Watson Pipeline Documentation

Martina Lucà

Last update: Thu 26th Sept, 2024

Contents

Part I

Python Wrapper

The Python wrapper is used to launch the pipeline analysis.

1 Start the pipeline

The pipeline is launched in the `input` folder (see an example in **Figure 1**). There are 3 key file types that **must** be present in each `Run` folder:

1. the **FASTQ files** (in `fastq.gz` format) in their appropriate folder;
2. the **file list** (in `csv` format) containing the sample IDs with their respective file names for R1 and R2;
3. the **LIMS_supp_sample_sheet** (in `csv` format) containing the information for each sample.

Do not worry if more than one R1 and/or R2 file is present for one sample ID: as long as the file list is present, the program will handle them by itself.
Given these essential files, the `config.yaml` and the `sample_info.csv` (needed for the rest of the pipeline) will be generated automatically in Run's folder.
To start the pipeline, the following command needs to be run in the `input` folder:

```
./StartPipeline.py -t [THREADS]
```

❗ **This command will run the pipeline, but you will need to stay connected to the terminal.**
If you want to run the pipeline and disconnect, you should run the command as follows:

```
nohup ./StartPipeline.py -t [THREADS] &
```

The `-t` argument is optional and indicates the number of threads. If it is not specified, it will default to 39.[1] More information about multithreading can be found in **Section 4**.

[1]*Note: bcl2fastq requires at least 16 threads.*

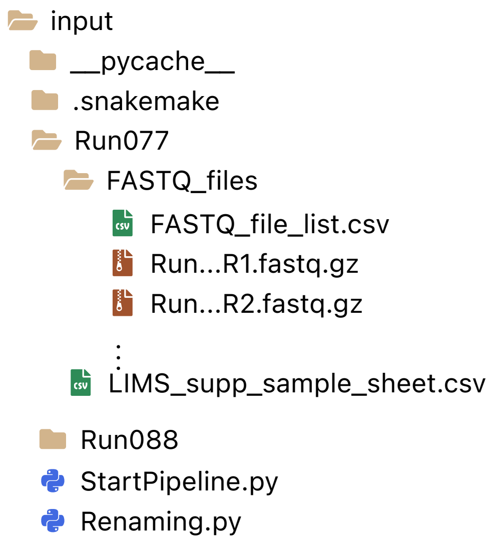


FIGURE 1. Example of the structure of the input folder.

Note that the runs' folders must have the following name format: `Runxxx`

1.1 Getting the necessary files

In the input folder, you will need to create the `Runxxx` folder and the `FASTQ_files` subfolder.

To get the necessary files, you need to navigate to the run's directory:

```
cd /media/datasafe1b/ICGC_ARGO_CORE/runxxx_xxxxxx/FASTQ_files
```

To copy the FASTQ files, run the following command:

```
for file in *.fastq.gz; do cp $PWD/$file ~/snakemake_watson/input/Runxxx/FASTQ_files/$file;
    done
```

You will then need to get the file list and the sample sheet. Do that as follows:

```
cp FASTQ_file_list.csv ~/snakemake_watson/input/Runxxx/FASTQ_files/
cp LIMS_supp_sample_sheet.csv ~/snakemake_watson/input/Runxxx/
```

1.2 Re-running the pipeline

When you are running the pipeline on FASTQs that were already processed by Watson or Holmes, the file names will not change automatically (since their names will not correspond to the ones in the file list). They most probably will be found in the following format:

```
08801_AHHC3VBGXV_Run088b01_R1.fastq.gz
```

You will need to rename them as follows:

```
Run088b01_R1.fastq.gz
```

2 Required files' format

2.1 LIMS_supp_sample_sheet.csv

This sheet should contain the following fields: LIMS_ID, Sample_ID, Project, SampleType, PreservationType, TumourType, Species, Gender, GermlineConsent, MatchNorm, AmountDNAused, LibConc, LibType, BaitsetID, SeqContID, TubeBarcode.
This sheet is needed to generate the following `sample_info.csv` file:

LIMS_ID	Gender	AmountDNAused	LibConc
Run077b09	Male	70	10.3
Run077b10	Male	70	13.8
Run077b11	Male	70	13.2
Run077b12	Male	70	11
Run077b13	Male	70	11.7
Run077b14	Female	70	12.2
Run077b15	Female	70	11.3
Run077b16	Male	70	11.2
Run010b10b	Female	50	208
Run010b11b	Female	50	193

If the `sample_info.csv` does not have the fields from this example, check the `LIMS_supp_sample_sheet.csv` file: most probably, the fields' names are not correct. To fix the issue, you can change the names of the corresponding fields.

2.2 FASTQ file list

This list is required to handle file names different from the one required for the snakefile to work. It also concatenates files coming from the same sample to have just one file for R1 and one for R2.
The format required for that file is the following:

LIMS_ID	Sample_ID	Filename_R1	Filename_R2	...
Run077b09	07709	9_S2_L002_R1_001.fastq.gz	9_S2_L002_R2_001.fastq.gz	...
Run077b09	07709	9_S2_L003_R1_001.fastq.gz	9_S2_L003_R2_001.fastq.gz	...
⋮				

In particular, the headings **Sample_ID**, **Filename_R1** and **Filename_R2** must be present (the order does not matter).

