ChatGPT

Designing a Comprehensive ChIP-Seq Simulation and Peak-Calling Pipeline

Review of Existing ChIP-Seq Pipelines and Strategies

Before constructing our custom pipeline, we surveyed several relevant pipelines and tools to gather best practices:

- **Rocketchip (2023)** – a Snakemake-based workflow designed for comparing ChIP-seq analysis components. Rocketchip allows researchers to easily swap aligners and peak callers, enabling systematic benchmarking [1] [2]. In a validation, Rocketchip ran *all combinations* of three aligners (BWA-MEM, Bowtie2, STAR) and four peak callers (MACS3, CisGenome, Genrich, PePr), with and without controls [3]. This demonstrates the feasibility of a full factorial design, similar to our goal of sweeping all parameter combinations. Rocketchip's results showed that alignment and peak-caller choices significantly influence peak outcomes – for example, Bowtie2 and BWA-MEM yielded slightly more peaks than STAR in one test [4]. This pipeline emphasizes reproducibility via Conda-managed environments [5] and organizes outputs by analysis step (e.g., separate folders for BAMs, bigWigs, and peak files) [6].

- **CHIPS (2021)** – a Snakemake pipeline from the Liu lab for ChIP/ATAC-seq processing [7]. It performs standard steps (read trimming, mapping, peak calling) and computes extensive QC metrics like the *fraction of reads in peaks (FRiP)*, PCR bottleneck coefficient, peak overlap with known regulatory sites, etc. [8] [9]. It also includes downstream analyses such as peak annotation, motif finding, and linking peaks to gene regulatory potential [8]. CHIPS illustrates a robust pipeline structure that is **flexible** (start from FASTQ or BAM, single or paired-end reads) [10] and easily extensible with additional analyses. This broad scope informs how our pipeline can remain maintainable and accommodate extra steps like visualization or annotation.

- **ChIPulate & ChIPs Simulators** – Tools like *ChIPulate* (PLOS Comput Biol 2019) and *ChIPs* (BMC Bioinf 2021) focus on **simulating** ChIP-seq data. **ChIPs** in particular is a command-line toolkit that generates realistic ChIP-seq reads under various experimental conditions [11]. It models ChIP-seq in modules (shearing, IP pulldown, PCR, sequencing) and takes known binding site locations with scores as input [12]. These simulators highlight how ground truth data can be used to benchmark peak callers – for instance, ChIPs was used to vary parameters (read depth, fragment length, noise levels) and evaluate peak-calling accuracy via recall and precision [13]. The approach of ChIPs confirms that **simulating reads with known "peaks" allows calculation of recall (fraction of true peaks detected) and precision (fraction of called peaks that are true)** for each experiment [13]. This underscores the value of our simulation-first strategy and guides how we compare called peaks to the known input distribution.

From these sources, we extract key architectural ideas: using Snakemake for workflow management, modularizing steps for flexibility, leveraging config files to swap tools (aligners/peakcallers), ensuring

reproducibility with Conda/containers, and preparing outputs and metrics for downstream QC and analysis. Next, we integrate these ideas into a tailored design.

Architectural Backbone of the Pipeline

Given our project goals, the pipeline will be organized into clear stages (simulation → alignment → peak calling → downstream analysis), with Snakemake orchestrating all steps. Below are the core components and design choices, inspired by the pipelines above and tailored to our needs:

- **Snakemake Workflow Management:** All tasks are defined as Snakemake rules in a `Snakefile` (and possibly included module files for clarity). Snakemake's wildcards and `expand` function will be heavily used to generate all combinations of parameters. This enables a full factorial sweep of parameters while letting Snakemake handle job scheduling and parallel execution [3]. We will define wildcards for each variable dimension: e.g. `aligner`, `peakcaller`, `genome`, `numPeaks_t`, `numPeaks_c`, `peakWidth_t`, `peakWidth_c`, `coverage_t`, `coverage_c`, etc., as well as any tool-specific flags.

