optimization Tools

SelectBest

Purpose: Select the single best item from analysis results for iterative optimization

Usage

```
best = SelectBest(
    input=analysis.output,          # Analysis results
    metric="binding_affinity",      # Primary optimization metric
    mode="max",                     # Maximize or minimize
    name="best_design"
)
```

Parameters

- `input`: Analysis datasheet with metrics
- `metric`: Primary metric to optimize
- `mode`: "max" or "min" to maximize or minimize metric
- `weights`: Dict of {metric_name: weight} for multi-objective selection
- `tie_breaker`: How to handle ties ("first", "random", or metric name)
- `composite_function`: How to combine metrics ("weighted_sum", "product", "min", "max")
- `name`: Name for the selected best item

Input

- Analysis datasheet with quantitative metrics
- Selection criteria and optimization parameters

Output **Datasheet**: Single row with best selected item - Same columns as input datasheet - Only the single best item based on criteria - Additional metadata about selection process

Multi-Objective Selection

```python
# Weighted combination of multiple metrics
best_multi = SelectBest(
    input=combined_analysis,
    metric="composite_score",
    weights={"binding_affinity": 0.6, "plddt_avg": 0.4},
    mode="max",
    tie_breaker="binding_affinity"
)
```

Use Cases

- **Iterative Design**: Select best design for next optimization cycle
- **Multi-Objective**: Balance multiple competing objectives

- **Quality Control**: Pick highest quality result from batch
- **Resource Optimization**: Select most promising candidate for expensive analysis

---

Utility Tools

MergeDatasheets

Purpose: Combine multiple datasheets with intelligent column handling

Usage

```python
merger = MergeDatasheets(
    input_datasheets=[analysis1.output, analysis2.output],
    join_key="id",
    name="combined_analysis"
)
```

Parameters

- `input_datasheets`: List of datasheet sources
- `join_key`: Column to merge on (default: "id")
- `name`: Merger identifier
- `handle_duplicates`: Strategy for duplicate columns

Input

- Multiple datasheets with compatible ID columns
- Join specifications

Output **Datasheet**: Combined datasheet - All unique columns from input datasheets - Outer join preserving all data - Collision handling for duplicate column names

---

LoadOutput

Purpose: Load results from completed pipeline runs

Usage

```
loaded = LoadOutput(
    result_file="output.json",     # Pipeline result metadata
    validate_files=True
)
```

Parameters

- `result_file`: Pipeline output JSON file
- `validate_files`: Check file existence
- `load_datasheets`: Load datasheet contents

Input

- Pipeline result JSON metadata
- Output directory structure

Output

- Access to all pipeline outputs
- Datasheet loading and validation
- File existence verification

Pipeline Examples

Basic Protein Design Pipeline

```
from PipelineScripts import *

# Generate novel structures
rfd = RFdiffusion(
    name="novel_designs",
    contigs=["A1-150"],
    num_designs=20,
    environment="rfdiffusion"
)

# Design sequences
pmpnn = ProteinMPNN(
    input=rfd.output,
    name="sequences",
    num_sequences=10,
    temperature=0.1,
    environment="proteinmpnn"
```

```python
)

# Predict structures
alphafold = AlphaFold(
    input=pmpnn.output,
    name="predictions",
    rank=True,
    num_relax=5,
    environment="alphafold"
)

# Analyze quality
analysis = Confidence(
    input=alphafold.output,
    name="quality"
)

# Filter best results
filter_best = Filter(
    input=analysis.output,
    expression="plddt_avg > 80 & contact_count > 100",
    name="high_quality",
    max_items=5
)

# Create pipeline
pipeline = Pipeline([rfd, pmpnn, alphafold, analysis, filter_best])
pipeline.save()
```

Protein-Ligand Complex Design

```python
# Generate compound library
compounds = CompoundLibrary(
    library={"scaffold": "c1ccccc1", "R1": ["F", "Cl", "N"]},
    name="drug_library",
    max_compounds=50,
    environment="rdkit"
)

# Design protein structures
rfd = RFdiffusion(
    name="binding_sites",
    length="100-200",
    num_designs=10,
    environment="rfdiffusion"
)

# Predict complexes
```

```
boltz = Boltz2(
    proteins=rfd.output,
    ligands=compounds.output,
    name="complexes",
    num_samples=3,
    environment="boltz2"
)

# Analyze binding
distance_analysis = ResidueAtomDistance(
    input=boltz.output,
    residue_selection="resi 50-80",
    atom_coords=[0.0, 0.0, 0.0],
    name="binding_distance"
)

# Filter viable complexes
viable = Filter(
    input=distance_analysis.output,
    expression="binding_distance < 4.0 & confidence_score > 0.7",
    name="viable_complexes"
)

pipeline = Pipeline([compounds, rfd, boltz, distance_analysis, viable])
```

