

BioPipelines Developer Manual

BioPipelines Developer Manual

Index

Development Environment

- [Setup](#)
- [Working with VS Code](#)
- [Working with Git](#)
- [Working with Claude Code](#)

Tool Development

- [Architecture Overview](#)
- [Base Classes](#)
- [Creating New Tools](#)
- [Input/Output Patterns](#)
- [Script Generation](#)
- [Testing Tools](#)

Best Practices

- [Code Principles](#)
- [Error Handling](#)
- [Documentation](#)

Examples

- [Simple Tool Example](#)
 - [Complex Tool Example](#)
-

Setup

Prerequisites

BioPipelines requires: - Python 3.8+ - Access to UZH cluster (for SLURM execution) - Conda/Mamba for environment management - Git - VS Code (recommended)

Project Structure

```
biopipelines/  
  PipelineScripts/          # Tool implementations  
    base_config.py          # Base class for all tools  
    folders.py              # Folder management class (will be changed to a yaml config in the future,  
    pipeline.py             # Pipeline orchestration
```

```

    <tool_name>.py          # Individual tool classes
    ...
HelpScripts/              # Runtime helper scripts
    pipe_*.py              # Scripts executed during SLURM runtime
    ...
Docs/                     # Documentation
    UserManual.md
    DeveloperManual.md
CLAUDE.md                 # Claude Code instructions
README.md                 # Basic documentation

```

Working with VS Code

Recommended Extensions

- **Python** (Microsoft) - Python language support
- **Pylance** - Fast Python language server
- **GitLens** - Git integration
- **Markdown All in One** - Markdown editing
- **Claude for VS Code** - AI assistance (if available)

Workspace Setup

1. Clone the repository locally
 2. Open biopipelines root in VS Code
 3. Set Python interpreter to your conda environment
 4. Use integrated terminal for testing (set debug=True during pipeline instantiation)
-

Working with Git

Daily Workflow

```

# Start work - check status
git status

# Pull latest changes
git pull origin main

# Create feature branch (optional but recommended)
git checkout -b feature/my-new-tool

# Make changes, then commit
git add PipelineScripts/my_tool.py
git commit -m "Add MyTool for specific analysis"

# Push to remote
git push origin feature/my-new-tool

```

Managing Example Pipelines

Example pipelines (in your workspace, not in the repo) often get modified during testing. Here's how to handle them:

What to Keep: - Modified pipelines in your workspace (don't commit test pipelines) - Tool implementations in PipelineScripts/ - Helper scripts in HelpScripts/ - Documentation updates in Docs/

What to Ignore: - Generated SLURM scripts (RunTime/, *.sh) - Job output folders (JobName_*/) - Test pipelines with personal paths - __pycache__/, *.pyc

Example .gitignore:

```
# Python
__pycache__/
*.pyc
*.pyo

# Pipeline outputs
*_ [0-9]*/
RunTime/
ToolOutputs/

# Personal test pipelines
examples/my_test_*.py
test_*.py
```

Resolving Merge Conflicts

Conflicts commonly occur in: - Tool files when multiple people update same tool - Documentation when multiple features added - folders.py when default paths change

Common Conflict Resolution:

1. Tool Files (PipelineScripts/*.py):

```
# Check what changed
git diff PipelineScripts/boltz2.py

# If conflict in tool file:
# 1. Open file in VS Code
# 2. Review <<<<<< HEAD and >>>>>> branch markers
# 3. Keep the version with latest features
# 4. Test the tool works:
python -c "from PipelineScripts.boltz2 import Boltz2; print('OK')"

# Stage resolved file
git add PipelineScripts/boltz2.py
```

2. Documentation (Docs/*.md):

```
# Usually both changes are valid - merge manually
# Open in VS Code, keep both additions
git add Docs/UserManual.md
```

3. folders.py:

```
# This file rarely conflicts
# If it does, prefer main branch version
git checkout --theirs PipelineScripts/folders.py
git add PipelineScripts/folders.py
```

Complete the merge:

```
git commit -m "Merge feature/my-tool - resolved conflicts in boltz2.py"
git push
```