3 Changing directories

If you need to change directories for static files, snakefile, reference, output or input, you can change them in the script:

```
### --- CONFIG FILE AND SAMPLE INFO COMPILATION --- ###
# Static files
static_path = "/home/watson/snakemake_watson/static_files"
bin_path = "/home/watson/snakemake_watson/bin"
reference_path = "/home/watson/snakemake_watson/static_files/GRCh38.fa"
```

```
output_path = "/home/watson/snakemake_watson/output"

# Get runs
input_dir = "/home/watson/snakemake_watson/input"
```

LISTING 1. Code section you will need to modify if you want to change directories.

4 Multithreading

Parallel computing is handled directly by snakemake via slurm, but you can specify the number of threads to pass to the snakefile via the `-t` argument.
Since bcl2fastq requires at least 16 threads, the Python wrapper handles this as follows:

- If the number of threads is not specified, it defaults to 39
- If the number of threads is specified, it checks if the number is at least 18 and the number of threads of the machine:
    - If the number of threads of the machine is at least 18, the program sets the number of threads to 16 and throws a warning
    - If the number of threads of the machine is less than 18, the program throws an error and exits

```
# Handle threads number
if args.threads < 16:
    threads_count = psutil.cpu_count(logical=True)
    if threads_count < 18:
        sys.exit(f"ERROR: bcl2fastq requires at least 16 threads! \n You only have {
    threads_count} threads!")
    else:
        print(f"WARNING: bcl2fastq requires at least 16 threads! \n You have {threads_count}
     threads, number of threads is now set to 16.")
        T = 16
else:
    T = args.threads
```

LISTING 2. Code section you will need to change if you want to modify how you handle multithreading.

5 Output handling

Every run will have its subfolder in the output directory:

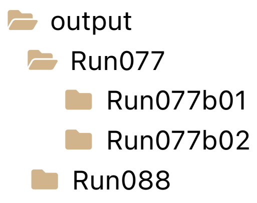


FIGURE 2. Example of the structure of the output folder.

The script will automatically remove all temporary files and the run folder in the input and rename the necessary files with the appropriate file name.

6 Checking if the pipeline was completed sucessfully

To check if every rule was completed without errors you can use the `CheckCompleted.py` script as follows:

```
./CheckCompleted.py -i output/Runxxx/logs
```

If the pipeline was completed successfully, you will see the following message:

```
All samples were analyzed successufully.
```

Otherwise, you will see a list of the rules that failed and the samples that were not analyzed. An example would be:

```
RuleName of sample SampleID was not completed.

Some samples failed. Check the logs for more information.
```

Part II
Snakemake

This Snakemake pipeline is designed to perform bioinformatics analyses on genomic data, specifically focusing on variant calling, quality control, and copy number variation analysis from DNA sequencing data (FASTQ files). The pipeline is modular, allowing for easy addition or modification of steps as needed.
The pipeline is composed of several rules, each defining a step in the analysis process. Each rule will be explained in detail, including the commands executed and the purpose of each step. Each rule contains a logging parameter that specifies where to write log outputs, allowing for easy troubleshooting and monitoring of the pipeline's execution.

7 Rules

7.1 Alignment

This rule is responsible for aligning paired-end FASTQ files, sorting and indexing the resulting BAM file, and removing duplicates.

```
bwa mem -t 12 -R '@RG\\tID:sampleID\\tSM:sampleID' REF sample.fastq.gz | bamsort
inputformat=sam markduplicates=1 rmdup=1 fixmates=1 inputthreads=8 outputthreads=8 M=
sample_metrics.txt index=1 O=sample.bam indexfilename=sample.bai;
touch {output.done};
```

Step 1

```
bwa mem -t 12 -R '@RG\\tID:sampleID\\tSM:sampleID' REF sample.fastq.gz
```

This command runs the BWA MEM algorithm, which aligns the paired-end FASTQ files to the reference genome. The options have the following functions:

- `-t` specifies the number of threads to use.
- `-R` provides the read group information for the BAM file. It includes metadata that helps downstream tools (e.g., GATK) differentiate between data from different samples or sequencing runs. In this case, it specifies the read group ID and sample name.

Step 2

```
bamsort inputformat=sam markduplicates=1 rmdup=1 fixmates=1 inputthreads=8 outputthreads
=8 M=sample_metrics.txt index=1 O=sample.bam indexfilename=sample.bai
```

This command sorts the aligned reads, marks duplicates, removes duplicates, and fixes mate-pair information. Here are what the different parameters do:

- `inputformat=sam` sort the input SAM file (the output of `bwa mem`).
- `markduplicates=1` mark the duplicate reads that result from PCR amplification during sequencing.
- `rmdup=1` remove the duplicate reads.