Iterative Design Optimization

```
# Initial design
rfd = RFdiffusion(
    name="initial_designs",
    contigs=["A1-100"],
    num_designs=50,
    environment="rfdiffusion"
)

# Analyze quality
confidence = Confidence(
    input=rfd.output,
    name="initial_confidence"
)

# Select best for refinement
best_initial = SelectBest(
    input=confidence.output,
    metric="plddt_avg",
    mode="max",
    name="best_initial"
)
```

```
# Design sequences for best structure
pmpnn = ProteinMPNN(
    input=best_initial.output,
    name="refined_sequences",
    num_sequences=20,
    environment="proteinmpnn"
)

# Predict refined structures
alphafold = AlphaFold(
    input=pmpnn.output,
    name="refined_structures",
    environment="alphafold"
)

# Multi-objective analysis
final_confidence = Confidence(
    input=alphafold.output,
    name="final_confidence"
)

binding_analysis = ResidueAtomDistance(
    input=alphafold.output,
    residue_selection="resi 50-70",
    atom_coords=[0.0, 0.0, 0.0],
    name="binding_site"
)

# Combine metrics
combined = MergeDatasheets(
    input_datasheets=[final_confidence.output, binding_analysis.output],
    name="combined_metrics"
)

# Select optimal design
final_best = SelectBest(
    input=combined.output,
    metric="composite_score",
    weights={"plddt_avg": 0.7, "binding_site_distance": 0.3},
    mode="max",
    name="optimal_design"
)

pipeline = Pipeline([rfd, confidence, best_initial, pmpnn, alphafold,
                    final_confidence, binding_analysis, combined, final_best])
```

Multi-Analysis Filtering Pipeline

```python
# Load existing structures
structures = LoadOutput(result_file="previous_run.json")

# Multiple analyses
confidence_analysis = Confidence(
    input=structures.output,
    name="confidence"
)

distance_analysis = ResidueAtomDistance(
    input=structures.output,
    residue_selection="resi 100-120",
    atom_coords=[10.0, 15.0, 20.0],
    name="active_site"
)

# Combine analyses
combined = MergeDatasheets(
    input_datasheets=[confidence_analysis.output, distance_analysis.output],
    name="all_metrics"
)

# Complex filtering
final_filter = Filter(
    input=combined.output,
    expression="(plddt_avg > 75) & (active_site_distance < 5.0) & (helix_content > 0.3)",
    name="final_selection",
    max_items=10
)
```

Troubleshooting

Common Issues

1. File Path Errors **Problem**: Tools cannot find input files **Solution**: Ensure upstream tools completed successfully and files exist

2. Environment Conflicts **Problem**: Tool fails due to conda environment issues **Solution**: Verify environment names match available conda environments

3. Memory Issues with Large Libraries **Problem**: Compound libraries too large for processing **Solution**: Use `max_compounds` parameter to limit library size

4. Filter Expression Errors **Problem**: Filter expressions fail with syntax errors **Solution**: Use pandas query syntax, quote string columns

5. Missing Dependencies **Problem**: Tools fail due to missing software dependencies **Solution**: Check that all required tools are installed in specified environments

Debugging Tips

1. **Check Log Files**: Each tool generates logs in the main pipeline directory
2. **Validate Datasheets**: Inspect CSV outputs to understand data flow
3. **Use Small Test Cases**: Start with minimal examples before scaling up
4. **File Existence**: Verify all input files exist and are accessible
5. **Environment Testing**: Test tools individually in their target environments

Best Practices

1. **Incremental Development**: Build pipelines step by step
2. **Data Validation**: Check intermediate outputs before chaining tools
3. **Resource Planning**: Consider computational requirements for each tool
4. **Error Handling**: Plan for failed jobs and restart strategies
5. **Documentation**: Document custom parameters and pipeline logic