- **Configuration for Parameters:** A YAML config (e.g. `config.yaml`) will list the values to sweep for each parameter. For example:

```yaml
genomes: ["celegans1pct.fa", "human.fa"]
aligners: ["bowtie2", "bwa"]
peakcallers: ["macs2", "epic2"]
num_peaks: [1, 2, 3, 4, 5]
peak_broadness: [1, 2, 3, 4, 5, 6]
coverage: [1, 2]      # e.g., 1X or 2X
macs2_nomodel: [true, false]  # tool-specific param for MACS2
```

This config-driven approach makes the pipeline easily extensible – to add a new aligner or peak caller, one would add it to the list and provide any required parameters (and possibly a new rule or command template, see below). Snakemake can then include those in its wildcard expansions automatically. For instance, Rocketchip uses a similar config where the user specifies alignment software (`aligner: bowtie2/bwa_mem/...`) and peak caller options [14] [15]. We will also use the config to specify file paths for reference genomes and indices, ensuring the pipeline can swap reference sequences (C. elegans fragment vs. full human genome) transparently.

- **Unique Naming of Runs:** Each combination of parameters will be associated with a unique identifier or output path so that results don't overwrite each other. We can encode parameters in file names or directory structure. For clarity, we might organize results in a hierarchy of folders, e.g. `results/{genome}/{aligner}/{peakcaller}/{paramCombo}/...`. For example, a run with bowtie2 + epic2 + specific simulation settings could output to `results/human/bowtie2/epic2/peaks_num5_vs_num1_width5_vs_2/`. This approach (used in Rocketchip and others) keeps outputs separated and easily traceable to input parameters [6]. Alternatively, a shorter combo ID can index a parameter set, with a log file mapping IDs to parameter values for reference.

- **Rule 1 – Simulation of Reads:** The pipeline's first rule invokes our custom Python simulation script to generate *paired* FASTA (or FASTQ) files for treatment and control reads, as well as the CSV with the intrinsic probability mass function (PMF) of reads across the genome. This rule will take as input the reference genome sequence (fasta) and the simulation parameters for that run. We will run it twice per combination (once for treatment, once for control) or as two sub-rules. For example:

```
rule simulate_treatment:
    output:
        "sim_reads/{genome}/{id}/treatment.fasta",
        "sim_reads/{genome}/{id}/treatment_pmf.csv"
    params:
        cov = lambda wildcards: config["coverage_value_for"][wildcards.id]
["treat"],
        num_peaks = lambda wildcards: config["param_sets"][wildcards.id]
["numPeaks_t"],
        peak_broad = ...,  # similar for broadness
        seed = 123  # (optional: to ensure reproducibility of simulation)
    shell:
        "python chipsim.py --genome {input.genome} --out {output[0]} "
        "--coverage {params.cov} --num-peaks {params.num_peaks} --peak-width
{params.peak_broad} --seed {params.seed} > {output[1]}"
```

And similarly a `simulate_control` rule for control FASTA/PMF (with `numPeaks_c`, `peak_broadness_c`, etc.). Using separate wildcards for treatment vs control parameters allows them to differ within one run (e.g. treatment might simulate 5 peaks while control simulates 1 peak, as in the example). The simulation outputs include sorted reads in FASTA (or FASTQ if quality scores added later) and a CSV listing the simulated "ground truth" distribution of reads (genomic position, strand, probability). These outputs provide the basis for later comparison to called peaks.

Why Snakemake: By making simulation a rule, we can parallelize simulations for different parameter sets and ensure each set is generated once and then reused for all downstream aligners/peakcallers. This avoids re-simulating the same conditions redundantly. If needed, we can incorporate multiple random seeds to simulate replicates, but initially one per combination is fine.

- **Rule 2 – Alignment of Reads:** For each alignment tool (bowtie2, BWA-MEM, etc.), we align the simulated reads to the reference genome. We will produce sorted BAM and index (BAI) for each of the treatment and control FASTA files. There are two possible designs for alignment rules:
- **Single generalized rule with wildcard** `aligner`**:** We use one Snakemake rule `align_reads` with an `aligner` wildcard. Inside the shell or script, we branch based on `{wildcards.aligner}` to call the appropriate aligner with its specific command-line options. For example:

```
rule align_reads:
    input:
        reads="sim_reads/{genome}/{id}/{sample}.fasta"
```

```
    output:
        bam="align/{genome}/{aligner}/{id}/{sample}.sorted.bam",
        bai="align/{genome}/{aligner}/{id}/{sample}.sorted.bam.bai"
    params:
        idx = lambda w: config["indexes"][w.genome][w.aligner]  # path to
pre-built index
    shell:
        """
        {config[aligner_envs][wildcards.aligner]}  # e.g., load conda env
        if [ "{wildcards.aligner}" == "bowtie2" ]; then
            bowtie2 -x {params.idx} -U {input.reads} -S tmp.sam --very-
sensitive;
        elif [ "{wildcards.aligner}" == "bwa" ]; then
            bwa mem {params.idx} {input.reads} > tmp.sam;
        fi
        samtools view -bS tmp.sam | samtools sort -o {output.bam};
        samtools index {output.bam};
        rm tmp.sam
        """
```

This approach uses conditional logic in one rule. It makes it easy to add a new aligner (extend the if/elif) and avoids duplicating rules. It keeps alignment outputs in a structured directory by aligner.