- `fixmates=1` fix the mate-pair information in the BAM file.
- `inputthreads=8` use 8 threads for input processing.
- `outputthreads=8` use 8 threads for output processing.
- `M=sample_metrics.txt` save the metrics of the sorting process in a file.
- `index=1` create an index file for the BAM file.
- `O=sample.bam` save the sorted BAM file.
- `indexfilename=sample.bai` specify the name of the index file.

Step 3

```
touch {output.done}
```

This command creates an empty `output.done` file to signal that the alignment rule has completed successfully. This file is used for tracking progress and directing the workflow. In the following rules, the explanation for this command will be omitted.

7.2 Alignment QC

This rule generates alignment statistics from BAM files using `Samtools stats` and processes these stats with a custom Python script.

```
samtools stats sample.bam > sample_samtools_stats.txt;
path/to/calc_align_qc.py OUTPUT_DIR sample_samtools_stats.txt;
```

Step 1

```
samtools stats sample.bam > sample_samtools_stats.txt;
```

This command generates alignment statistics from the BAM file. It provides information such as the number of reads mapped, coverage depth, and error rates.

Step 2

```
path/to/calc_align_qc.py OUTPUT_DIR sample_samtools_stats.txt;
```

This command runs a custom Python script that calculates alignment quality control metrics. The script takes two arguments: the output directory and the alignment statistics file. The script reads the alignment statistics, computes additional metrics, and writes the results to a TSV file. The additional metrics computed include total unique sequences (in megabases), percentage of unmapped reads, percentage of discordant pairs, percentage of non-reference bases and percentage of duplicate reads.

7.3 On Target Calculation

7.3.1 Generate on target calculation files

This rule processes each sample's BAM file to generate on-target statistics and depth of coverage information for specific regions (SNPs, coding, and others).

```
samtools view -F 1024 -L path/to/CORE_covered_regions.bed -u sample.bam | samtools stats
 > sample_onTarget_samtools_stats.txt;
samtools depth -b path/to/CORE_ROI_SNP.bed -o sample_SNP_depth.tmp sample.bam;
samtools depth -b path/to/CORE_ROI_CODING.bed -o sample_CODING_depth.tmp sample.bam;
samtools depth -b path/to/CORE_ROI_OTHER.bed -o sample_OTHER_depth.tmp sample.bam;
```

Step 1

```
samtools view -F 1024 -L path/to/CORE_covered_regions.bed -u sample.bam | samtools stats
 > sample_onTarget_samtools_stats.txt;
```

samtools view. Filters the BAM file by excluding duplicate reads. The flag `1024` indicates duplicate reads; only unique reads are processed. The `-L` option lmits the reads to only those that overlap the regions defined in the baitset BED file (`CORE_covered_regions.bed`).

samtools stats. Takes the output from the previous command (filtered BAM) and calculates alignment statistics (e.g., number of reads aligned to the target regions, average depth, ...). The output is saved in `sample_onTarget_samtools_stats.txt`.

Step 2 (SNPs)

```
samtools depth -b path/to/CORE_ROI_SNP.bed -o sample_SNP_depth.tmp sample.bam;
```

This command calculates the depth of coverage (number of reads covering each base) for specific regions in the BAM file. The `-b` option specifies the BED file with the regions of interest (SNPs), and the output is saved in `sample_SNP_depth.tmp`.

Step 3 (coding regions)

```
samtools depth -b path/to/CORE_ROI_CODING.bed -o sample_CODING_depth.tmp sample.bam;
```

This command calculates the depth of coverage for specific regions in the BAM file. The `-b` option specifies the BED file with the regions of interest (coding regions), and the output is saved in `sample_CODING_depth.tmp`.

Step 4 (others)

```
samtools depth -b path/to/CORE_ROI_OTHER.bed -o sample_OTHER_depth.tmp sample.bam;
```

This command calculates the depth of coverage for specific regions in the BAM file. The `-b` option specifies the BED file with the regions of interest (other regions), and the output is saved in `sample_OTHER_depth.tmp`.

7.3.2 On target calculation

This rule calculates the on-target metrics and average depth for each sample using a custom Python script and cleans up temporary files that were generated in the previous step.

```
path/to/calc_onTarget_qc.py OUTPUT_DIR sample_onTarget_samtools_stats.txt sample.bam
path/to/CORE_covered_regions.bed STATIC_DIR;
touch {output.done_3};
rm -f OUTPUT_DIR/sample_*_depth.tmp;
```

Step 1

```
path/to/calc_onTarget_qc.py OUTPUT_DIR sample_onTarget_samtools_stats.txt sample.bam
path/to/CORE_covered_regions.bed STATIC_DIR;
```

This command runs the `calc_onTarget_qc.py` script that calculates various QC metrics related to the coverage and on-target performance for sequencing data for each sample.
It reads the samtools stats file and extracts key quality control parameters (e.g., total sequences, average read length) into a dictionary. And the performs the following actions:

- Compute the total size of the target regions (in base pairs) from the baitset BED file.
- Compute the average depths for specific regions of interest (SNPs, coding regions, and other regions) by reading depth information from temporary depth files.
- Compute several metrics:
  - On-target megabases: total number of bases aligned to the target regions.
  - Design size: total size of the target regions, converted to megabases.
  - Average depth: average depth of coverage across all target regions.
  - Average depth in specific regions (SNP, coding, other): average depth of coverage in each region.
  - On-target percentage: percentage of reads aligned to the target regions.

