Pipelining Tools for HPC Workflows

Using Bash, Snakemake and Nextflow

Yale Center for Research Computing

Agenda

- **The Problem**: why pipelines?
- **Pipelining concepts**
- **An example workflow**
- **Bash & Slurm**
- **Snakemake**: turning our example into a Snakemake pipeline
- **Break**: 10minute break
- **Nextflow**: using pipelines from the research community
- **Resources**

Setup

Log in to the cluster and clone the workshop repository:

```
git clone https://github.com/ycrc/pipelines-workshop.git
cd pipelines-workshop
ls examples/
```

You will need a terminal and a text editor.

We recommend an Open OnDemand VS Code session.

The Problem

Your Workflow

- Multiple steps that process input to produce output
- Some steps depend on others completing first
- It works — now you need to run it many times, scale it up, or share it

```
# step 1: process raw data
./clean.sh raw.dat > clean.dat

# step 2: run analysis
./analyze.sh clean.dat > results.dat

# step 3: make figures
./plot.sh results.dat > fig.png
```

What Can Go Wrong

- Script versions multiply
- Data folders accumulate
- "It worked on my machine"
- A step fails halfway — is the output valid?

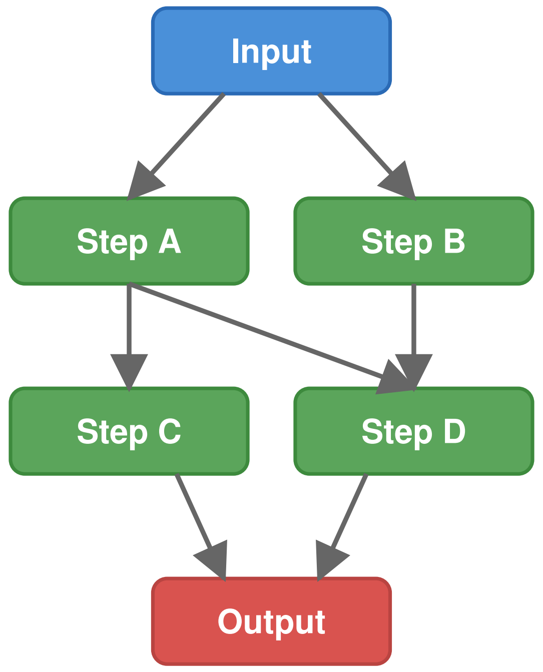
Today's Learning Goals

- Understand key concepts for constructing data pipelines
- Build a simple workflow using bash scripts and Slurm
- Translate that workflow into a Snakemake pipeline
- Run a community-maintained pipeline using Nextflow and nf-core

Pipelining Concepts

Flowcharts and DAGs

- A workflow is a **directed acyclic graph** (DAG)
- Nodes are tasks, edges are dependencies
- No cycles — a task can't depend on its own output



Atomicity

- Every step of a pipeline should be **atomic**: it either fully succeeds, or fully fails.
- If a step fails, it should not produce partial output
- Prevents downstream steps from running on bad data

Reproducibility

- **Same input** + **same options** = **same output**
- Portable: works the same on any system
- Version control your pipeline, not just your analysis
- Pipelining tools have features to log exactly what processing was run in what order, with what parameters.

Our Example Workflow

The Input Data

- 10 plays by William Shakespeare
- UTF-8 plaintext files
- Stand-in for your real data: genomic reads, simulation output, etc.
- Small enough to run in a workshop, but the tools scale

