
NGSPipes Documentation

Release 1.0

NGSPipes Team

May 31, 2017

Contents

1	NGSPipes overview	3
1.1	NGSPipes Team	4
2	NGSPipes DSL	5
2.1	Primitives	5
2.2	Full NGSPipes DSL syntax	9
2.3	Examples	10
3	NGSPipes repository	21
3.1	Tool names	21
3.2	Tool descriptors	22
3.3	Tool configurators	29
3.4	Defining your own tool repository	31
3.5	Tool Types	31
4	NGSPipes Editor	35
4.1	Download NGSPipes Editor	35
4.2	Execute NGSPipes Editor	35
4.3	NGSPipes Editor Sections	38
4.4	Select the tools repository	41
4.5	Creating a new Pipeline	43
4.6	Generate the final pipeline version to execute	46
4.7	Loading an existing pipeline	47
4.8	Multiple loaded pipelines	47
4.9	Error Reporting	47
4.10	Multiple inputs	49
5	Engines	51
5.1	Engine for workstation	51
5.2	Engine for cloud	66
6	Running Examples	73
6.1	A pipeline used on epidemiological surveillance	73
6.2	A pipeline used on ChIP-Seq analysis	81
6.3	A pipeline using listing tools	85

NGSPipes is a framework to easily design and use pipelines, relying on state of the art cloud technologies to execute them without users need to configure, install and manage tools, servers and complex workflow management systems.

NGSPipes overview

NGSPipes is a framework to easily design and use pipelines, relying on state of the art cloud technologies to execute them without users need to configure, install and manage tools, servers and complex workflow management systems.

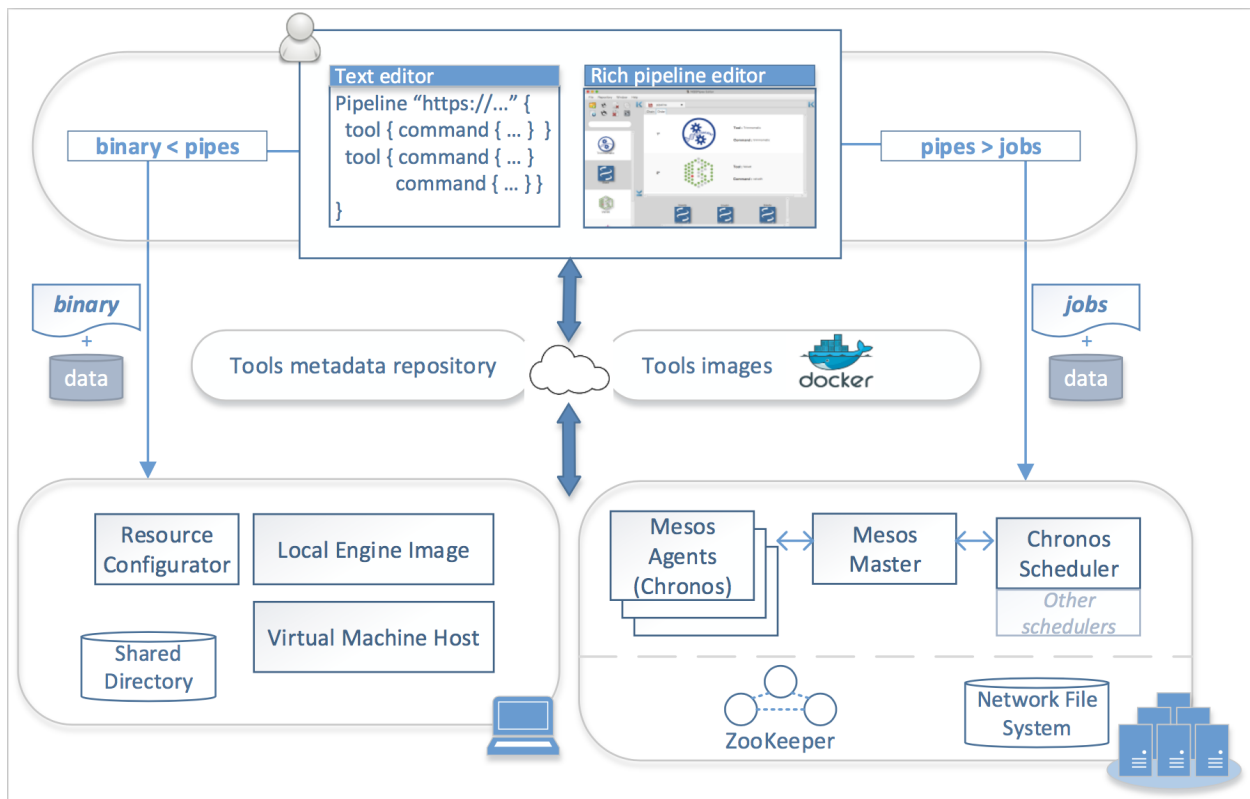


Figure 1.1: Overview of NGSPipes System.