Step 2

```
rm -f OUTPUT_DIR/sample_*_depth.tmp;
```

This command removes the temporary depth files that were created in the previous steps for calculating depth in specific regions.

7.4 Contamination Index Calculation

This rule calculates the contamination index, which helps assess the level of contamination in the sample by looking at specific SNPs that can indicate contamination when the observed allele frequencies deviate from expectations.

```
path/to/calc_ContaminationIndex.py path/to/CORE_cont_check_SNPs.bed OUTPUT_DIR sample.
bam SAMPLE_INFO;
touch {output.done_4};
rm -f sample_tmp_xy.bed sample_tmp_aut.bed;
```

Step 1

```
path/to/calc_ContaminationIndex.py path/to/CORE_cont_check_SNPs.bed OUTPUT_DIR sample.
bam SAMPLE_INFO;
```

This command runs the `calc_ContaminationIndex.py` script that estimates the contamination index for each analyzed sample using sequencing data. It creates temporary BED files for the XY chromosomes and autosomes by splitting the baitset file and retrieves gender and sample-related information from the sample information file.
Contamination calculation:

- **Y-based contamination.** For samples with female or unknown gender, the script counts the reads from chromosome Y and autosomes using samtools to detect contamination. It normalizes the read counts and calculates the percentage of contamination using predefined means and standard deviations.
- **X-based contamination.** For male samples, the script generates a pileup of SNPs on chromosome X and compares observed alleles against expected alleles from the baitset. It calculates the fraction of non-genotype alleles to estimate the contamination level.

If there is insufficient depth (too few reads), the script records that the contamination result is "Unknown - insufficient depth".

As a final step, the script appends the contamination percentages to a results tsv file. It also appends sample metadata such as gender, amount of DNA used, and library concentration.

Step 2

```
rm -f sample_tmp_xy.bed sample_tmp_aut.bed;
```

This command removes the temporary files used during the contamination index calculation.

7.5 Clipping

This rule processes BAM files by clipping reads (hard clipping), filtering out unwanted reads (those not in proper pairs, failing quality checks, or duplicates), and generates a new BAM file and its corresponding index.

```
samtools view -@ 4 -u -f 2 -F 3840  sample.bam | java -jar path/to/ClipBamOverlap.jar --
fasta REF --clipMode HARD --coordSort --output sample_clip.bam;
samtools index sample_clip.bam;
```

Step 1

```
samtools view -@ 4 -u -f 2 -F 3840  sample.bam | java -jar path/to/ClipBamOverlap.jar --
fasta REF --clipMode HARD --coordSort --output sample_clip.bam
```

samtools view. The option `-u` outputs the data in uncompressed BAM format to speed up processing, the option `-f 2` filters out reads that are not in proper pairs, and the option `-F 3840` filters out reads that are duplicates, fail quality checks, or are secondary alignments. The option `-@ 4` specifies the number of threads to use.

java. This command runs the clipping tool. he clipping mode is set to "HARD" clipping, which means the bases outside the alignment region are removed completely rather than being soft clipped (which would retain them in the sequence but mark them as clipped).

Step 2

```
samtools index sample_clip.bam;
```

After clipping, this command indexes the new clipped BAM file.

7.6 Variant Calling

This is the part of the workflow responsible for calling genetic variants using Mutect2. It takes as input BAM files and outputs a VCF file containing the called variants and uses a panel of normals (pon) for filtering.

```
path/to/java -jar path/to/gatk Mutect2 -R REF -I sample_clip.bam -O sample.vcf.gz --
panel-of-normals pon.vcf.gz --native-pair-hmm-threads 12
```

7.7 Variant Filtering

```
zless [input.vcf] | grep -v ';PON'| bcftools query - f "%CHROM\\t%POS\\t[%AD\\t%DP\\n]"
| sed 's!,!\\t!' | awk '$5>=20 && $4>=5 {{print $1"\\t"$2}}' | gzip > sample_filtered.
vcf.gz
```

Step 1

```
bcftools query - f "%CHROM\\t%POS\\t[%AD\\t%DP\\n]"
```

bcftools query. Uses bcftools, a tool for the manipulation of VCF files; it allows to extract specific information about variants. `-f` specifies the output format:

- **%CHROM**: chromosome;
- **%POS**: position on the chromosome;
- **%AD**: allele depth (ref and alt);
- **%DP**: depth of coverage.

The output is a table with comma-separated columns containing the above information.

sed. Used to substitute the comma with a tab in the output of bcftools query.

awk. Filters the output of the previous commands, selecting only the rows where the depth of coverage is greater than or equal to 20 and the allele depth of the alternative allele is greater than or equal to 5. The output is a table with two columns (chromosome and position) containing the selected variants.