The Goal

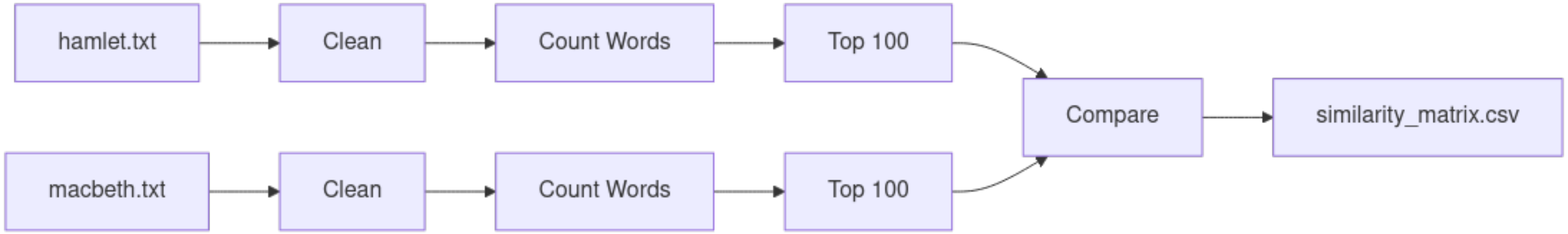
- Compute a measure of similarity between each pair of plays based on their most common words.

The Workflow

1. **Clean** each play (lowercase, remove punctuation)
2. **Count** word frequencies
3. **Extract** top 100 words per play
4. **Compare** every pair of plays (Jaccard similarity)
5. **Combine** into a similarity matrix CSV

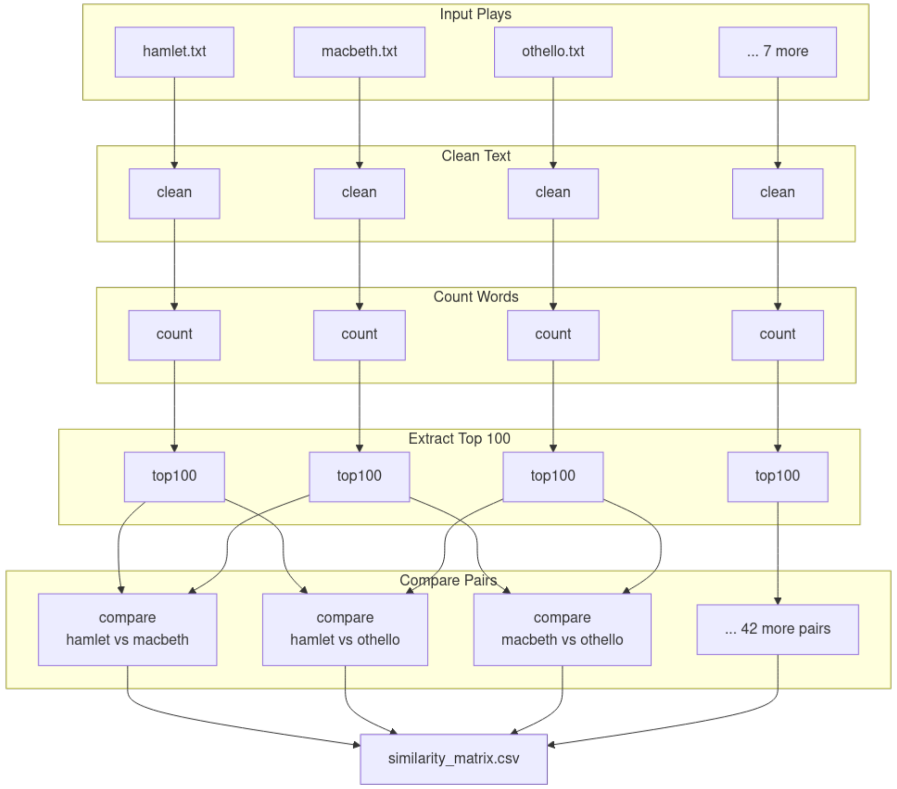
The DAG (Simplified)

For two plays, the workflow looks like this:



The DAG (Full)

With all 10 plays, the DAG fans out — 45 compare steps:



The Bash Scripts

Our original scripts are found in the workshop repository under `examples/bash/`:

Script	Purpose
`01_analyze_play.sh`	Clean text, count words, extract top 100
`02_compare_plays.sh`	Jaccard similarity between two plays
`03_combine_results.sh`	Aggregate results into CSV
`00_run_all.sh`	Run everything in order

01_analyze_play.sh — Overview

Takes one play name as input, produces its top 100 words.

```
# Usage: ./analyze_play.sh <play>

PLAY="$1"
INPUT="data/${PLAY}.txt"
```