NGSPipes Team

- Alexandre Almeida, ADEETC, ISEL, Instituto Politécnico de Lisboa
- Bruno Dantas, ADEETC, ISEL, Instituto Politécnico de Lisboa
- Calmenelias Fleitas, ADEETC, ISEL, Instituto Politécnico de Lisboa
- João Forja, ADEETC, ISEL, Instituto Politécnico de Lisboa
- Alexandre P. Francisco, INESC-ID / CSE Dept, IST, Universidade de Lisboa
- José Simão, INESC-ID / ADEETC, ISEL, Instituto Politécnico de Lisboa
- Cátia Vaz , INESC-ID / ADEETC, ISEL, Instituto Politécnico de Lisboa

For more information please contact us at ngspipes_at_gmail.com

The NGSPipes DSL is a domain specific language for describing pipelines. The syntax is described following a EBNF notation alike. As a programming language, it has some primitive building blocks with the expressiveness to define data processing, namely flow processing can be modeled as a direct acyclic graph. These primitives are defined by syntactic and semantic rules which describe their structure and meaning respectively. The primitives and the full syntax will be presented in this section. For further explaining the expressiveness of each primitive, we also incrementally introduce an example in this section, as well as the full example.

Primitives

The primitives of NGSPipes DSL are `Pipeline`, `tool`, `command`, `argument` and `chain`. In the following subsections it will be introduced the purpose of this primitives, illustrating with some examples.

Pipeline

Since a `Pipeline` is composed by the execution of one or more tools, it must be defined the tools repository, i.e., all the information necessary with respect to the available tools. To define this repository in the pipeline it is necessary to identify not only where it is stored, but also the type of storage (locally or remotely, like github) to know how to process that information. In Example 2.1 is depicted a part of a pipeline specification.

```
Pipeline "Github" "https://github.com/ngspipes/tools">{
```

Example 2.1: A partial pipeline specification, using a remote repository.

In the example of listing 2.1, “Github” is the repository type and “https://github.com/ngspipes/tools” is the location of the tool repository. The case of being a local repository is very similar, as it can be observed in Example 2.2.

```
Pipeline "Local" "E:\ngspipes" {
```

Example 2.2: A partial pipeline specification, using a local repository.

In the previous example, the tool repository is on the directory named as “ngspipes”, found at drive “E:”. Formally, the pipeline must follow the grammar in Listing 2.1.

```
pipeline: 'Pipeline' repositoryType repositoryLocation '{' (tool)+ '}' ;
```

Listing 2.1: Partial specification of the DSL grammar: pipeline specification grammar

In Listing 2.1, (tool)+ represents that a pipeline is composed by the execution of one or more tools (notice that, as will be further explained, the tool execution may include the execution of one or more commands).

tool

Each tool is specified in the pipeline by its name, its configuration file name (without extension) and by the set of commands within the tool that will be executed within this pipeline. For instance, in Example 2.3 the pipeline is composed only by one tool, which only includes a command.

```
javascript
Pipeline "Github" "https://github.com/ngspipes/tools" {
  tool "Trimmomatic" "DockerConfig" {
    command "trimmomatic" {
      argument "mode" "SE"
      argument "quality" "-phred33"
      argument "inputFile" "ERR406040.fastq"
      argument "outputFile" "ERR406040.filtered.fastq"
      argument "fastaWithAdaptersEtc" "adapters/TruSeq3-SE.fa"
      argument "seed mismatches" "2"
      argument "palindrome clip threshold" "30"
      argument "simple clip threshold" "10"
      argument "windowSize" "4"
      argument "requiredQuality" "15"
      argument "leading quality" "3"
      argument "trailing quality" "3"
      argument "minlen length" "36"
    }
  }
}
```

Example 2.3: A pipeline specification composed only by one tool, including only one command.