Step 2

```
bcftools view -R [sample_filtered_ID.vcf.gz] [input.vcf] > sample_filtered.vcf.gz
```

`-R` specifies an input file containing a list of regions (chromosome and position) to be extracted from the compressed VCF file (`sample_filtered_ID.vcf.gz`). The variants in the input file (`input.vcf`) are filtered based on the regions in the list. The result is saved in `sample_filtered.vcf.gz`.

Step 3

```
bcftools norm -m -both [sample_filtered.vcf.gz] | ~/PROGS/vt/vt decompose_blocksub - -o
[sample_decom.vcf]
```

This command combines bcftools and vt to normalize and decompose the variants in the input file (`sample_filtered.vcf.gz`). The output is saved in `sample_decom.vcf`. The goal is to transform complex variants in simpler ones and to decompose them for a more detailed analysis.

bcftools norm. Normalizes the variants in the input file.

- `-m` merges multiallelic variants into a single record.
- `-both` decomposes both multiallelic (variants with more than one alternative allele) and complex variants (multi-nucleotide polymorphisms, MNPs) in simpler variants.

The final result is a file in wich the complex variants are separated in single variants for each allele.

Vt. A tool for variant normalization and decomposition. `decompose_blocksub` specifies for the decomposition of "block substitutions", which are variants in which multiple adjacent nucleotides are substituted simultaneously. This command subtitutes those variants with a series of single nucleotide substitutions.

Step 4

```
bcftools view --types snps [sample_decom.vcf] | bgzip > [sample_snps.vcf.gz]
```

`--types snps` filters the variants in the input file (`sample_decom.vcf` obtained in step 3) selecting only the single nucleotide polymorphisms (SNPs).
The output is saved in `sample_snps.vcf.gz`.

Step 5

```
bcftools index --tbi [sample_snps.vcf.gz]
```

Creates a TBI (Tabix Index) file for the compressed VCF file `sample_snps.vcf.gz`.

Step 6

```
bcftools view --types indels [sample_decom.vcf] | bcftools norm --fasta-ref [REF] --
output-type z --output [sample_indel_filt_norm.vcf.gz]
```

bcftools view. Filters the variants in the input file (`sample_decom.vcf` obtained in step 3) selecting only the indels.

bcftools norm. Normalizes the indels in the input file using a fasta reference (`--fasta-ref [REF]`). Normalizing variants means to align them correctly with the reference sequence, removing redundancy and representing them in a standard way.

Steps 7, 8, 9.

Basically the same as steps 4, 5, 6.

Step 10

```
bcftools view --types snps [sample_decom_CN.vcf] | bgzip > sample_decom_CN.vcf.gz
```

Filters the variants in the input file (`sample_decom_CN.vcf` obtained in step 9) selecting only the SNPs. The output is saved in `sample_decom_CN.vcf.gz`.

7.8 SV Calling (Pindel)

7.8.1 Prepare Pindel BAM

The rule generates a BAM file for Pindel, the tool used for structural variant detection, by filtering out supplementary alignment reads and creating an index for the processed BAM file.

```
samtools view -@ 4 -b -F 2048 -o sample_pindel.bam sample.bam;
samtools index sample_pindel.bam;
```

Step 1

```
samtools view -@ 4 -b -F 2048 -o sample_pindel.bam sample.bam;
```

Samtools view filters out supplementary alignment reads from the input BAM file (`sample.bam`) and saves the result in a new BAM file (`sample_pindel.bam`). The option `-F 2048` filters out supplementary alignment reads.

Step 2

```
samtools index sample_pindel.bam;
```

This command creates an index for the filtered BAM file.

7.8.2 Prepare Pindel Config

This rule generates a configuration file for Pindel by combining the sample-specific BAM file with metadata like the average insert size and the reference normal BAM file.

```
with open(params.sample_qc_file, 'r') as stream:
    for line in stream:
        s = line.strip()
        sp = s.split('\t')
        if sp[0] == 'AVERAGE_INSERT_SIZE_BP':
            sample_ins = round(float(sp[1]))
```

```
with open(output.config_pindel, 'a') as stream2:
    stream2.write( str(input.bam_pindel) + '\t' + str(sample_ins) + '\t' + str(params.
sampleID) + '\n' + str(params.normal_bam) + '\t150\tNormal')
```

7.8.3 Pindel

This rule is responsible for detecting structural variants (SVs) in a sample's genome using Pindel, generating and filtering the resulting VCF file, and then realigning indels.

```
pindel -T 10 --fasta REF --config-file sample_pindel_config.txt --output-prefix
sample_SV --anchor_quality 10 --include path/to/CORE_covered_regions.bed --
minimum_support_for_event 20 --name_of_logfile log_dir/011-SVCalling_rule_sample.log;
pindel2vcf --reference REF --reference_name GRCh38 --reference_date 20160222 --
pindel_output_root sample_SV;
bgzip sample_SV.vcf;
bcftools index --tbi sample_SV.vcf.gz;
bcftools norm --fasta-ref REF sample_SV.vcf.gz --output sample_SV_norm.vcf;
python3 path/to/filter_SV.py sample_SV_norm.vcf sample_SV_norm_filt.vcf;
bgzip sample_SV_norm_filt.vcf;
bcftools index --tbi sample_SV_norm_filt.vcf.gz;
bcftools view sample_SV_norm_filt.vcf.gz --samples sampleID -O z -o
sample_SV_filt_fin_sample.vcf.gz;
bcftools index --tbi sample_SV_filt_fin_sample.vcf.gz;
```