Three steps: **clean** → **count** → **extract top 100**

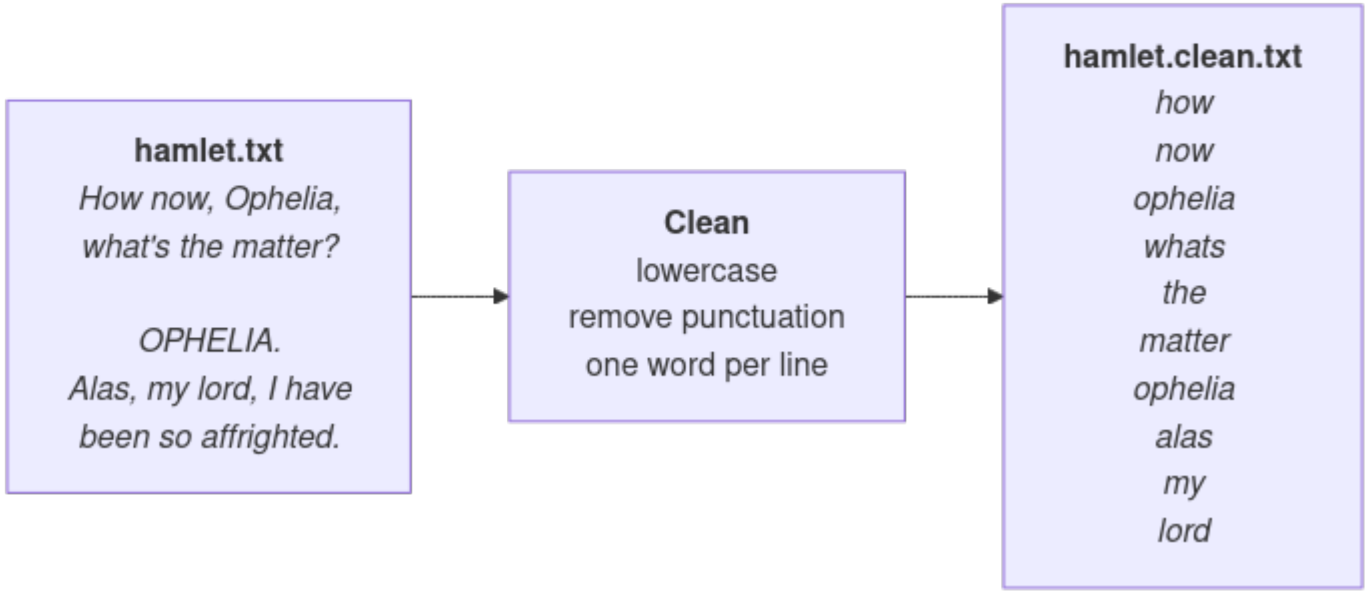
01 — Step 1: Clean the Text

Convert to lowercase, remove punctuation, one word per line:

```
cat "$INPUT" \
    | tr '[:upper:]' '[:lower:]' \
    | tr -d '[:punct:]' \
    | tr -s '[:space:]' '\n' \
    > output/${PLAY}.clean.txt
```

- `tr '[:upper:]' '[:lower:]'` — lowercase everything
- `tr -d '[:punct:]'` — delete punctuation
- `tr -s '[:space:]' '\n'` — squeeze whitespace, one word per line

01 — Step 1: What It Looks Like



01 — Step 2: Count Word Frequencies

Sort words, count unique occurrences, sort by frequency:

```
cat output/${PLAY}.clean.txt \
    | sort \
    | uniq -c \
    | sort -rn \
    > output/${PLAY}.counts.txt
```