- **Separate rules per aligner:** Alternatively, create `rule bowtie2_align` and `rule bwa_align` etc., each with fixed shell commands. Then use a higher-level aggregation to combine them. However, given we want to seamlessly loop through aligners, the wildcard method is cleaner.

We will apply a mapping quality filter and proper sorting to ensure outputs meet peak caller requirements (most peak callers need sorted, indexed BAMs, often with duplicates marked/removed and optionally excluding low-quality alignments). As needed, we can add a post-alignment step to mark/remove duplicates (using Picard or samtools) and to filter multi-mappers, following best practices [16] [17]. These could be optional substeps toggled via config (e.g., `remove_dups: True/False`). For maintainability, we keep these steps modular – e.g., a separate `rule deduplicate` can take an aligned BAM and output a deduplicated BAM if enabled. In Rocketchip's tests, including a deduplication step was one axis of comparison [18], and they easily switched between *samtools*, *Picard*, or *sambamba* for this [18]. We can mirror that flexibility if needed, though initially we might default to one method (say, samtools markdup) for simplicity.

- **Rule 3 – Peak Calling:** This stage takes the treatment vs control BAMs and calls peaks using the specified peak caller. We'll define a wildcard for `peakcaller` and have either a unified rule or separate rules for each tool:
- Similar to alignment, a single rule `call_peaks` with wildcard `{peakcaller}` can dispatch to different commands. For example:

```
rule call_peaks:
    input:
        treat="align/{genome}/{aligner}/{id}/treatment.sorted.bam",
        control="align/{genome}/{aligner}/{id}/control.sorted.bam"
    output:
        peaks="peaks/{genome}/{aligner}/{peakcaller}/{id}/peaks.bed"
    params:
        pcaller_opts = lambda w:
            ("--nomodel" if w.peakcaller=="macs2" and
{config[macs2_nomodel]} else "")
    shell:
        """
        {config[peakcaller_envs][wildcards.peakcaller]}
        if [ "{wildcards.peakcaller}" == "macs2" ]; then
            macs2 callpeak -t {input.treat} -c {input.control} -f BAM -g
{config[genome_size][wildcards.genome]} -n {wildcards.id}
{params.pcaller_opts} --outdir peaks/{wildcards.genome}/
{wildcards.aligner}/macs2/{wildcards.id};
        elif [ "{wildcards.peakcaller}" == "epic2" ]; then
            epic2 -t {input.treat} -c {input.control} -genome
{wildcards.genome} -out {output.peaks};
        fi
        """
```

In this pseudocode, we illustrate how to conditionally add the `--nomodel` flag only for MACS2 runs (and only if the config indicates that variant). By using config flags (`macs2_nomodel: True/False`), the pipeline can **ignore tool-specific parameters when they don't apply**, achieving the requested flexibility. For example, when `peakcaller=epic2`, the `pcaller_opts` would be empty and the epic2 command simply doesn't use any MACS2-specific options. This ensures combinations like *epic2* are not "contaminated" by irrelevant parameters, while still allowing us to sweep options like MACS2's `nomodel`.

- We will output peaks in a standardized format (BED or narrowPeak). MACS2 by default produces a `.narrowPeak` file with peak summit and score info, whereas epic2 outputs a BED or similar. For consistency, we can have the rule rename or convert outputs to a common `.bed` format (containing peak intervals). Each peak caller's results will be kept in separate sub-folders (e.g. `06_{peakcaller}_peaks/` as Rocketchip does [19]) to avoid confusion. If broad peaks are simulated, we might also test MACS2's `--broad` mode or use a broad-peak caller; our design allows adding a peakcaller like SICER or MACS3-broad easily by extending config and adding a branch in the peak-calling rule.

- *Control handling:* The pipeline pairs each treatment with its control by the combination ID. We ensure the Snakemake wildcards for `id` are consistent so that the treatment/control from the same simulation run are used together. In line with common practice, if a control file is not desired for certain peak callers (or certain runs), we could allow `control=None` in the rule; but since our goal is explicitly to study background, we will always use a control in peak calling (except possibly to test

algorithms that can run without, which can be another parameter toggle). Rocketchip's analysis showed that including or excluding the control can change peak counts (Genrich, for example, found more peaks when control was omitted) [20], so our pipeline could incorporate a parameter to run peak calling with vs without control to see the difference (this would effectively treat the control BAM as optional input in Snakemake for those runs).