Step 1

```
pindel -T 10 --fasta REF --config-file sample_pindel_config.txt --output-prefix
sample_SV --anchor_quality 10 --include path/to/CORE_covered_regions.bed --
minimum_support_for_event 20 --name_of_logfile log_dir/011-SVCalling_rule_sample.log;
```

This command runs Pindel. An event will be considered only if it has an anchor quality score of 10, is in the specific region of interest (baitset BED file), and has a minimum support of 20 reads.

Step 2

```
pindel2vcf --reference REF --reference_name GRCh38 --reference_date 20160222 --
pindel_output_root sample_SV;
```

This command converts the output from Pindel to a VCF file format using the reference genome GRCh38.

Step 3

```
bgzip sample_SV.vcf;
```

This compresses the generated VCF file using `bgzip` to produce a `.gz` file.

Step 4

```
bcftools index --tbi sample_SV.vcf.gz;
```

This command creates a TBI (Tabix Index) file for the compressed VCF file `sample_SV.vcf.gz`.

Step 5

```
bcftools norm --fasta-ref REF sample_SV.vcf.gz --output sample_SV_norm.vcf;
```

This command normalizes the variants in the input file (`sample_SV.vcf.gz`) using the reference genome. Normalizing variants means to align them correctly with the reference sequence, removing redundancy and representing them in a standard way.

Step 6

```
python3 path/to/filter_SV.py sample_SV_norm.vcf sample_SV_norm_filt.vcf;
```

This command runs a Python script that filters the variants. For each variant (non-header line), it checks the genotype (GT) and allele depth (AD) for both the normal sample and the current sample. It only keeps variants where the normal sample has a genotype of 0/0 (no variant). It looks for variants in the sample that are not 0/0 and calculates the allele depth (AD) based on the genotype (0/1, 1/1, or 1/.). It only retains variants where the sample's allele depth (AD) is greater than 20.

Step 7

```
bgzip sample_SV_norm_filt.vcf;
```

This compresses the filtered VCF file using `bgzip` to produce a `.gz` file.

Step 8

```
bcftools index --tbi sample_SV_norm_filt.vcf.gz;
```

This command creates a TBI (Tabix Index) file for the compressed VCF file `sample_SV_norm_filt.vcf.gz`.

Step 9

```
bcftools view sample_SV_norm_filt.vcf.gz --samples sampleID -O z -o
sample_SV_filt_fin_sample.vcf.gz;
```

This command filters the variants in the input file (`sample_SV_norm_filt.vcf.gz`) to keep only the variants for the sample of interest (`sampleID`).

Step 10

```
bcftools index --tbi sample_SV_filt_fin_sample.vcf.gz;
```

This command creates a TBI (Tabix Index) file for the compressed VCF file `sample_SV_filt_fin_sample.vcf.gz`.

7.9 BRASS

This rule uses the tool Brass for structural variant (SV) calling, focusing on rearrangements supported by at least 10 reads, and annotates them with additional feature information.
The `run` directive is used instead of `shell` the sample quality control file is generated during the pipeline execution (i.e., it does not exist when launching the pipeline).

```
sample_bam = OUTPUT_DIR + '/' + params.sampleID + '/' + params.sampleID + '_clip.bam'
sample_qc = OUTPUT_DIR + '/' + params.sampleID + '/' + params.sampleID + '_qc.tsv'
normal_qc = os.path.join(STATIC_DIR, 'Normals_watson', 'normal_pool_qc.txt')
maxInsSize = getMaxInsertSizeForBrass(sample_qc, normal_qc)
cmd1 = "brass-group -q 10 -n 4 -F " + params.sv_repeats + " -I " + params.sv_noncontigs
+ " -m " + str(maxInsSize) + " " + sample_bam + " " + params.normal_bam + " >" + params.
brass_out
cmd2 = "grep -v \'#\' " + params.brass_out + " | awk \'$29>=10 && $9+$10+$11+$12+$13+$14
+$15+$16+$17+$18+$19+$20+$21+$22+$23+$24+$25+$26+$27+$28==0 {print $0}\' >" + params.
brass_out_filt
cmd3 = params.py_annot_brass + " " + params.brass_out_filt + " " + params.
gene_footprints + " " + output.brass_final
print(cmd1)
subprocess.call(cmd1, shell=True)
print(cmd2)
subprocess.call(cmd2, shell=True)
print(cmd3)
subprocess.call(cmd3, shell=True)
```

Step 1

```
maxInsSize = getMaxInsertSizeForBrass(sample_qc, normal_qc)
```

The rule first computes the maximum insert size for Brass, by calling a custom function that reads quality control files from both the sample and the normal control.