Output looks like:

```
   1138 the
    674 and
    594 of
    ...
```

Y | CRC

01 — Step 3: Extract Top 100

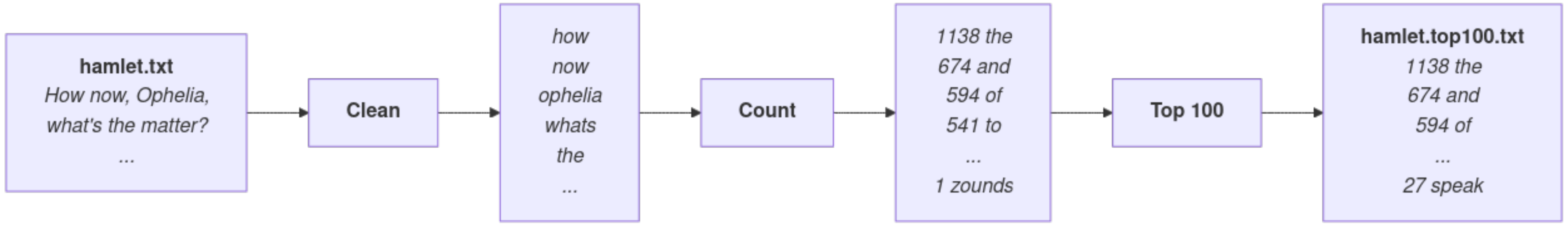
Keep only the 100 most frequent words, clean up intermediates:

```
head -100 output/${PLAY}.counts.txt > output/${PLAY}.top100.txt

rm output/${PLAY}.clean.txt
rm output/${PLAY}.counts.txt
```

- `data/hamlet.txt` → `output/hamlet.top100.txt`
- Intermediate `.clean.txt` and `.counts.txt` are deleted

01 — The Full Picture



02_compare_plays.sh — Overview

Takes two play names, computes their **Jaccard similarity**.

```
PLAY1="$1"
PLAY2="$2"
FILE1="output/${PLAY1}.top100.txt"
FILE2="output/${PLAY2}.top100.txt"
```

Jaccard = |intersection| / |union| of their top-100 word sets.

02 — Step 1: Extract Word Lists

Strip the count column, keep just the words:

```
awk '{print $2}' "$FILE1" > output/${PLAY1}.words.txt
awk '{print $2}' "$FILE2" > output/${PLAY2}.words.txt
```

02 — Step 2: Find Common Words

Use `comm` to find the intersection of sorted word lists:

```
comm -12 \
    <(sort output/${PLAY1}.words.txt) \
    <(sort output/${PLAY2}.words.txt) \
    > output/common.txt
```

- `comm -12` suppresses lines unique to either file
- Only lines common to **both** files are kept

02 — Step 3: Calculate Jaccard Similarity

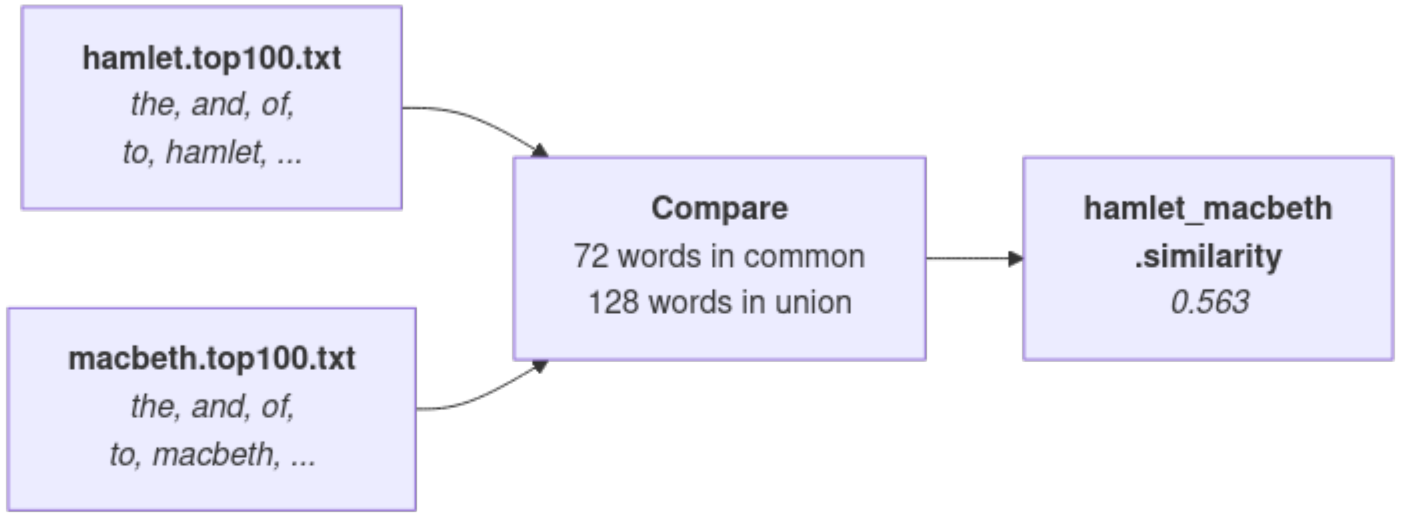
```
COMMON=$(wc -l < output/common.txt)
TOTAL1=$(wc -l < output/${PLAY1}.words.txt)
TOTAL2=$(wc -l < output/${PLAY2}.words.txt)

UNION=$((TOTAL1 + TOTAL2 - COMMON))
SIMILARITY=$(echo "scale=3; $COMMON / $UNION" | bc)

echo "${SIMILARITY}" > output/${PLAY1}_${PLAY2}.similarity
```