Stashing Changes

When you need to switch branches with uncommitted changes:

```
# Save current work
git stash save "WIP: working on MyTool validation"

# Switch branches
git checkout main
git pull

# Return to your work
git checkout feature/my-tool
git stash pop
```

Checking File History

```
# See who changed what
git log --oneline PipelineScripts/boltz2.py

# See specific changes
git show <commit-hash>

# Compare with previous version
git diff HEAD~1 PipelineScripts/boltz2.py
```

Common Git Issues

Issue: “Your branch is behind”

```
# Pull and merge
git pull origin main

# Or pull with rebase (cleaner history)
git pull --rebase origin main
```

Issue: “Accidentally committed to main”

```
# Move commits to new branch
git branch feature/my-changes
git reset --hard origin/main
git checkout feature/my-changes
```

Issue: “Need to undo last commit”

```
# Keep changes, undo commit
git reset --soft HEAD~1

# Discard changes completely
git reset --hard HEAD~1
```

Working with Claude Code

Best Practices

1. **Use Plan Mode When Possible** Always start complex tasks in plan mode to review what will be done before execution:

/plan Create a new tool for binding energy calculation

This allows you to: - Review the approach before code is written - Catch potential issues early - Ensure consistency with existing tools - Verify file locations and patterns

2. Review Code Before Accepting Always read the code Claude generates, especially: - New tool implementations - Changes to existing tools - Updates to core classes - Script generation logic

Don't blindly accept changes - understand what's being modified.

3. Be Suspicious of Core Class Changes Red flags - be very careful if Claude suggests changes to:

Critical Files (rarely need changes): - `base_config.py` - Base class for all tools - `pipeline.py` - Pipeline orchestration - `folders.py` - Folder management - `standardized_output.py` - Output interface

What to do: 1. Question why core files need modification 2. Check if the issue can be solved in the tool class instead 3. Review the change very carefully 4. Test that existing tools still work 5. Consider if this affects all tools

Usually the answer is: The tool should adapt to the base class, not vice versa.

4. Ask Claude to Review Existing Tools For consistency, always reference existing implementations:

Good prompts:

"Before creating BindingAnalyzer, read ResidueAtomDistance and PLIP to understand the pattern for analyzing

"Check how Boltz2 handles multiple input types and use the same pattern in my new tool"

"Look at how LigandMPNN uses `_setup_file_paths()` and apply the same pattern"

This ensures: - Consistent code style - Proven patterns - No reinventing the wheel - Easier maintenance

Prompting Principles

Claude Code works best when you follow these principles from CLAUDE.md:

5. Never Implement Fallbacks or Guessed Values The code must crash explicitly rather than guess or use fallbacks.

Bad:

"Create a function to read config, with fallback to default values if not found"

Good:

"Create a function to read config. If the config file doesn't exist, raise `ValueError` with the exact path"

How to spot bad Claude-generated code:

```
# Bad - fallback value
def get_config(path):
    if not os.path.exists(path):
        return {"default": "value"} # Fallback!
    return read_config(path)

# Good - explicit failure
def get_config(path):
    if not os.path.exists(path):
        raise ValueError(f"Config file not found: {path}")
    return read_config(path)
```

6. Specify Values Once, Pass Appropriately Bad:

```
def generate_script(self):
    output_csv = os.path.join(self.output_folder, "results.csv")
    # ... 100 lines later ...
    summary_csv = os.path.join(self.output_folder, "results.csv") # Duplicated!
```

Good:

```
def _setup_file_paths(self):
    """Define all file paths once."""
    self.results_csv = os.path.join(self.output_folder, "results.csv")
    self.summary_csv = os.path.join(self.output_folder, "summary.csv")

def generate_script(self):
    # Use self.results_csv consistently
```

7. Be Explicit About Requirements Bad:

"Update the tool to handle MSAs"

Good:

"Update Boltz2 tool to:

1. Accept msas parameter of type Union[str, ToolOutput]
2. If ToolOutput, extract MSA files from datasheets.msas
3. Copy MSA files to self.msa_cache_folder
4. Pass MSA folder path to boltz predict command with --msa-cache flag
5. Do NOT create fallback MSA generation"