- **Aggregating All Combinations:** We will use the `rule all` (the final target) to launch the full parameter sweep. For example, we can dynamically generate all combinations in Python and then expand the final peak outputs:

```
import itertools
param_space = []
for genome, aligner, peakcaller in itertools.product(config["genomes"],
config["aligners"], config["peakcallers"]):
    for num_t, num_c in itertools.product(config["num_peaks"],
config["num_peaks"]):
        for broad_t, broad_c in itertools.product(config["peak_broadness"],
config["peak_broadness"]):
            for cov_t, cov_c in itertools.product(config["coverage"],
config["coverage"]):
                # incorporate special cases for peakcaller-specific param
                for nomodel in ([False] if peakcaller!="macs2" else
config["macs2_nomodel"]):
                    combo = {
                       "genome": genome, "aligner": aligner, "peakcaller":
peakcaller,
                       "numPeaks_t": num_t, "numPeaks_c": num_c,
                       "broad_t": broad_t, "broad_c": broad_c,
                       "cov_t": cov_t, "cov_c": cov_c,
                       "macs2_nomodel": nomodel
                    }
                    param_space.append(combo)
# Assign an ID to each combo and perhaps write to a CSV for reference
```

Then:

```
rule all:
    input:
        expand("peaks/{genome}/{aligner}/{peakcaller}/{id}/peaks.bed",
               zip, genome=[c["genome"] for c in param_space],
                    aligner=[c["aligner"] for c in param_space],
                    peakcaller=[c["peakcaller"] for c in param_space],
                    id=range(len(param_space)))
```

This pseudo-configuration ensures that Snakemake will create a job for each combination's peak-calling output, thereby chaining through simulation and alignment steps as needed. The structure gracefully handles the *macs2* `nomodel` parameter by only including it in combos when applicable (for other peakcallers, we either set a default or exclude it). The user can thus specify any subset of parameters to sweep, and the pipeline can scale to large sweeps or be restricted for tests by adjusting config. This design is highly **configurable** and **automatable** – new parameters (e.g., adding a `fragment_length` list for simulation) can be included with minimal changes to the Snakefile, thanks to programmatic expansion.

- **Software Environment and Reproducibility:** We will utilize either Conda environments or Docker/Singularity containers for each tool, as recommended in modern workflows [5]. Snakemake can manage Conda environments per rule (via `conda:` directives), ensuring that bowtie2, bwa, macs2, epic2, etc. run in controlled environments with specified versions. This not only makes the pipeline execution reproducible on any system but also eases maintenance (no need for the user to manually install every tool). Containerization (e.g., using a Singularity container with all tools) is an alternative; it's not essential, but we may mention it as an option if portability is a concern. Given the pipeline's complexity, we will at least use Conda as in CHIPS and Rocketchip for consistent software versions.

- **Logging and Metadata:** The pipeline will produce log files for each step (Snakemake can capture stdout/stderr to logs). We will keep these in a `logs/` directory with names indicating the rule and parameter combo. This echoes Rocketchip's `00_logs` folder for debugging [21]. We will also output a summary CSV or JSON listing all runs and the corresponding parameters (this can be generated from the `param_space` in the Snakefile). This makes it easier to track results or merge with analysis scripts later.

- **Modularity and Maintainability:** To keep the pipeline maintainable, we can split the Snakefile into include files (e.g., a `rules/` directory containing `simulation.smk`, `alignment.smk`, `peakcalling.smk`, etc.). Snakemake allows including or importing subworkflows, which is useful as the pipeline grows. Each module can define tool-specific details, making it straightforward to update or add a new aligner/peakcaller by editing one section. We will thoroughly comment the Snakefile/rules and provide example config files. The design follows a logical data flow from raw reads to final peaks, making it easier for collaborators to understand or modify one part without breaking others. By adhering to Snakemake best practices (rule names, wildcard constraints, proper input-output naming), the pipeline will be robust to changes.