- `bc` handles decimal division (bash only does integers)
- Output: a single file like `output/hamlet_macbeth.similarity`

02 — The Full Picture



03_combine_results.sh

Loop through all `.similarity` files, build a CSV:

```
echo "play1,play2,similarity" > output/similarity_matrix.csv

for file in output/*.similarity; do
    basename=$(basename "$file" .similarity)
    play1=$(echo "$basename" | cut -d'_' -f1)
    play2=$(echo "$basename" | cut -d'_' -f2-)
    similarity=$(cat "$file")

    echo "${play1},${play2},${similarity}" \
        >> output/similarity_matrix.csv
done
```

- Final output: `output/similarity_matrix.csv`

00_run_all.sh — The Orchestrator

```bash
# Step 1: Analyze all plays
for play in data/*.txt; do
    name=$(basename "$play" .txt)
    ./analyze_play.sh "$name"
done

# Step 2: Compare all pairs
plays=(data/*.txt)
for ((i=0; i<${#plays[@]}; i++)); do
    for ((j=i+1; j<${#plays[@]}; j++)); do
        ./compare_plays.sh "$name1" "$name2"
    done
done

# Step 3: Combine results
./combine_results.sh
```

What's Wrong With This?

- Runs everything **serially** — no parallelism
- No **dependency tracking** — if one step fails, downstream runs anyway
- No **checkpointing** — must restart from scratch on failure
- **Manual cleanup** of intermediate files

Moving to Slurm

Our script works, but we're running it on the login node. We need to:

- **Request dedicated resources** — CPU, memory, time
- **Run in the background** — submit the job and come back later
- **Get notified** — email when the job finishes or fails

We can wrap `00_run_all.sh` in a Slurm job script with `#SBATCH` directives. This is better, but still a single serial job — no parallelism.

Slurm Job Script Review

Add `#SBATCH` directives at the top of your script to request resources:

`--job-name`	`--partition`	`--time`
`--cpus-per-task`	`--mem`	`--output`
`--mail-type`	`--mail-user`	

```
#SBATCH --partition=day
#SBATCH --time=00:30:00
```

Full reference: docs.ycrc.yale.edu/clusters-at-yale/job-scheduling

Hands-On: Bash + Slurm

1. Open `examples/bash/run_pipeline.sh` in your editor
2. Add `#SBATCH` directives to set job name, partition, time, resources (CPU and Memory), output file, and email notifications
3. Submit: `sbatch run_pipeline.sh`
4. Watch progress: `tail -f pipeline.out`
5. When done, check `output/similarity_matrix.csv`

The completed version is in `run_pipeline_solution.sh`.

Snakemake

What is Snakemake?

- Python-based workflow management tool
- Define **rules** with inputs, outputs, and commands that produce output from input.
- Snakemake builds the DAG and runs tasks in the right order.
- Snakemake allows you to run shell code, or Python code in your scripts.