8. Reference Existing Patterns in Prompts Good:

"Create a new tool following the same pattern as ResidueAtomDistance:

- Accept structures parameter (not input)
- Use StandardizedOutput interface
- Generate analysis CSV with DatasheetInfo
- Use _setup_file_paths() pattern from ligand_mpn.py"

Example Prompts for Common Tasks

Adding a New Tool

Create a new tool called ProteinAnalyzer in PipelineScripts/protein_analyzer.py:

1. Inherit from BaseConfig
2. Accept structures parameter (Union[ToolOutput, StandardizedOutput])
3. Accept analysis_type parameter (str, must be one of: "surface", "volume", "charge")
4. DEFAULT_ENV = "ProteinEnv"
5. Generate analysis CSV with columns: id, source_structure, {analysis_type}_value
6. Follow the pattern from residue_atom_distance.py for:
 - Parameter validation
 - File path setup (_setup_file_paths method)
 - Output definition with DatasheetInfo
7. Do NOT implement fallbacks - crash if structures are invalid

Updating Documentation

Update Docs/UserManual.md to add ProteinAnalyzer tool:

1. Add under Analysis section after ResidueAtomDistance
2. Use the exact same format as other tools:
 - Parameters with proper types
 - Outputs with table format (not brackets)
 - Example code block
3. For datasheets, use this format:
 - `datasheets.analysis`:

```
| id | source_structure | {analysis_type}_value |
|----|-----|-----|
```

4. Do NOT add extra explanations - follow the concise style

Debugging Issues

The Boltz2 tool is failing with "MSA file not found" error.

Debug this by:

1. Read boltz2.py lines 460-520 (MSA handling section)
2. Check what path is being constructed for MSA files
3. Compare with the actual output from previous tool (show the get_output_files method)
4. Identify the mismatch
5. Fix by updating the path construction to match the actual output
6. Do NOT add fallback logic - the paths must match exactly

Verification After Changes

Always verify Claude's changes work correctly:

1. Quick Syntax Check

```
# Check tool imports correctly
python -c "from PipelineScripts.my_tool import MyTool; print('OK')"
```

```
# Check all imports still work
python -c "from PipelineScripts import *; print('All imports OK')"
```

2. Test in Debug Pipeline

```
from PipelineScripts.pipeline import Pipeline
from PipelineScripts.my_tool import MyTool

# Use debug=True to test without SLURM
pipeline = Pipeline("TestPipe", "TestJob", "Testing", debug=True)

result = pipeline.add(MyTool(
    structures="test.pdb",
    param="value"
))

# Check outputs are predicted correctly
print(result.get_output_files())
```

3. Check Core Files Unchanged

```
# If suspicious, diff against last commit
git diff base_config.py
git diff pipeline.py

# Should see minimal/no changes unless intentional
```

4. Verify Consistency

```
# Ask Claude to verify
claude "Check that MyTool follows the same patterns as ResidueAtomDistance for:
1. Parameter validation
2. File path setup
3. DatasheetInfo usage
4. Error messages"
```

Working with Claude in Terminal

When using `claude` command:

```
# Be specific about context
claude "Read PipelineScripts/boltz2.py and update the MSA handling to..."

# Reference specific patterns
claude "Update fuse.py to remove 'input' parameter following the same pattern used in residue_atom_distance.py"

# Request verification
claude "Check all tools in PipelineScripts/ and list which ones still use DEFAULT_RESOURCES"

# Ask for consistency check
claude "Compare my_tool.py with residue_atom_distance.py and list any pattern differences"
```

Architecture Overview

Core Principles

1. **Prediction-Based:** Tools predict filesystem structure before execution
2. **No File Copying:** Tools reference files in their original locations
3. **Tool Agnostic:** Each tool works independently via StandardizedOutput
4. **Explicit Failures:** Code must crash rather than guess or use fallbacks
5. **Single Definition:** Define paths/values once, pass them appropriately