Detailed Pipeline Workflow

Bringing together the components above, here's how a complete run of the pipeline would work for a given set of parameters:

1. **Simulate ChIP-seq Reads (Treatment & Control):** Using our Python simulator, generate synthetic ChIP-seq reads. The simulator will create *biased coverage* across the genome according to the specified number of peaks and peak height/broadness. For example, if `num_peaks_treatment=5` and `peak_broadness_treatment=5`, the treatment FASTA will contain reads concentrated in 5 regions (with a specified width and depth), whereas `num_peaks_control=1`, `peak_broadness_control=2` might produce a much flatter distribution with maybe 1 weak

"peak" region in the control. The **coverage** parameter (e.g. 1× vs 2×) controls total reads generated (depth), which we pass to the simulator to scale the output accordingly. Each simulation also yields a CSV with three columns (genomic base, strand, probability) representing the PMF used – effectively the "ground truth" profile of signal vs background for that sample.

2. **Align Reads to Reference:** For each aligner specified (say Bowtie2 and BWA-MEM), the pipeline aligns both treatment and control FASTAs to the reference genome. We obtain sorted BAM files and index them (e.g., `treatment.bam`, `treatment.bam.bai`, and similarly for control). Using two different aligners on the same simulated reads allows us to compare their performance downstream (since the reads and true peaks are identical for both alignments). Both Bowtie2 and BWA are widely used for ChIP-seq; Bowtie2 is known for speed and memory efficiency, while BWA-MEM is a robust mapper for longer reads [22]. Prior research suggests Bowtie2 and BWA-MEM perform similarly in peak detection, with only slight differences in peak counts [4]. Our pipeline will confirm if those differences hold under various background scenarios. Alignment rules will optionally remove duplicates and filter low-quality alignments to mimic a real processing pipeline (these steps improve peak calling specificity by reducing noise [23] [24]). After this step, for each combination, we have two BAM files ready for peak calling.

3. **Call Peaks with ChIP-Seq Peak Callers:** The aligned treatment vs control reads are then fed into each peak caller (MACS2 and Epic2 for now). MACS2 is a standard tool for narrow peak finding (often used for transcription factors), whereas Epic2 is a faster implementation of SICER, suitable for broad peaks (like histone marks). We will run MACS2 in its default (narrow peak) mode by default, but we include its `--nomodel` option as a tunable parameter in our sweeps. The `nomodel=True` setting in MACS2 skips model-based peak width adjustment – testing this could show how well MACS2 performs on broad peaks when not modeling fragment size [25]. Epic2, on the other hand, doesn't require a control but does accept one; we will always provide the control to mirror a typical ChIP vs Input experiment. Each peak caller will output a set of peaks (e.g., a BED file of regions). The pipeline will store these in an organized manner, e.g. `peaks/{genome}/{aligner}/{peakcaller}/{id}/peaks.bed`. This clear segregation means we can later compare, say, MACS2 vs Epic2 results on the same data (they'll be in parallel folders).

4. **Post-Peak Calling Processing:** After obtaining raw peak calls, the pipeline can include additional rules to process or index these results for analysis. For example, converting peak BEDs to BigBed for viewing in genome browsers, or generating bedGraph/BigWig coverage tracks from the BAMs for visualization overlay. Tools like **deepTools** `bamCoverage` can create normalized coverage bigWigs [26] [27], which we can produce as an optional output (especially since visualization is mentioned as a downstream goal). If enabled, a rule can take each sorted BAM and output a BigWig (perhaps normalized to RPGC – reads per genome coverage – so that different coverages are comparable [28]). Likewise, we could include a rule to run MultiQC over the alignment and QC logs to summarize the run, as done in some pipelines [29].

5. **Final Outputs:** The primary outputs at the end of the pipeline are:

6. Simulated read files (FASTA) for each treatment/control (for record-keeping and potential reuse).
7. Alignment files: sorted BAMs and their indices.
8. Peak files from each peak caller (in a standard format with coordinates and scores).
9. The PMF CSVs from simulation (ground truth data).

10. Optionally, coverage tracks (bigWigs) and QC reports (logs, alignment stats).
11. A summary table of all runs and perhaps basic metrics (peak counts, FRiP, etc., see below).

All these outputs are systematically arranged by parameter combination. This structure ensures compatibility with downstream analysis scripts or interactive exploration. For example, one can load the bigWig and peak BED for any given run into IGV to visually inspect how background reads influenced the called peaks, since we'll have both the signal and peaks tracks readily available.