The tool configuration file name in Listing 2.4 is “DockerConfig”, *i.e.*, it must exist in the tool repository “https://github.com/ngspipes/tools”, within the tool information “https://github.com/ngspipes/tools/tree/master/Trimmomatic” (notice that this repository structure is directory based, as explained in https://github.com/ngspipes/tools/wiki), a configuration file named “DockerConfig”, with JSON Format. This file must define a JSON object with the property builder set as “DockerConfig”. In this case, this JSON file is https://github.com/ngspipes/tools/blob/master/Trimmomatic/DockerConfig.json. With this information together with the repository information, the environment for executing the Trimmomatic command is specified.

command

As mentioned before, there may exist a set of commands within the tool that should be executed within a pipeline. Example 2.4 depicts an example with this feature.

```
Pipeline "Github" "https://github.com/ngspipes/Repository" {
  tool "Trimmomatic" "DockerConfig" {
    command "trimmomatic" {
```

```

    argument "mode" "SE"
    argument "quality" "-phred33"
    argument "inputFile" "ERR406040.fastq"
    argument "outputFile" "ERR406040.filtered.fastq"
    argument "fastaWithAdaptersEtc" "adapters/TruSeq3-SE.fa"
    argument "seed mismatches" "2"
    argument "palindrome clip threshold" "30"
    argument "simple clip threshold" "10"
    argument "windowSize" "4"
    argument "requiredQuality" "15"
    argument "leading quality" "3"
    argument "trailing quality" "3"
    argument "minlen length" "36"
  }
}
tool "Velvet" "DockerConfig" {
  command "velveth" {
    argument "output_directory" "velvetdir"
    argument "hash_length" "21"
    argument "file_format" "-fastq"
    chain "filename" "outputFile"
  }
  command "velvetg" {
    argument "output_directory" "velvetdir"
    argument "-cov_cutoff" "5"
  }
}

```

Example 2.4: A pipeline specification composed by more than one tool and more than one command.

In example depicted in Example 2.4, the pipeline will run two tools, where the second one executes two commands of the Velvet tool, namely `velveth` and `velvetg`.

Therefore, the tools specification must follow the grammar presented in Listing 2.2.

```
tool: 'tool' toolName configurationName '{' (command)+ '}'
```

Listing 2.2: Partial specification of the DSL grammar: tool specification grammar

In Listing 2.2 `(command)+` represents that there may exist set of commands with at least a command, within the tool that should be executed within a pipeline.

For executing each command, it is necessary to identify its name, which is unique in the tool context and to set the values for each required parameters (optional parameters may not be specified). We refer the command parameters in NGSPipes language as *arguments*, since we only specify in the pipeline the parameters which we have values to set. For instance, in the previous pipeline example, the argument `filename` of the command `velveth` has as value `-fastq`, *i.e.*, the input file for this command has a FASTQ format.

Thus, the command specification must follow the grammar in Listing 2.3.

```
command : 'command' commandName '{' (argument | chain)+ '}'
```

Listing 2.3: Partial specification of the DSL grammar: command specification grammar

In Listing 2.3 `(argument | chain)+` represents that there may exist a list of arguments within this command as well as a list of chains. Chain is also a primitive in NGSPipes, as we will further explain in the subsection *chain*.

argument

As defined in the previous example, the argument definition has the syntax presented in Listing 2.4.

```
argument : 'argument' argumentName argumentValue;
```

Listing 2.4: argument syntax.

For instance, in the previous pipeline specification the `format_file` is an argument for the `velveth` tool, namely:

```
argument "file_format" "-fastq"
```

chain

The `chain` primitive allows to set an argument of a command with the produced output of other command. Sometimes the produced output is returned as files with names given internally by the command. Alternatively the output files name may be given explicitly as an argument to the command. In both situations, it is common that other commands use these output files for continue processing the pipeline. For instance, consider the following example:

```
Pipeline "Github" "https://github.com/ngspipes/Repository" {
  tool "Trimmomatic" "DockerConfig" {
    command "trimmomatic" {
      argument "mode" "SE"
      argument "quality" "-phred33"
      argument "inputFile" "ERR406040.fastq"
      argument "outputFile" "ERR406040.filtered.fastq"
      argument "fastaWithAdaptersEtc" "adapters/TruSeq3-SE.fa"
      argument "seed mismatches" "2"
      argument "palindrome clip threshold" "30"
      argument "simple clip threshold" "10"
      argument "windowSize" "4"
      argument "requiredQuality" "15"
      argument "leading quality" "3"
      argument "trailing quality" "3"
      argument "minlen length" "36"
    }
  }
  tool "Velvet" "DockerConfig" {
    command "velveth" {
      argument "output_directory" "velvetdir"
      argument "hash_length" "21"
      argument "file_format" "-fastq"
      chain "filename" "outputFile"
    }
    command "velvetg" {
      argument "output_directory" "velvetdir"
      argument "-cov_cutoff" "5"
    }
  }
  tool "Blast" "DockerConfig" {
    command "makeblastdb" {
      argument "-dbtype" "prot"
      argument "-out" "allrefs"
      argument "-title" "allrefs"
      argument "-in" "allrefs.fna.pro"
    }
    command "blastx" {
      chain "-db" "-out"
      chain "-query" "Velvet" "velvetg" "contigs_fa"
      argument "-out" "blast.out"
    }
  }
}
```

```
}
}
```

Example 2.5: A pipeline specification using the chain primitive.