Execution Flow

Pipeline Time (Python):

```
Tool.__init__() - Store parameters
Tool.validate_params() - Validate inputs
Tool.configure_inputs() - Set up file paths
Tool.generate_script() - Create bash script
Tool.get_output_files() - Predict outputs
```

SLURM Time (Bash):

```
Execute generated script
Check for predicted outputs
```

Mark as COMPLETED or FAILED

Tool Communication

Tools communicate via StandardizedOutput:

```
# Tool A predicts outputs
class ToolA(BaseConfig):
    def get_output_files(self):
        return {
            "structures": [self.output_folder + "/structure.pdb"],
            "datasheets": {"analysis": DatasheetInfo(...)},
            ...
        }

# Tool B uses those predictions
class ToolB(BaseConfig):
    def configure_inputs(self, pipeline_folders):
        if isinstance(self.structures, StandardizedOutput):
            # Access predicted structure paths
            structure_files = self.structures.structures
            # Access datasheet paths
            analysis_csv = self.structures.datasheets.analysis
```

Base Classes

BaseConfig

All tools inherit from BaseConfig which provides:

Class Attributes

```
class MyTool(BaseConfig):
    TOOL_NAME = "MyTool"           # Tool identifier
    DEFAULT_ENV = "ProteinEnv"     # Default conda environment
```

Instance Attributes (Automatic)

- self.tool_name - Tool identifier
- self.job_name - User-provided job name
- self.environment - Conda environment
- self.pipeline_ref - Reference to pipeline
- self.execution_order - Order in pipeline (1, 2, 3...)
- self.output_folder - Tool's output directory
- self.folders - Dict of all pipeline folders
- self.dependencies - List of dependent tool configs

Required Methods

```
@abstractmethod
def validate_params(self):
    """Validate tool-specific parameters.

    Must raise ValueError with specific error messages.
    Do NOT use fallback values.
```

```

    """
    pass

@abstractmethod
def configure_inputs(self, pipeline_folders: Dict[str, str]):
    """Configure input sources from pipeline context.

    Must set self.folders and configure all input paths.
    Must append to self.dependencies for tool inputs.
    Do NOT copy files - reference them in place.
    """
    pass

@abstractmethod
def generate_script(self, script_path: str) -> str:
    """Generate bash script for tool execution.

    Must return complete bash script as string.
    Use paths from self.folders for tool locations.
    """
    pass

@abstractmethod
def get_output_files(self) -> Dict[str, List[str]]:
    """Predict output files after execution.

    Must return dict with standardized keys:
    - structures: List[str]
    - structure_ids: List[str]
    - sequences: List[str]
    - sequence_ids: List[str]
    - datasheets: Dict[str, DatasheetInfo]
    - output_folder: str

    Do NOT check file existence - predict paths only.
    """
    pass

```

Optional Methods

```

def get_config_display(self) -> List[str]:
    """Return configuration info for display."""
    config_lines = super().get_config_display()
    config_lines.extend([
        f"PARAM: {self.param}",
        f"OUTPUT: {self.output_type}"
    ])
    return config_lines

def to_dict(self) -> Dict[str, Any]:
    """Serialize configuration for saving."""
    base_dict = super().to_dict()
    base_dict.update({
        "tool_params": {

```

```

        "param1": self.param1,
        "param2": self.param2
    }
})
return base_dict

```

StandardizedOutput

Provides unified interface to tool outputs:

```

# Access via dot notation
structures = tool_output.structures           # List[str]
structure_ids = tool_output.structure_ids      # List[str]
sequences = tool_output.sequences             # List[str]
datasheets = tool_output.datasheets           # DatasheetContainer

# Datasheet access
csv_path = tool_output.datasheets.analysis    # str (path)
info = tool_output.datasheets.info("analysis") # DatasheetInfo
column_ref = tool_output.datasheets.analysis.distance # (DatasheetInfo, str)

```

DatasheetInfo

Metadata for CSV/datasheet files:

```

DatasheetInfo(
    name="analysis",
    path="/path/to/analysis.csv",
    columns=["id", "distance", "energy"],
    description="Distance analysis results",
    count=100
)