Ensuring Extensibility and Maintainability

The pipeline is built to be **extensible**: - **Adding New Parameters:** Suppose we want to add a new simulation parameter (e.g., `fragment_length` of reads, or a parameter controlling noise level). We would add it to the config with a list of values and incorporate it into the `param_space` expansion (and pass it to the simulator script via the rule params). Because of our systematic wildcard scheme, this is straightforward and doesn't require rewriting the whole pipeline. Unused parameters can be safely ignored or given default behavior (Snakemake allows setting default values for wildcards not used in certain rules). - **Adding New Aligners/Peakcallers:** To add, for example, the STAR aligner or a new peak caller (say, HOMER or MACS3), one would: (1) install or containerize the tool, (2) add the tool name to the config lists, and (3) update the alignment or peak calling rule to handle that tool's command syntax. Thanks to the modular rule design, this might be as simple as adding another `elif` branch in the shell command. If a tool requires a very different approach (e.g., a two-step process), we can create a separate rule file and include it. Our pipeline's structure – separating concerns of simulation, alignment, and peak calling – ensures adding a tool in one stage doesn't affect the others. This aligns with Rocketchip's philosophy of easily swapping components to test different methods [30].

- **Configuration and Metadata:** By keeping all adjustable parameters in the config, we make the pipeline **self-documenting**. The config plus the output logs convey exactly what was run. Additionally, capturing the pipeline version (via Git commit or Snakefile checksum) and tool versions (via `conda list` or container hashes) in the log can help reproducibility. We will include version reporting for key tools at runtime (e.g., `bowtie2 --version` output in logs, `macs2 --version` in peak caller logs).

- **Resource Management:** Snakemake allows specifying resources (CPU, memory) per rule. We will configure reasonable defaults (e.g., alignment might use 4 threads; peak calling 1 thread by default). This ensures efficient use of computational resources, especially since a full sweep could involve many jobs. Users can scale up or down by adjusting Snakemake `--cores` and rule-specific thread usage.

In summary, the architectural backbone combines the **flexibility of Snakemake** in handling complex parameter sweeps with lessons from existing pipelines to ensure each component (simulation, mapping, peak calling) is robust and interchangeable. The result is a pipeline that can **systematically explore the impact of various parameters** (including background noise) on ChIP-seq peak detection in a reproducible and maintainable way.

Downstream Analysis for the Principle Goal

With the pipeline producing peak files under diverse conditions, we can now analyze how background reads affect peak calling outcomes. We propose several analyses and tools to accomplish the principal goal:

- **Peak Count and Peak Quality Metrics:** First, simply examine how the number of peaks called varies with different backgrounds and aligners/peakcallers. As reported by Rocketchip, different aligners or peak callers can yield significantly different peak counts on the same data [4] [31]. We will tabulate peak counts for each run (this can be extracted from MACS2 output or by counting lines in the peak BED). We should also compute the FRiP score for each treatment sample: the fraction of all treatment reads that fall within the called peak regions [9]. FRiP is a useful indicator of signal-to-noise – a high background (noisy control) might reduce FRiP by causing fewer peaks or lower peaks that capture fewer reads [9]. Our pipeline can compute FRiP easily: e.g., using `bedtools intersect` to count overlaps between BAM and peak BED, or leveraging CHIPS's approach of using a 4M read subsample [9] for consistency across coverage differences.

- **True Peak Recovery (Precision/Recall Analysis):** Since we have ground truth from simulation, we can evaluate how well each run recovered the simulated "true" peaks:

- **Recall (Sensitivity):** What fraction of the known peak regions (from the simulation's PMF or internal random peaks) were successfully called by the peak caller? We can define a simulated peak region as, say, the top X bases by probability or a window around the planted peak centers. Then use `bedtools intersect` to see if each true region overlaps a called peak. This gives recall. In the context of our simulation, recall tells us how background noise might cause true signals to be missed.
- **Precision:** What fraction of the called peaks correspond to true simulated peaks versus spurious calls in background regions? Since our control input represents background noise, any peak called that doesn't overlap a true signal region could be considered a false positive (likely caused by random clustering of background reads). By overlapping called peaks with true peak coordinates, we count how many calls are correct. Precision drops when background noise is high, as the peak caller might call peaks that are just noise.
- **F1 Score:** Combine precision and recall into a single metric to compare overall performance of each pipeline configuration [13]. This was exactly the approach used in the ChIPs simulator paper: they varied parameters and measured recall, precision, and F1 to assess peak-calling accuracy [13]. We can plot these metrics across different background levels or aligner/peakcaller choices.