As it can be seen in Example 2.5, in command `blastx`, the argument `query` receives as value the file `“contigs_fa”`, which is an output of the command `\verb+velvetg+` of the tool `velvet` (notice that in this case, the name of the file is given internally by the command).

The primitive `chain` has a simplified version, which can be used when the output is from a the previous command in the pipeline specification. In this case, we only specify the name of the output file to chain with the given argument. As an example, we can see the argument `filename` of the `velveth` command chained with the output file, named as `outputFile`, of the command `trimmomatic`.

A last version of the primitive `chain` is when the name of the tool can be omitted, but it is necessary to specify the name of the command, of the argument and also the output. This apply to cases where the chain occurs between two commands of the same tool.

Thus, the chain specification must follow the grammar depicted in Listing 2.5.

```
chain : 'chain' argumentName ( ( toolName )? commandName)? outputName;
```

Listing 2.5: Partial specification of the DSL grammar: chain specification grammar

Full NGSPipes DSL syntax

In Listing 2.6 is depicted the full NGSPipes DSL grammar.

```
pipeline: 'Pipeline' repositoryType repositoryLocation '{' (tool)+ '}' ;
tool: 'tool' toolName configurationName '{' (command)+ '}' ;
command : 'command' commandName '{' (argument | chain)+ '}' ;
argument : 'argument' argumentName argumentValue;
chain : 'chain' argumentName ( ( toolName )? commandName)? outputName;
repositoryType : String;
repositoryLocation : String;
toolName : String;
configurationName : String;
commandName : String;
argumentName : String;
argumentValue : String;
outputName : String;
toolPos: Digit;
commandPos : Digit;
String : '"' (ESC | ~["\\])* '"';
Digit : [0-9]+;
fragment ESC : '\\\' ([\\"/bfnrt] | UNICODE);
```

```
fragment UNICODE : 'u' HEX HEX HEX HEX;
fragment HEX : [0-9a-fA-F];
WS : [ \t\r\n]+ -> skip;
```

Listing 2.6: Specification of the NGSPipes Full DSL grammar.

Examples

A pipeline used on epidemiological surveillance

In this section we present a pipeline used on epidemiological surveillance. The aim is to characterize bacterial strains through allelic profiles . When sequencing a bacterial strain by paired end methods with desired depth of coverage of 100x (in average each position in the genome will be covered by 100 reads), the output from the sequencer will be two FASTQ files containing the reads. Each read typically will have 90-250 nucleotides length, using Illumina technology. The first data processing step is to trim the reads for removing the adapters used in the sequencing process and any tags used to identify the experiment in a run.

In de novo assembly, software such as Velvet is used to obtain a draft genome composed of contigs, longer DNA sequences resulting from assembling multiple reads. The draft genome can be compared to databases of gene alleles for multiple loci using BLAST. Given BLAST results we can create an allelic profile characterizing the strain.

```
Pipeline "Github" "https://github.com/ngspipes/tools" {
  tool "Trimmomatic" "DockerConfig" {
    command "trimmomatic" {
      argument "mode" "SE"
      argument "quality" "-phred33"
      argument "inputFile" "ERR406040.fastq"
      argument "outputFile" "ERR406040.filtered.fastq"
      argument "fastaWithAdaptersEtc" "adapters/TruSeq3-SE.fa"
      argument "seed mismatches" "2"
      argument "palindrome clip threshold" "30"
      argument "simple clip threshold" "10"
      argument "windowSize" "4"
      argument "requiredQuality" "15"
      argument "leading quality" "3"
      argument "trailing quality" "3"
      argument "minlen length" "36"
    }
  }
  tool "Velvet" "DockerConfig" {
    command "velveth" {
      argument "output_directory" "velvetdir"
      argument "hash_length" "21"
      argument "file_format" "-fastq"
      chain "filename" "outputFile"
    }
    command "velvetg" {
      argument "output_directory" "velvetdir"
      argument "-cov_cutoff" "5"
    }
  }
  tool "Blast" "DockerConfig" {
    command "makeblastdb" {
      argument "-dbtype" "prot"
      argument "-out" "allrefs"
    }
  }
}
```

```
    argument "-title" "allrefs"
    argument "-in" "allrefs.fna.pro"
  }
  command "blastx" {
    chain "-db" "-out"
    chain "-query" "Velvet" "velvetg" "contigs_fa"
    argument "-out" "blast.out"
  }
}
```

Example 2.6: A pipeline used on epidemiological surveillance.