```

Creating New Tools

Step 1: Create Tool File

Create PipelineScripts/my_tool.py:

```

"""
MyTool description - what it does and when to use it.
"""

import os
from typing import Dict, List, Any, Union

try:
    from .base_config import BaseConfig, ToolOutput, StandardizedOutput, DatasheetInfo
except ImportError:
    import sys
    sys.path.append(os.path.dirname(__file__))
    from base_config import BaseConfig, ToolOutput, StandardizedOutput, DatasheetInfo

class MyTool(BaseConfig):

```

```

"""
Tool for [specific purpose].

Takes [input type] and outputs [output type].

Commonly used for:
- Use case 1
- Use case 2
"""

# Tool identification
TOOL_NAME = "MyTool"
DEFAULT_ENV = "ProteinEnv"

def __init__(self,
              structures: Union[ToolOutput, StandardizedOutput],
              param1: str,
              param2: int = 10,
              **kwargs):
    """
    Initialize MyTool.

    Args:
        structures: Input structures from previous tool
        param1: Description of param1 (required)
        param2: Description of param2 (default: 10)
        **kwargs: Additional parameters
    """
    self.structures_input = structures
    self.param1 = param1
    self.param2 = param2

    # Initialize base class
    super().__init__(**kwargs)

    # Set up dependency
    if hasattr(structures, 'config'):
        self.dependencies.append(structures.config)

    # Initialize file paths
    self._initialize_file_paths()

def _initialize_file_paths(self):
    """Initialize file path placeholders."""
    self.results_csv = None
    self.output_pdb = None

def _setup_file_paths(self):
    """Set up all file paths after output_folder is known."""
    self.results_csv = os.path.join(self.output_folder, "results.csv")
    self.output_pdb = os.path.join(self.output_folder, "output.pdb")

def validate_params(self):
    """Validate MyTool parameters."""

```

```

    if not isinstance(self.structures_input, (ToolOutput, StandardizedOutput)):
        raise ValueError("structures must be ToolOutput or StandardizedOutput")

    if not self.param1:
        raise ValueError("param1 is required")

    if self.param2 <= 0:
        raise ValueError("param2 must be positive")

def configure_inputs(self, pipeline_folders: Dict[str, str]):
    """Configure input structures."""
    self.folders = pipeline_folders
    self._setup_file_paths()

    # Get structure files from input
    if hasattr(self.structures_input, 'structures'):
        self.input_structures = self.structures_input.structures
    else:
        raise ValueError("Could not get structures from input")

    if not self.input_structures:
        raise ValueError("No structures found in input")

def generate_script(self, script_path: str) -> str:
    """Generate MyTool execution script."""
    tool_exe = os.path.join(self.folders["HelpScripts"], "pipe_my_tool.py")

    script_content = f"""#!/bin/bash
# MyTool execution script
# Generated by BioPipelines pipeline system

echo "Running MyTool"
echo "Input structures: {len(self.input_structures)}"

python {tool_exe} \\\
--structures "{','.join(self.input_structures)}" \\\
--param1 "{self.param1}" \\\
--param2 {self.param2} \\\
--output "{self.results_csv}"

if [ $? -eq 0 ]; then
    echo "MyTool completed successfully"
else
    echo "Error: MyTool failed"
    exit 1
fi
"""

    return script_content

def get_output_files(self) -> Dict[str, List[str]]:
    """Get expected output files."""
    datasheets = {
        "results": DatasheetInfo(
            name="results",

```

```

        path=self.results_csv,
        columns=["id", "param1_value", "param2_value"],
        description="MyTool analysis results",
        count=len(self.input_structures) if hasattr(self, 'input_structures') else 0
    )
}

return {
    "structures": [],
    "structure_ids": [],
    "sequences": [],
    "sequence_ids": [],
    "datasheets": datasheets,
    "output_folder": self.output_folder
}

def to_dict(self) -> Dict[str, Any]:
    """Serialize configuration."""
    base_dict = super().to_dict()
    base_dict.update({
        "tool_params": {
            "param1": self.param1,
            "param2": self.param2
        }
    })
    return base_dict