Key Concepts

- **Snakefile**: The main file that defines the workflow
- **Rules**: Define a single step in the pipeline
  - Has an `input`, `output`, and a `shell` element.
- **Wildcards**: Create input lists from filename patterns

The Default Target: `rule all`

Snakemake works **backwards** from a target. `rule all` declares what the pipeline should produce:

```
rule all:
    input:
        "output/similarity_matrix.csv"
```

- This is always the **first rule** in the Snakefile
- Snakemake traces dependencies backwards to figure out what needs to run
- Nothing runs unless it's needed to produce this target

Translating: Clean Text

Bash

```bash
cat "$INPUT" \
  | tr '[:upper:]' '[:lower:]' \
  | tr -d '[:punct:]' \
  | tr -s '[:space:]' '\n' \
  > output/${PLAY}.clean.txt
```

Snakemake

```
rule clean_text:
    input:
        "data/{play}.txt"
    output:
        temp("output/{play}.clean.txt")
    shell:
        """
        cat {input} \
          | tr '[:upper:]' '[:lower:]' \
          | tr -d '[:punct:]' \
          | tr -s '[:space:]' '\\n' \
          > {output}
        """
```

- `{play}` is a **wildcard** — one rule handles all 10 plays
- `temp()` marks the file for automatic cleanup

Translating: Count Words

Bash

```
cat output/${PLAY}.clean.txt \
  | sort \
  | uniq -c \
  | sort -rn \
  > output/${PLAY}.counts.txt
```

Snakemake

```
rule count_words:
    input:
        "output/{play}.clean.txt"
    output:
        temp("output/{play}.counts.txt")
    shell:
        """
        sort {input} \
          | uniq -c \
          | sort -rn > {output}
        """
```

- Snakemake knows `count_words` depends on `clean_text` because the **output of one matches the input of the other**

Translating: Top 100 Words

Bash

```bash
head -100 \
  output/${PLAY}.counts.txt \
  > output/${PLAY}.top100.txt

rm output/${PLAY}.clean.txt
rm output/${PLAY}.counts.txt
```

Snakemake

```
rule top_words:
    input:
        "output/{play}.counts.txt"
    output:
        "output/{play}.top100.txt"
    shell:
        """
        head -100 {input} > {output}
        """
```

- No manual `rm` needed — `temp()` files are cleaned up automatically
- This output is **not** `temp()` because downstream rules depend on it

Translating: Compare Plays

Bash

```
comm -12 \
  <(awk '{print $2}' "$FILE1" \
    | sort) \
  <(awk '{print $2}' "$FILE2" \
    | sort) \
  > output/common.txt
# ... compute Jaccard ...
```

Snakemake

```
rule compare_plays:
    input:
        top1="output/{play1}.top100.txt",
        top2="output/{play2}.top100.txt"
    output:
        "output/{play1}_{play2}.similarity"
    shell:
        """
        COMMON=$(comm -12 \
          <(awk '{{print $2}}' \
            {input.top1} | sort) \
          <(awk '{{print $2}}' \
            {input.top2} | sort) \
          | wc -l)
        ...
        """
```

Two wildcards `{play1}` and `{play2}` handle all 45 pairs

Translating: Combine Results

This rule needs to know about **all** pair combinations upfront. We build the list at the top of the Snakefile:

```
# At the top of the Snakefile:
PLAYS, = glob_wildcards("data/{play}.txt")
PAIRS = []
for i, p1 in enumerate(PLAYS):
    for p2 in PLAYS[i+1:]:
        PAIRS.append((p1, p2))
```

- The loop generates all 45 pairs of input files automatically

Translating: Combine Results

```
rule combine_results:
    input:
        [f"output/{p1}_{p2}.similarity"
         for p1, p2 in PAIRS]
    output:
        "output/similarity_matrix.csv"
    shell:
        """
        echo "play1,play2,similarity" > {output}
        for file in {input}; do
          # parse filename, append row
        done
        """
```

Running Snakemake

When executing `snakemake`, it will find a `Snakefile` in the current directory.