A visual representation of this pipeline described in Example 2.6 is presented in the Figure 2.1. Moreover, in this figure is also possible to observe other execution orders that are feasible to execute this pipeline in the engine for workstation.

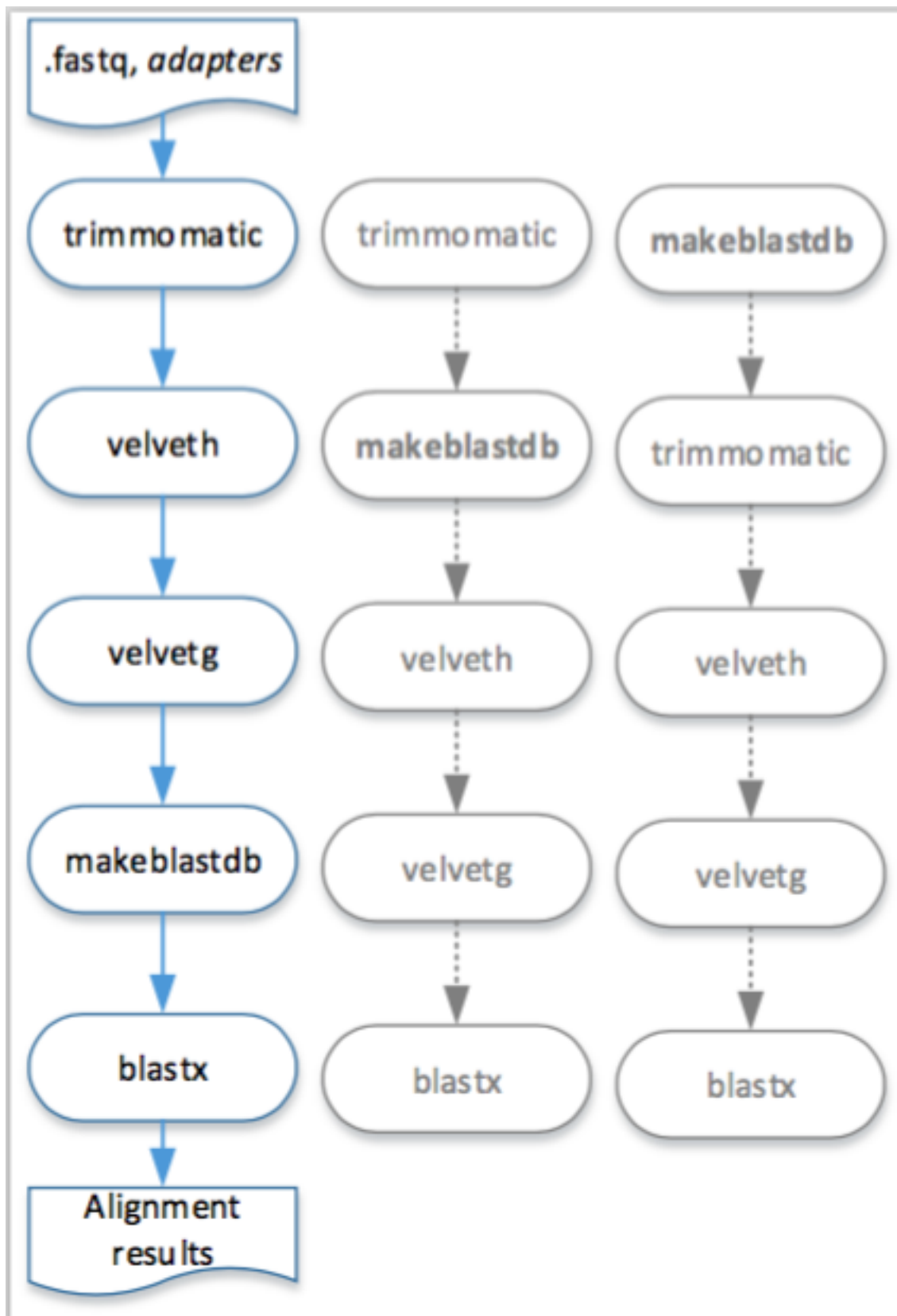


Figure 2.1: Visual representation of the execution, in the engine for workstation, of the pipeline described in

Example 2.6.

In the engine for cloud, different steps of the pipeline can be executed in different machines, it is only necessary to respect its dependencies, as it is shown in the Figure 2.2.