```

Step 2: Update Documentation

Add to Docs/UserManual.md under appropriate section:

```

### MyTool
**Environment**: ProteinEnv

**Parameters**:
- structures: Union[ToolOutput, StandardizedOutput] (required) - Input structures
- param1: str (required) - Description
- param2: int = 10 - Description

**Outputs**:
- datasheets.results:

    | id | param1_value | param2_value |
    |----|-----|-----|

**Example**:


```
python
result = pipeline.add(MyTool(
 structures=boltz,
 param1="value",
 param2=20
))

```


```

Step 3: Create Helper Script

Create HelpScripts/pipe_my_tool.py:

```
#!/usr/bin/env python3
"""
Helper script for MyTool - executed during SLURM runtime.
"""

import argparse
import pandas as pd

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--structures', required=True)
    parser.add_argument('--param1', required=True)
    parser.add_argument('--param2', type=int, required=True)
    parser.add_argument('--output', required=True)
    args = parser.parse_args()

    structures = args.structures.split(',')

    # Process structures
    results = []
    for struct in structures:
        result = {
            'id': os.path.basename(struct).replace('.pdb', ''),
            'param1_value': args.param1,
            'param2_value': args.param2
        }
        results.append(result)

    # Save results
    df = pd.DataFrame(results)
    df.to_csv(args.output, index=False)
    print(f"Saved {len(results)} results to {args.output}")

if __name__ == "__main__":
    main()
```

Input/Output Patterns

Input Handling

Pattern 1: StandardizedOutput Input (Preferred)

```
def __init__(self, structures: Union[ToolOutput, StandardizedOutput], ...):
    self.structures_input = structures
    if hasattr(structures, 'config'):
        self.dependencies.append(structures.config)

def configure_inputs(self, pipeline_folders):
    if hasattr(self.structures_input, 'structures'):
        self.input_structures = self.structures_input.structures
```

```

else:
    raise ValueError("Could not get structures from input")

```

Pattern 2: Multiple Input Types

```

def __init__(self, proteins: Union[str, List[str], ToolOutput, StandardizedOutput], ...):
    if isinstance(proteins, StandardizedOutput):
        self.input_sequences = proteins.sequences
    elif isinstance(proteins, ToolOutput):
        self.input_sequences = proteins
        self.dependencies.append(proteins.config)
    elif isinstance(proteins, list):
        self.input_sequences = proteins
    else:
        self.input_sequences = [str(proteins)]

```

Pattern 3: Datasheet References

```

def __init__(self, metric: Union[str, tuple], ...):
    # Can accept direct column reference
    # tool.datasheets.analysis.distance -> (DatasheetInfo, "distance")

def configure_inputs(self, pipeline_folders):
    if isinstance(self.metric, tuple):
        datasheet_info, column_name = self.metric
        self.metric_datasheet = datasheet_info.path
        self.metric_column = column_name

```

Output Definition

Standard Output Structure

```

def get_output_files(self) -> Dict[str, List[str]]:
    return {
        # Structure outputs
        "structures": [list of PDB paths],
        "structure_ids": [list of IDs],

        # Sequence outputs
        "sequences": [list of CSV/FASTA paths],
        "sequence_ids": [list of IDs],

        # Compound outputs
        "compounds": [list of compound CSV paths],
        "compound_ids": [list of IDs],

        # Datasheets with metadata
        "datasheets": {
            "analysis": DatasheetInfo(...),
            "results": DatasheetInfo(...)
        },

        # Tool folder
        "output_folder": self.output_folder
    }

```

DatasheetInfo Pattern

```
datasheets = {
    "analysis": DatasheetInfo(
        name="analysis",
        path=os.path.join(self.output_folder, "analysis.csv"),
        columns=["id", "metric1", "metric2"],
        description="Analysis results description",
        count=expected_row_count
    )
}
```

