**Supplemental**

**CONTAINER IMAGE AND REFERENCE DATA**

The following directory structure must be present on the host machine:

[+] working\_dir/  
[+] configs/  
[f] WES\_config.groovy (optional)  
[f] WGS\_config.groovy (optional)  
[ ] [...] (additional pipeline control scripts)  
[+] fastq/  
[+] normal/  
[ ] [...] (normal fastq.gz files)  
[+] tumor/  
[ ] [...] (tumor fastq.gz files)  
[+] tmp/  
[f] bpipe.config (optional)  
[+] reference\_dir/  
[+] hg19/  
[ ] [...] (hg19 reference files)  
[+] mouse\_GRCm38/  
[ ] [...] (GRCm38 reference files)

A reference bundle is also available from the software download site; it will expand to the correct directory configuration.

At container launch, the “working” (working\_dir) and “reference” (reference\_dir) directories should be mounted into the container at specific filesystem paths (working\_dir ==> /Results and reference\_dir ==> /ReferenceData). The pipeline-start.sh script automates this step.

The *working directory* should contain a configs directory for custom configuration files, a directory containing the input data for the pipeline, and a tmp directory that will be used by several tools for transient storage.

For tumor/normal runs, the pipeline must identify the difference between “tumor” inputs and “normal” inputs. By default, simply having the word “tumor” anywhere in the input file’s path marks it as a tumor sample, and having the word “normal” in the path marks a file as a normal sample. The easiest way to organize files to satisfy this requirement is to have the tumor and normal sub-directories of the data directory as shown.

The pipeline expects paired-read data, with a *fastq* file for each “end” of the paired read. The naming convention that the pipeline expects by default is that the two paired *fastq* files will contain “\_R1” and “\_R2”, respectively, in their filenames. So the following pair of files would be recognized: “SAMPLE001\_L001\_R1.fastq.gz” and “SAMPLE001\_L001\_R2.fastq.gz” Note that the two names are identical except for the “pair tags” \_R1 and \_R2. If file pairs are identified differently, the configuration files “WES\_config.groovy” or “WGS\_config.groovy” may be customized to indicate the correct pattern. Default versions of these files may be retrieved in the running container with the get-pipeline-config command.

**VCF CONSENSUS AND MERGING**

The pipeline is configured to produce consensus VCF outputs in addition to the outputs normally produced by each of the included variant callers. The following consensus outputs are produced: Union of all small variant caller outputs (“consensus:union” available in WES and WGS modes), Intersection of all small variant caller outputs (“consensus:intersection” available in WES and WGS modes), Union of all structural variant callers (“consensus:sv\_union”, WGS mode only), Intersection of all structural variant callers (“consensus:sv\_intersection”), Union of the “consensus:union” and “consensus:sv\_union” call sets (“consensus:union:union”, WGS only), and Union of the “consensus:intersection” and “consensus:sv\_union” call set (“consensus:intersection:union”, WGS only). Since some structural variants cannot be represented effectively in VCF format, we recommend also examining the raw outputs of each of the structural variant callers, which are retained in the “structural\_variants” output directory.

The process for producing each of the consensus outputs is detailed below.

**consensus:union**

The small variant union is produced by the method:

* + - * 1. Copy all VCF outputs from the small variant callers to a single directory.
        2. Run Jacquard to merge all VCFs in the directory and produce an intermediate output VCF.
        3. Correct the format of Jacquard’s output VCF to match the input requirements of Picard SortVcf by running custom Python scripts. (These scripts are available in the ‘/usr/local/lib’ directory of the container image.)
        4. Run Picard Tools SortVcf on the intermediate VCF file to produce a sorted VCF file. This file is the output of the “consensus:union” method.

**consensus:intersection**

The small variant intersection is produced by the method:

* 1. Using the “consensus:union” VCF as input, run GATK SelectVariants to select the intersection. The output of this step is the output of the “consensus:intersection” method.

**consensus:sv\_union**

The structural variant union is produced by the method:

* + 1. Using GATK CombineVariants, create the union of each VCF representation of each structural variant caller. The output of this step is the output of the “consensus:sv\_union” method.

**consensus:sv\_intersection**

The structural variant intersection is produced by the method:

* + - 1. Using the output of “consensus:sv\_union” as input, use GATK SelectVariants to select the intersection. The output of this step is the output of the “consensus:sv\_intersection” method.

**consensus:union-union**

The union of the “consensus:union” and “consensus:sv\_union” sets is produced by the method:

* + - * 1. Copy the VCF output from “consensus:union” and “consensus:sv\_union” into a temporary directory.
        2. Correct the format of Jacquard’s output VCF to match VCF format standards by running a custom Python script. (This script is available in the ‘/usr/local/lib’ directory of the container image.) The output of this stage is the output of the “consensus:union-union” method.

**consensus:intersection-union**

The union of the “consensus:intersection” and “consensus:sv\_union” sets is produced by the method:

Copy the VCF output from “consensus:intersection” and “consensus:sv\_union” into a temporary directory.

Correct the format of Jacquard’s output VCF to match VCF format standards by running a custom Python script. (This script is available in the ‘/usr/local/lib’ directory of the container image.) The output of this stage is the output of the “consensus:intersection-union” method.