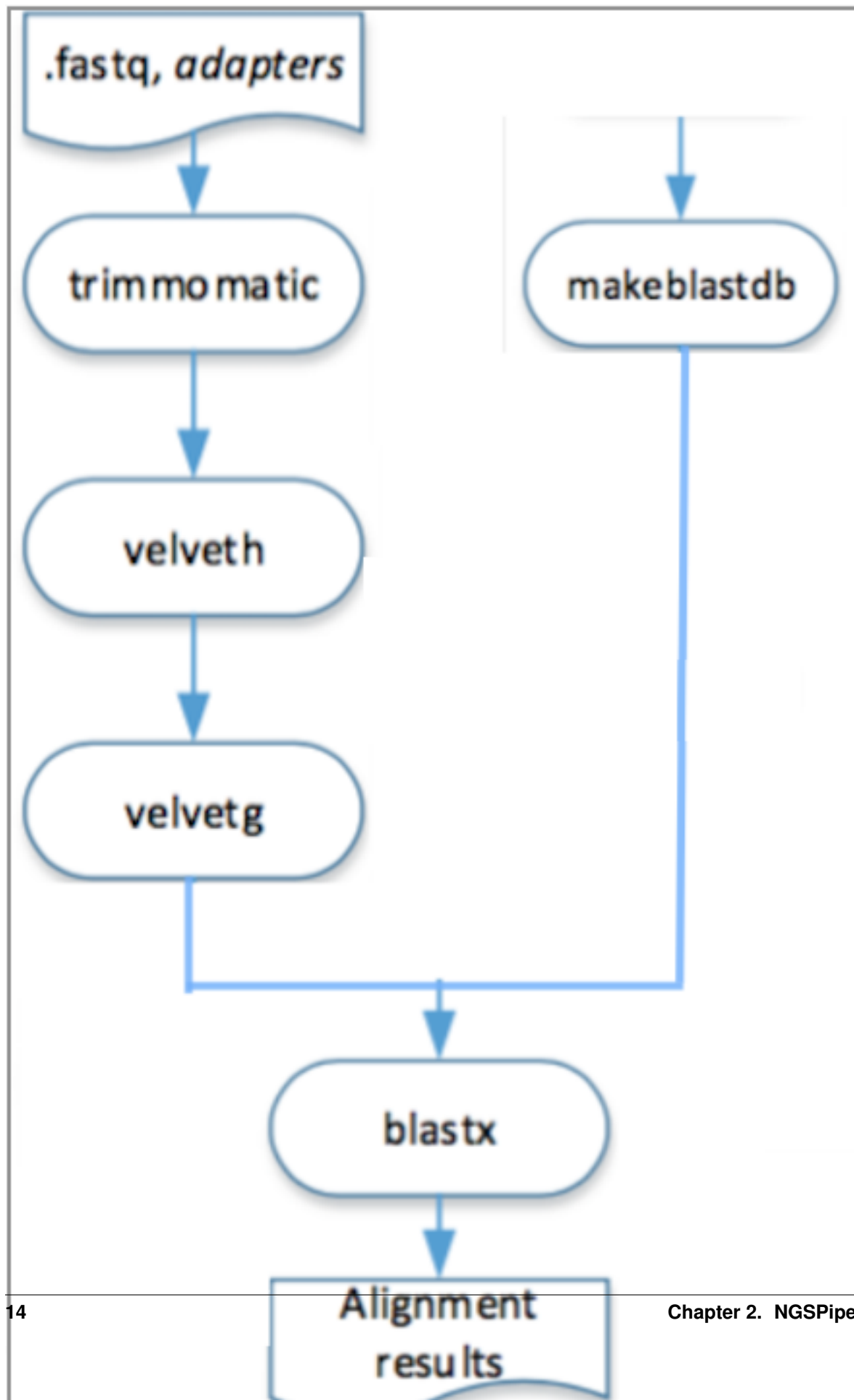


Figure 2.2: Visual representation of the execution, in the engine for cloud, of the pipeline described in Example 2.6.