Script Generation

Script Template

```
def generate_script(self, script_path: str) -> str:
    """Generate execution script."""

    # 1. Define paths using self.folders
    tool_exe = os.path.join(self.folders["ToolDir"], "executable")
    helper_script = os.path.join(self.folders["HelpScripts"], "pipe_helper.py")

    # 2. Build script content
    script_content = f"""#!/bin/bash
# {self.TOOL_NAME} execution script
# Generated by BioPipelines pipeline system

echo "Running {self.TOOL_NAME}"

# Main command
{tool_exe} \\\
--input "{self.input_file}" \\\
--output "{self.output_folder}" \\\
--param {self.param_value}

# Check success
if [ $? -eq 0 ]; then
    echo "{self.TOOL_NAME} completed successfully"
else
    echo "Error: {self.TOOL_NAME} failed"
    exit 1
fi
"""

    return script_content
```

Multi-Step Scripts

```
def generate_script(self, script_path: str) -> str:
    """Generate multi-step script."""

    script_content = f"""#!/bin/bash
# {self.TOOL_NAME} execution script
```

```

# Step 1: Preparation
echo "Step 1: Preparing inputs"
python {self.prep_script} "{self.input_file}" "{self.temp_folder}"

# Step 2: Main execution
echo "Step 2: Running main tool"
{self.main_exe} --input "{self.temp_folder}" --output "{self.output_folder}"

# Step 3: Post-processing
echo "Step 3: Processing results"
python {self.postprocess_script} "{self.output_folder}"

echo "All steps completed"
"""

    return script_content

```

Testing Tools

Unit Test Template

Create tests/test_my_tool.py:

```

import pytest
import os
from PipelineScripts.my_tool import MyTool
from PipelineScripts.pipeline import Pipeline

def test_mytool_initialization():
    """Test tool initialization."""
    tool = MyTool(
        structures="test.pdb",
        param1="value",
        param2=10
    )
    assert tool.param1 == "value"
    assert tool.param2 == 10

def test_mytool_validation():
    """Test parameter validation."""
    with pytest.raises(ValueError, match="param1 is required"):
        tool = MyTool(structures="test.pdb", param1="", param2=10)
        tool.validate_params()

def test_mytool_in_pipeline():
    """Test tool in pipeline context."""
    pipeline = Pipeline("TestPipeline", "TestJob", "Testing", debug=True)

    result = pipeline.add(MyTool(
        structures="test.pdb",
        param1="value",
        param2=10
    ))

```

```

# Check outputs are predicted correctly
outputs = result.get_output_files()
assert "datasheets" in outputs
assert outputs["output_folder"]

```

Integration Test

```

def test_tool_chain():
    """Test tool in chain with other tools."""
    pipeline = Pipeline("TestPipeline", "ChainTest", "Test chain", debug=True)

    # Add tools in sequence
    tool1 = pipeline.add(ToolA(param="value"))
    tool2 = pipeline.add(MyTool(structures=tool1, param1="test"))

    # Verify dependency chain
    assert tool1.config in tool2.dependencies

    # Check outputs are accessible
    assert tool2.get_output_files()["datasheets"]

```

Code Principles

1. Explicit Failures

Never: - Use default/fallback values - Guess file paths - Silently skip errors

Always: - Raise ValueError with exact error message - Specify what was expected vs what was found - Crash explicitly if requirements not met

```

# Bad
if not os.path.exists(file_path):
    file_path = "default.txt" # Fallback

# Good
if not os.path.exists(file_path):
    raise ValueError(f"File not found: {file_path}. Expected in {expected_dir}")

```

2. Single Definition Principle

Define paths and values once, use everywhere:

```

# Bad
def generate_script(self):
    csv_path = os.path.join(self.output_folder, "results.csv")
    # ... later ...
    another_csv = os.path.join(self.output_folder, "results.csv") # Duplicate

# Good
def _setup_file_paths(self):
    self.results_csv = os.path.join(self.output_folder, "results.csv")
    self.summary_csv = os.path.join(self.output_folder, "summary.csv")

```

```
def generate_script(self):
    # Use self.results_csv consistently
```

3. Tool Agnostic Design

Tools must work without knowing about other specific tools:

```
# Bad
if isinstance(input_tool, Boltz2):
    structures = input_tool.boltz2_specific_output

# Good
if hasattr(input_tool, 'structures'):
    structures = input_tool.structures
else:
    raise ValueError("Input must have structures attribute")
```