- `snakemake -n` for a dry run
- `snakemake` to execute the pipeline
- `snakemake --dag | dot -Tpng > dag.png` to visualize

What You Get for Free

- Automatic dependency resolution
- Only re-runs steps whose inputs changed
- Parallel (multiple processes) execution with `-j`
- DAG visualization
- Dry-run mode

Snakemake on Slurm

- `--executor slurm` — each rule becomes a separate Slurm job
- Snakemake monitors and schedules automatically

```
snakemake -j4 --executor slurm \
  --default-resources slurm_partition=day mem_mb=1000 cpus_per_task=1
```

Hands-On: Snakemake

1. `cd examples/snakemake` and `module load snakemake`
2. Dry run: `snakemake -n`
3. Execute: `snakemake -j1`
4. Check: `cat output/similarity_matrix.csv`
5. Simulate a data change and dry-run — only affected steps re-execute:

```
touch ../data/hamlet.txt
snakemake -n   # 14 of 77 jobs will re-run
```

Demo: Snakemake on Slurm

A head job orchestrates, submitting each rule as a child Slurm job:

```
#!/bin/bash
#SBATCH --partition=day
#SBATCH --time=00:10:00
#SBATCH --mem=1G
#SBATCH --output=pipeline.out

module load snakemake
snakemake -j2 --executor slurm --latency-wait 30 \
  --default-resources slurm_partition=day \
  mem_mb=1000 cpus_per_task=1 runtime=5
```

Break

10 minutes

Nextflow

What is Nextflow?

- Groovy-based workflow management
- **Processes** and **channels**
- Built-in container support (Docker, Apptainer)
- Dataflow programming model

Key Concepts

- **Processes**: define tasks with inputs, outputs, scripts
- **Channels**: connect processes, data flows through them
- **Operators**: transform and combine channels

Snakemake vs Nextflow

	Snakemake	Nextflow
Language	Python	Groovy
Approach	File-based (rules produce files)	Dataflow (channels pass data)
Learning curve	Lower (Python syntax)	Higher (Groovy + channels)
Config	Snakefile + config.yaml	nextflow.config + profiles
Community pipelines	Snakemake Catalog	nf-core

Snakemake is more intuitive, while Nextflow has more features for complex workflows.

Side by Side: clean_text

Snakemake

```
rule clean_text:
    input:
        "data/{play}.txt"
    output:
        temp("output/{play}.clean.txt")
    shell:
        """
        cat {input} \
          | tr '[:upper:]' '[:lower:]' \
          | tr -d '[:punct:]' \
          | tr -s '[:space:]' '\\n' \
          > {output}
        """
```

Nextflow

```
process clean_text {
    input:
    path play

    output:
    tuple val(play.baseName),
          path("${play.baseName}.clean.txt")

    script:
    """
    cat ${play} \
      | tr '[:upper:]' '[:lower:]' \
      | tr -d '[:punct:]' \
      | tr -s '[:space:]' '\\n' \
      > ${play.baseName}.clean.txt
    """
}
```

Nextflow: Channels and Workflow

Where does `path play` come from? The **workflow** block wires it up:

```
// Create a channel from all .txt files
plays_ch = Channel.fromPath("data/*.txt")

workflow {
    cleaned = clean_text(plays_ch)   // each file flows into clean_text
    counted = count_words(cleaned)   // output flows into count_words
    top100  = top_words(counted)     // and so on...
}
```

- A **channel** is a stream of data flowing between processes
- Nextflow automatically parallelizes: 10 files in the channel = 10 concurrent tasks
- No wildcards or filename patterns — data flows through the DAG

Hands-On: Nextflow Shakespeare

Our Shakespeare workflow is implemented in Nextflow in `examples/nextflow/shakespeare/`.

```
cd examples/nextflow/shakespeare
module load Nextflow
nextflow run main.nf
```

When finished, inspect the output files and the execution report in `results/`.