A pipeline used on ChIP-Seq analysis

In this section we present a pipeline used on ChIP-Seq analysis. This pipeline includes mapping with bowtie2, converting the output to bam format, sorting the bam file, creating a bam index file, running flagstat command, and removing duplicates with picard. So, this pipeline can be used in a ChIP-Seq pipeline that uses the resulting bam file for peak calling and creating heatmaps. Since those steps are generic that can be used for ATAC-Seq analysis too.

```
Pipeline "Github" "https://github.com/ngspipes/tools" {
  tool "Bowtie2" "DockerConfig" {
    command "bowtie2-build" {
      argument "reference_in" "sequence.fasta"
      argument "bt2_base" "sequence"
    }
  }
  tool "Bowtie2" "DockerConfig" {
    command "bowtie2" {
      argument "-U" "SRR386886.fastq"
      argument "-x" "sequence"
      argument "--trim3" "1"
      argument "-S" "eg2.sam"
    }
  }
  tool "SAMTools" "DockerConfig" {
    command "view" {
      argument "-b" "-b"
      argument "-o" "eg2.bam"
      chain "input" "-S"
    }
  }
  tool "SAMTools" "DockerConfig" {
    command "sort" {
      argument "-o" "eg2.sorted.bam"
      chain "input" "-o"
    }
  }
  tool "Picard" "DockerConfig" {
    command "MarkDuplicates" {
      chain "INPUT" "-o"
      argument "OUTPUT" "marked_duplicates.bam"
      argument "REMOVE_DUPLICATES" "true"
      argument "METRICS_FILE" "metrics.txt"
    }
  }
}
```

Example 2.7: A pipeline used on ChIP-Seq analysis.

A visual representation of this pipeline is presented in the next figure.

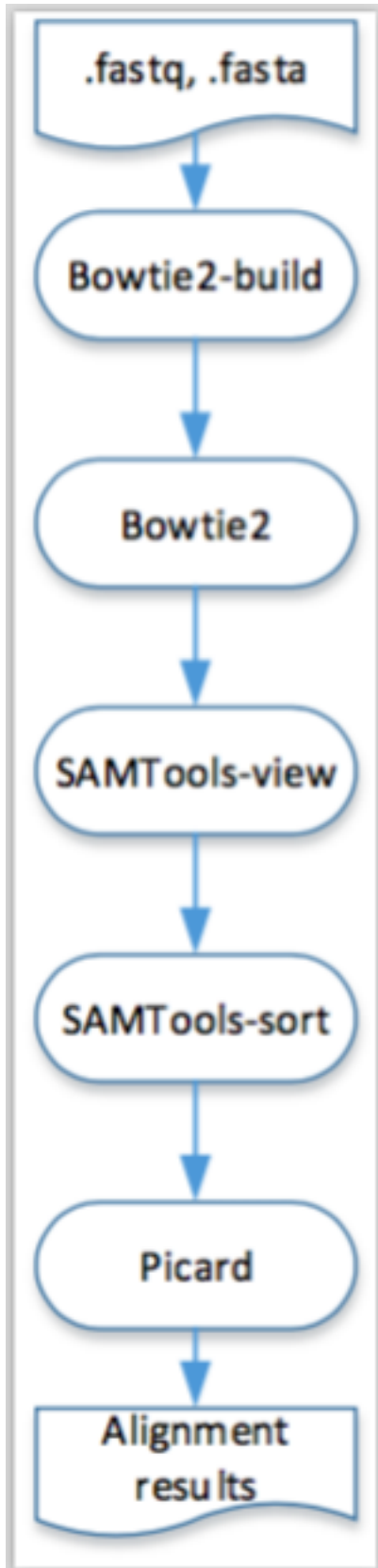


Figure 2.3: Visual representation of the execution, in both engines, of the pipeline described in Example 2.6.

A pipeline using listing tools (for executing only with Engine for Cloud)

A specific use of NGS data in public health is the determination of the relationship between samples potentially associated with a foodborne pathogen outbreak. This relationship can be determined from the phylogenetic analysis of a DNA sequence alignment containing only variable positions, which we refer to as a SNP matrix. The applications of such a matrix include inferring a phylogeny for systematic studies and determining within traceback investigations whether a clinical sample is significantly different from environmental/product samples.