4. Prediction-Based Paths

Never check file existence during configuration:

```
# Bad
def configure_inputs(self):
    if os.path.exists(predicted_file):
        self.input_file = predicted_file

# Good
def configure_inputs(self):
    # Predict path without checking existence
    self.input_file = os.path.join(self.folders["data"], "file.txt")
    # File will exist at SLURM runtime
```

Error Handling

Validation Errors

```
def validate_params(self):
    """Provide specific, actionable error messages."""

    # Check required parameters
    if not self.required_param:
        raise ValueError(
            f"required_param is mandatory for {self.TOOL_NAME}"
        )

    # Check types
    if not isinstance(self.numeric_param, (int, float)):
        raise ValueError(
            f"numeric_param must be int or float, got {type(self.numeric_param)}"
        )

    # Check ranges
    if self.numeric_param <= 0:
        raise ValueError(
            f"numeric_param must be positive, got {self.numeric_param}"
        )
```

```

    )

    # Check enums
    valid_options = ["opt1", "opt2", "opt3"]
    if self.choice_param not in valid_options:
        raise ValueError(
            f"choice_param must be one of {valid_options}, got '{self.choice_param}'"
        )

```

Configuration Errors

```

def configure_inputs(self, pipeline_folders):
    """Provide context in error messages."""

    self.folders = pipeline_folders

    if isinstance(self.structures, StandardizedOutput):
        if not hasattr(self.structures, 'structures'):
            raise ValueError(
                f"Input StandardizedOutput missing 'structures' attribute. "
                f"Available: {list(self.structures._data.keys())}"
            )

        if not self.structures.structures:
            raise ValueError(
                f"Input has structures attribute but it's empty. "
                f"Source tool: {self.structures.tool_type}"
            )

    self.input_structures = self.structures.structures

```

Documentation

Docstring Format

```

class MyTool(BaseConfig):
    """
    Single-line description of what the tool does.

    Longer description explaining when to use this tool,
    what it's good for, and any important limitations.

    Commonly used for:
    - Specific use case 1
    - Specific use case 2
    """

    def __init__(self, param1: str, param2: int = 10, **kwargs):
        """
        Initialize MyTool configuration.

        Args:
            param1: Description of param1 (required)

```

```

    param2: Description of param2 (default: 10)
    **kwargs: Additional parameters passed to BaseConfig

Examples:
    Basic usage:
    >>> tool = MyTool(param1="value", param2=5)

    With tool output:
    >>> tool = MyTool(structures=boltz_output, param1="test")
    """

```

Method Documentation

```

def configure_inputs(self, pipeline_folders: Dict[str, str]):
    """Configure input sources from pipeline context.

    Sets up:
    - self.folders: Pipeline folder structure
    - self.input_structures: List of structure file paths
    - self.dependencies: Tool execution dependencies

    Raises:
    ValueError: If structures cannot be extracted from input
    ValueError: If required files don't exist
    """

```

Quick Reference

Tool Development Checklist

- ☐ Create tool file in PipelineScripts/
- ☐ Inherit from BaseConfig
- ☐ Set TOOL_NAME and DEFAULT_ENV
- ☐ Implement validate_params()
- ☐ Implement configure_inputs()
- ☐ Implement generate_script()
- ☐ Implement get_output_files()
- ☐ Create helper script pipe_.py in HelpScripts/ if needed
- ☐ Add documentation to UserManual.md
- ☐ Test with simple pipeline

Common Patterns

File paths: Define once in `_setup_file_paths()`, use everywhere **Validation:** Raise `ValueError` with specific messages, no fallbacks **Dependencies:** Append to `self.dependencies` for tool inputs **Outputs:** Use `DatasheetInfo` for all CSV outputs **Errors:** Crash explicitly with helpful error messages

Key Commands

```

# Test tool
python -c "from PipelineScripts import Pipeline, MyTool; ..."

# Run pipeline
python my_pipeline.py

```

```
# Submit to SLURM  
sbatch --job-name=JobName path/to/slurm.sh
```