Step 2

```
cmd1 = "brass-group -q 10 -n 4 -F " + params.sv_repeats + " -I " + params.sv_noncontigs
+ " -m " + str(maxInsSize) + " " + sample_bam + " " + params.normal_bam + " >" + params.
brass_out
```

The first command runs Brass (brass-group) with the following options:

- `-q 10` sets the quality threshold for supporting reads to 10;
- `-n 4` specifies the number of threads to use;
- `-F` and `-I` specify the repeat regions and non-contiguous regions to exclude from the analysis;
- `-m` sets the maximum insert size;

The sample's BAM file and the normal BAM file are used as inputs to detect rearrangements.

Step 3

```
cmd2 = "grep -v \'#\' " + params.brass_out + " | awk \'$29>=10 && $9+$10+$11+$12+$13+$14
+$15+$16+$17+$18+$19+$20+$21+$22+$23+$24+$25+$26+$27+$28==0 {print $0}\' >" + params.
brass_out_filt
```

The second command filters the Brass output to keep only rearrangements supported by at least 10 reads (column 29) and with no evidence of other types of rearrangements.

Step 4

```
cmd3 = params.py_annot_brass + " " + params.brass_out_filt + " " + params.
gene_footprints + " " + output.brass_final
```

The third command runs a Python script that extracts genomic information about breakpoints from the Brass output. For each breakpoint (A and B), it checks if the genomic coordinates overlap with any genes in the gene_footprint.bed file. If genes are found, it annotates the rearrangement with the gene names. If not, it annotates with a "`.`" indicating no overlap.

7.10 Copy Number (PureCN)

This rule identifies copy number variations (CNVs) in a tumor sample by processing coverage data from BAM files, normalizing it for GC content, and applying a segmentation algorithm to detect CNVs. This rule also annotates detected CNVs with gene information, including the cytoband location.
The code is lengthy and complex, so we will not provide a detailed explanation here. The rule involves several steps:

1. `Rscript $PURECN/Coverage.R`. This R script calculates GC-normalized coverage from the tumor's BAM file using predefined genomic intervals (`CNwindows_hg38.txt`) and generates the coverage file (`sample_cov_loess`).

2. `SnpSift.jar annotate dbsnp.vcf.gz`. This Java command annotates the decomposed VCF file with known SNPs from the `dbsnp.vcf.gz` file, creating `vcf_snps_CN_annot`.

3. `PureCN.R`. This R script runs the PureCN algorithm to detect CNVs using the coverage data and the annotated VCF. It applies the "betabin" model and a segmentation algorithm (PSCBS) to identify CNVs and further optimizes the results by factoring in variant allele frequencies.

4. Several awk and grep commands are used to process the CNV gene list:

   - The CSV file (`CN_genes`) is filtered and formatted to include key columns such as gene symbol, chromosome, start/end positions, copy number, focality, and LOH (loss of heterozygosity).
   - The script then checks the CNV list against a core gene list (`CORE_genes_list.txt`) and selects only relevant genes for further analysis.

5. `cytobands_tracks.py`. This script is run to annotate the final CNV gene list with cytoband information from the `cytobands_hg38.bed` file.

6. `add_purity_to_QC.py`. This script adjusts the QC report with tumor purity information derived from the copy number analysis.

7.11 Variant Annotation

This rule is used annotate and filter variants. As for the previous rule, the code is long and complex, so we will not provide a detailed explanation here. The rule involves several steps:

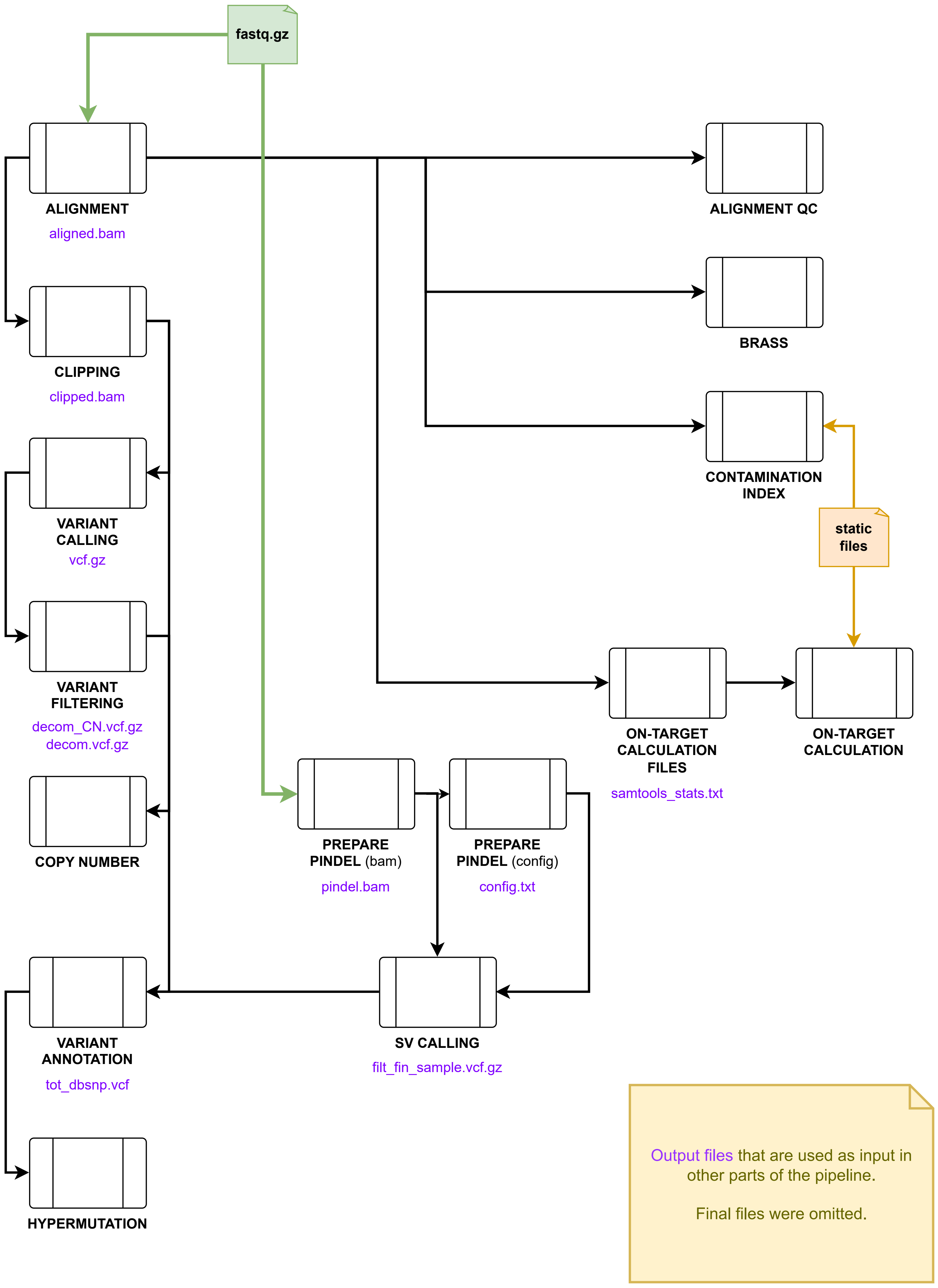
1. VEP Annotation. VEP (Variant Effect Predictor) is run on the input VCFs to annotate each variant. It uses several options to include information such as existing variations, gnomAD allele frequencies, and protein impact predictions (SIFT, PolyPhen).
2. SnpSift Annotation. After VEP annotation, the VCF is further annotated with dbSNP using SnpSift. This step integrates known dbSNP variants into the VCF.
3. Filtering. An intermediate annotation file is created by filtering variants for significance. Filters are applied to exclude benign/tolerated variants, low-impact variants, or variants in certain regions like HLA. Variants passing these filters are written to the output. For example, variants with SIFT scores of "benign" or "tolerated" are excluded. Only variants with moderate to high impact or splicing effects are kept. Structural variants undergo a similar process, but the filtering criteria are adjusted for larger changes like structural rearrangements.
4. Final annotation. The final annotation and filtering are handled by `annotateFinalFilt_new.py`, which integrates several genomic datasets (e.g., gnomAD allele counts and frequencies) into the final filtered outputs. For each variant from `finalfilt.txt`, it looks up the CN window it falls into by comparing chromosome and position to `CNwindows.bed`, and appends the CN window information. It then matches the variants with gnomAD allele frequency, count, and number data by comparing chromosome, position, reference, and alternate alleles from `gnomad_*` files.

7.12 Hypermutation

This rule identifies hypermutated samples by calculating the mutation rate and comparing it to a predefined threshold. The rule involves the following steps:

1. Tumor Mutation Burden (TMB) calculation. The script extracts relevant columns from the `vcf_var_tot_dbsnp` file, which contains all variants for the sample, filtered against dbSNP. It applies several filters to keep only variants with certain functional consequences, such as missense mutations or specific intron variants. The script calculates mutation frequencies and coverage statistics for these variants. Variants from the Panel of Normals (PoN) are excluded to avoid considering common sequencing artifacts. The TMB value is calculated as the number of mutations per megabase of sequenced genome.
2. Microsatellite Instability (MSI) calculation. The script compresses and indexes the `vcf_SV_dbsnp` file. It extracts variants located in known microsatellite regions (from the `MS_track` file) and calculates mutation frequencies and coverage statistics for these regions. The script ensures the extracted variants are filtered according to homopolymeric region length and Panel of Normals exclusion. The script `microsatellite_tracks.py` is then used to further analyze the MSI variants. Finally, the percentage of unstable microsatellite regions is calculated.
3. Old MSI calculation. The rule also calculates an "old" version of MSI by running the process on a subset of the data, using a slightly different method.

Watson flowchart



Output files that are used as input in other parts of the pipeline.

Final files were omitted.