Tools: **bedtools** (for overlap counts) or an R package like **ChIPQC** can aid in computing these statistics. ChIPQC, for example, can take BAMs and peaks to output various quality measures including overlaps and enrichment profiles [32]. However, a custom Python or R script using interval trees might be simplest given we have the simulated truth data.

- **Differential Peak Comparison:** Another way to gauge background impact is to compare peak sets from runs with different control backgrounds. For example, fix the treatment simulation and aligner/peakcaller, and vary the control's number of peaks or coverage – do we see peaks in the treatment+low-background scenario that disappear in the high-background scenario? We could do pairwise peak set comparisons:

- Use **BEDTools compare** or **intersect** to classify peaks into categories: peaks present with low background that are lost with high background (false negatives induced by noise), new peaks that appear only when background is high (likely false positives due to noise), and stable peaks present in both. Summarizing these categories would directly address how background noise leads to *loss of true peaks and gain of false peaks*.

- Visualize specific examples: using IGV or pyGenomeTracks, we can take representative regions where background caused a change and visually confirm that, e.g., a peak in treatment gets obscured by high input coverage.

- **Peak Shape and Enrichment Profiles:** We might analyze the height and shape of called peaks relative to the input background. For instance, using deepTools **plotProfile** or **computeMatrix**, one can plot average ChIP signal vs control signal around peak centers to see how signal-to-noise ratio changes. Additionally, cross-correlation analysis (as in phantomPeak tools) could be done on the alignments to see if high background lowers the strand cross-correlation peak (which it should, as true signal is less dominant) [33] [34]. Such analysis, while more about quality control, supports the interpretation of background effects.

- **Visualization:** For qualitative assessment, load the bigWig coverage tracks and peak BEDs into a genome browser. Because we parameterized the genome (C. elegans 1% or human, etc.), we can visualize at known coordinates. If using a small test genome region, it will be easy to see how peaks overlap with the probability distribution (the CSV can even be converted to a bigWig of "probability" per base). This visual check can be invaluable: for example, seeing that in a high-background run, the coverage difference between treatment and control is minimal for some true binding sites, potentially causing the peak caller to miss them.

- **Statistical Comparison of Peak Sets:** To rigorously quantify differences, we could use **diffbind** (an R/Bioconductor package for differential binding analysis) treating each scenario as a "condition". DiffBind could identify "peaks" that are significantly different between a low-background and high-background scenario, highlighting where background suppressed signal. This might be overkill for simulated data, but it's a possible extension.

- **Reproducibility Metrics:** If we simulate replicates or run each scenario multiple times (with different random seeds), we could examine consistency. Rocketchip found that certain aligner+caller combinations yielded non-deterministic peak counts across repeats [35] [36]. We can similarly check if our pipeline produces stable results or if some combinations show variability (e.g., does a certain aligner produce borderline peaks that appear/disappear with slight random differences?).

For implementing these analyses, we can integrate them into the pipeline or do them in separate notebooks. A reasonable approach is to output intermediate files (like the overlap of peaks with truth) and then use a Jupyter notebook or R script to compile the final figures and tables. The pipeline's job is to produce all necessary data; the analysis scripts can then read the summary CSV, peak files, and truth data to compute metrics. We will document and perhaps automate these analyses so that with one command, the entire parameter sweep runs and then generates a report (potentially using RMarkdown or Python matplotlib for plots).

To illustrate concrete analysis, consider one scenario: we simulate 5 true peaks in the treatment and none in the control (pure background). We run Bowtie2+MACS2 and Bowtie2+Epic2. Suppose MACS2 (with default settings) calls 5 peaks and epic2 calls 3 peaks. We find MACS2 had recall 100% (all 5 true peaks found) and maybe 0 false positives, whereas epic2 missed 2 true peaks (recall 60%). Why? Perhaps epic2, tuned for broad peaks, was less sensitive in narrow peak detection. In another scenario, we introduce 5 fake peaks in the control. Now MACS2 might call only 3 peaks (two true peaks got drowned in input) and possibly a false peak where input had a strong random region; epic2 might call even fewer. The FRiP scores would drop as well. Plotting recall vs number of control peaks (background level) would show a downward trend, illustrating the principle that more background reduces true peak recovery – which is exactly the principal goal of this project.

Through these analyses, supported by tools like bedtools, deepTools, and custom scripts, we will obtain a comprehensive understanding of how each parameter (especially background noise) impacts peak calling. The pipeline's design not only automates data generation for these tests, but its output structure also facilitates such comparisons. By leveraging ideas from CHIPS and others, we have also ensured the pipeline remains useful for real data processing beyond simulation, as it can produce standard QC metrics and visualizable outputs that are meaningful in any ChIP-seq workflow.