Nextflow in Practice

Rather than re-implement our Shakespeare workflow, we'll focus on the **most common real-world use case**: running an existing, community-maintained pipeline.

- Thousands of researchers use Nextflow this way every day
- Someone has already written, tested, and optimized the pipeline
- You provide your data and configuration — Nextflow does the rest

Nextflow Configuration

Configuration is separate from the pipeline code:

- `nextflow.config` — executor, resources, containers
- **Profiles** — switch between environments (local, Slurm)
- On our cluster, we use the `apptainer` profile for containers

```
// nextflow.config example for Slurm
process {
    executor = 'slurm'
    queue    = 'day'
}
apptainer {
    enabled  = true
    cacheDir = '~/scratch/apptainer_cache'
```

NF-Core Pipelines

What is nf-core?

- Community of **100+ curated Nextflow pipelines**
- Standardized structure: every pipeline works the same way
- Containerized: all software dependencies bundled
- Tested and documented by active maintainers
- Browse pipelines at https://nf-co.re/pipelines

Why Use Pre-Built Pipelines?

- **Tested by hundreds of users** — bugs found and fixed
- **Reproducible out of the box** — containers, pinned versions
- **Saves months of development** — focus on your science
- **Consistent interface** — learn one, use them all:

```
nextflow run nf-core/<pipeline> -profile test,apptainer --outdir results
```

Hands-On Setup: Start This Now

While I walk through the next slides, run this to download container images:

```
salloc
module load Nextflow
export NXF_APPTAINER_CACHEDIR=~/scratch/apptainer_cache
mkdir -p $NXF_APPTAINER_CACHEDIR
nextflow pull nf-core/rnaseq
```

This caches Apptainer images so the pipeline runs faster later.

nf-core/rnaseq

The most widely-used nf-core pipeline: bulk RNA-seq analysis.

Steps:

1. **FastQC** — raw read quality check
2. **Trim Galore** — adapter and quality trimming
3. **STAR** — align reads to reference genome
4. **Salmon** — quantify gene expression
5. **MultiQC** — aggregate QC into one report

Test profile uses a tiny yeast dataset (~50K reads).

Running nf-core/rnaseq

```
nextflow run nf-core/rnaseq -profile test,apptainer --outdir results
```

Flag	Purpose
`nf-core/rnaseq`	Pull and run the pipeline from nf-core
`-profile test`	Use built-in test dataset (yeast)
`-profile apptainer`	Use Apptainer containers
`--outdir results`	Where to write output

Runs in about **10 minutes** with 4 cores and 16GB RAM.

Inspecting Output

```
results/
├── multiqc/           # Start here: HTML summary report
├── star_salmon/       # Aligned reads + quantification
├── fastqc/            # Per-sample QC reports
├── trimgalore/        # Trimmed reads
└── pipeline_info/     # Execution timeline, versions
```

Open `results/multiqc/multiqc_report.html` for alignment rates, read quality, and gene detection at a glance.

Hands-On: nf-core/rnaseq

Run the pipeline with the test dataset:

```
nextflow run nf-core/rnaseq -profile test,apptainer --outdir results
```

While it runs, explore:

- Watch the live progress display
- When done, look at `results/multiqc/multiqc_report.html`
- Check `results/pipeline_info/` for the execution report

Finding Pipelines for Your Research

Browse https://nf-co.re/pipelines — examples:

Domain	Pipeline
RNA-seq	nf-core/rnaseq
Variant calling	nf-core/sarek
Single-cell	nf-core/scrnaseq
ATAC-seq	nf-core/atacseq
Amplicon (16S)	nf-core/ampliseq
Metagenomics	nf-core/mag
Fetch public data	nf-core/fetchngs

Resources & Next Steps

- nf-core documentation
- Nextflow training
- Snakemake documentation
- Yale HPC documentation and office hours

Workshop Feedback

Please help us improve this workshop by sharing feedback via a 2-minute anonymous survey. Thank you.

For access — click the link or scan the QR Code:
https://yalesurvey.ca1.qualtrics.com/jfe/form/SV_ac86jTriewu9l8W

Y | CRC

Questions?

Thank you!