Conclusion

In conclusion, we have designed a Snakemake-based ChIP-seq pipeline that is **tailored for extensive parameter sweeps** and simulation experiments, yet built on proven architectural components from real-world pipelines. It begins with a flexible read simulation step (allowing control over peak count, peak width, coverage, and genome), followed by modular alignment and peak-calling steps that can easily swap between tools like Bowtie2/BWA and MACS2/Epic2. All combinations of parameters (including tool-specific options) can be explored systematically, with the pipeline smartly handling cases where certain options don't apply. The outputs (BAMs, peak files, etc.) are organized for downstream analysis, and the pipeline can accommodate additional steps (bigWig generation, QC metrics, peak annotation) as needed without major restructuring.

Finally, we outlined several analyses to accomplish the project's primary goal: evaluating the impact of background reads on peak calling. By comparing peak outputs against known simulated truth, calculating recall/precision [13], FRiP [8], and other QC metrics, and utilizing visualization, the pipeline will enable a thorough dissection of how each factor (background noise, coverage, aligner choice, peak caller algorithm) influences the results. This end-to-end design ensures that not only is the data processing automated and reproducible, but the scientific insights (e.g., trends in peak loss/gain with background) can be readily obtained.

Overall, our pipeline is **efficient, elegant, and maintainable**. It leverages Snakemake's strengths in reproducibility and parallelism [37], uses community best-practices from pipelines like Rocketchip and CHIPS, and remains easily extensible for future needs (such as new genomes, parameters, or analysis modules). With this in place, we will be well-equipped to systematically investigate and visualize the effects of varying background (and other parameters) on ChIP-seq peak calling outcomes, achieving the core objectives of the project.

Sources:

1. Taing *et al.*, *CHIPS: A Snakemake pipeline for chromatin profiling data – F1000Research* (2021) [8] [9]
2. Haghani *et al.*, *Improving ChIP-seq analysis workflows with Rocketchip – F1000Research* (2023) [3] [4]
3. Zheng *et al.*, *ChIPs: a flexible ChIP-seq simulation toolkit – BMC Bioinformatics* (2021) [13]
4. JetBrains Research, *Snakemake ChIP-seq pipeline (GitHub)* – Pipeline README and config [6] [25]
5. Epigenomics Workshop, *ChIP-seq data processing tutorial* (2025) [26] [28]
6. Liu Lab, *CHIPS pipeline documentation and metrics* – FRiP and QC definitions [8] [9]
7. Rocketchip GitHub Documentation – Usage of aligners/peakcallers in config [14] [15]
8. Furey, *ChIP-seq peak calling algorithms overview* – EpiGenie (for general background on MACS2/Epic2 usage) [38] [39] (if needed for context).

---

[1] [2] [3] [4] [5] [18] [20] [30] [31] [35] [36] [37] [38] [39] Improving rigor and reproducibility in chromatin immunoprecipitation assay data analysis workflows with Rocketchip - PMC
https://pmc.ncbi.nlm.nih.gov/articles/PMC11275724/

[6] [14] [15] [19] [21] GitHub - vhaghani26/rocketchip
https://github.com/vhaghani26/rocketchip

[7] [8] [9] [10] liulab-dfci.github.io
https://liulab-dfci.github.io/resources/publications/F1000Rsch10_517.pdf

[11] [12] [13] A flexible ChIP-sequencing simulation toolkit | BMC Bioinformatics | Full Text
https://bmcbioinformatics.biomedcentral.com/articles/10.1186/s12859-021-04097-5

[16] [17] [23] [24] [26] [27] [28] [33] [34] ChIP-seq data processing tutorial — Epigenomics Workshop 2025 1 documentation
https://nbis-workshop-epigenomics.readthedocs.io/en/latest/content/tutorials/chipseqProc/lab-chipseq-processing.html

[22] Onkopipe: A Snakemake Based DNA- Sequencing Pipeline for ...
https://ebooks.iospress.nl/pdf/doi/10.3233/SHTI230694

[25] [29] GitHub - JetBrains-Research/chipseq-smk-pipeline: ChIP-Seq processing pipeline on snakemake
https://github.com/JetBrains-Research/chipseq-smk-pipeline

[32] ChIP-seq quality assessment using ChIPQC - GitHub Pages
https://hbctraining.github.io/Intro-to-ChIPseq/lessons/06_combine_chipQC_and_metrics.html