This case study is a pipeline which combines all the steps necessary to construct a reference-based SNP matrix from an NGS sample data set. The pipeline starts with the mapping of NGS reads to a reference genome using Bowtie2, then it continues with the processing of those mapping (BAM) files using SAMtools, identification of variant sites using VarScan3, and ends with the production of a SNP matrix using custom Python scripts (calling of SNPs at each variant site, combining the SNPs into a SNP matrix). The Python scripts are reused from the *CFSAN SNP Pipeline: an automated method for constructing SNP matrices from next-generation sequence data*. *PeerJ Computer Science 1:e20* <https://doi.org/10.7717/peerj-cs.20>. As it can be observed in this data set, there are four samples, whose dataflow process is more detailed in the [documentation page](#) of this pipeline.

```
Pipeline "Github" "https://github.com/Vacalexix/tools" {
  tool "snp-pipeline" "DockerConfig" {
    command "create_sample_dirs" {
      argument "-d" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪samples/*"
      argument "--output" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/sampleDirectories.txt"
    }
  }

  tool "Bowtie2" "DockerConfig" {
    command "bowtie2-build" {
      argument "reference_in" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/reference/lambda_virus.fasta"
      argument "bt2_base" "reference"
    }
    command "bowtie2" {
      argument "-p" "1"
      argument "-q" "-q"
      argument "-x" "reference"
      argument "-1" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪samples/sample1/sample1_1.fastq"
      argument "-2" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪samples/sample1/sample1_2.fastq"
      argument "-S" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪reads1.sam"
    }
    command "bowtie2" {
      argument "-p" "1"
      argument "-q" "-q"
      argument "-x" "reference"
      argument "-1" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪samples/sample2/sample2_1.fastq"
      argument "-2" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪samples/sample2/sample2_2.fastq"
      argument "-S" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪reads2.sam"
    }
    command "bowtie2" {
      argument "-p" "1"
      argument "-q" "-q"
```

```

        argument "-x" "reference"
        argument "-1" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪samples/sample3/sample3_1.fastq"
        argument "-2" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪samples/sample3/sample3_2.fastq"
        argument "-S" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪reads3.sam"
    }
    command "bowtie2" {
        argument "-p" "1"
        argument "-q" "-q"
        argument "-x" "reference"
        argument "-1" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪samples/sample4/sample4_1.fastq"
        argument "-2" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪samples/sample4/sample4_2.fastq"
        argument "-S" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪reads4.sam"
    }
}
tool "Listing" "DockerConfig" {
    command "startListing" {
        argument "referenceName" "reads.sam"
        argument "filesList" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/reads1.sam snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪reads2.sam snp-pipeline-master/snppipeline/data/lambdaVirusInputs/reads3.sam snp-
↪pipeline-master/snppipeline/data/lambdaVirusInputs/reads4.sam"
    }
}
tool "Samtools" "DockerConfig" {
    command "view" {
        argument "-b" "-b"
        argument "-S" "-S"
        argument "-F" "4"
        argument "-o" "reads.unsorted.bam"
        argument "input" "reads.sam"
    }
    command "sort" {
        argument "-o" "reads.sorted.bam"
        argument "input" "reads.unsorted.bam"
    }
    command "mpileup" {
        argument "--fasta-ref" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/reference/lambda_virus.fasta"
        argument "input" "reads.sorted.bam"
        argument "--output" "reads.pileup"
    }
}
tool "VarScan" "DockerConfig" {
    command "mpileup2snp" {
        argument "mpileupFile" "reads.pileup"
        argument "--min-var-freq" "0.90"
        argument "--output-vcf" "1"
        argument "output" "var.flt.vcf"
    }
}

```

```

tool "Listing" "DockerConfig" {
  command "stopListing" {
    argument "referenceName" "var.flt.vcf"
    argument "destinationFiles" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/samples/sample1/var.flt.vcf snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/samples/sample2/var.flt.vcf snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/samples/sample3/var.flt.vcf snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/samples/sample4/var.flt.vcf"
  }
}

tool "snp-pipeline" "DockerConfig" {
  command "create_snp_list" {
    argument "--vcfname" "var.flt.vcf"
    argument "--output" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/snplist.txt"
    argument "sampleDirsFile" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/sampleDirectories.txt"
  }
}

tool "Listing" "DockerConfig" {
  command "restartListing" {
    argument "referenceName" "reads.pileup"
  }
}

tool "snp-pipeline" "DockerConfig" {
  command "call_consensus" {
    argument "--snpListFile" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/snplist.txt"
    argument "--output" "consensus.fasta"
    argument "--vcfFileName" "consensus.vcf"
    argument "allPileupFile" "reads.pileup"
  }
}

tool "Listing" "DockerConfig" {
  command "stopListing" {
    argument "referenceName" "consensus.fasta"
    argument "destinationFiles" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/samples/sample1/consensus.fasta snp-pipeline-master/snppipeline/
↪data/lambdaVirusInputs/samples/sample2/consensus.fasta snp-pipeline-master/
↪snppipeline/data/lambdaVirusInputs/samples/sample3/consensus.fasta snp-pipeline-
↪master/snppipeline/data/lambdaVirusInputs/samples/sample4/consensus.fasta"
  }
}

tool "snp-pipeline" "DockerConfig" {
  command "create_snp_matrix" {
    argument "sampleDirsFile" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/sampleDirectories.txt"
    argument "--consFileName" "consensus.fasta"
    argument "--output" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/snpma.fasta"
  }
}
}

```

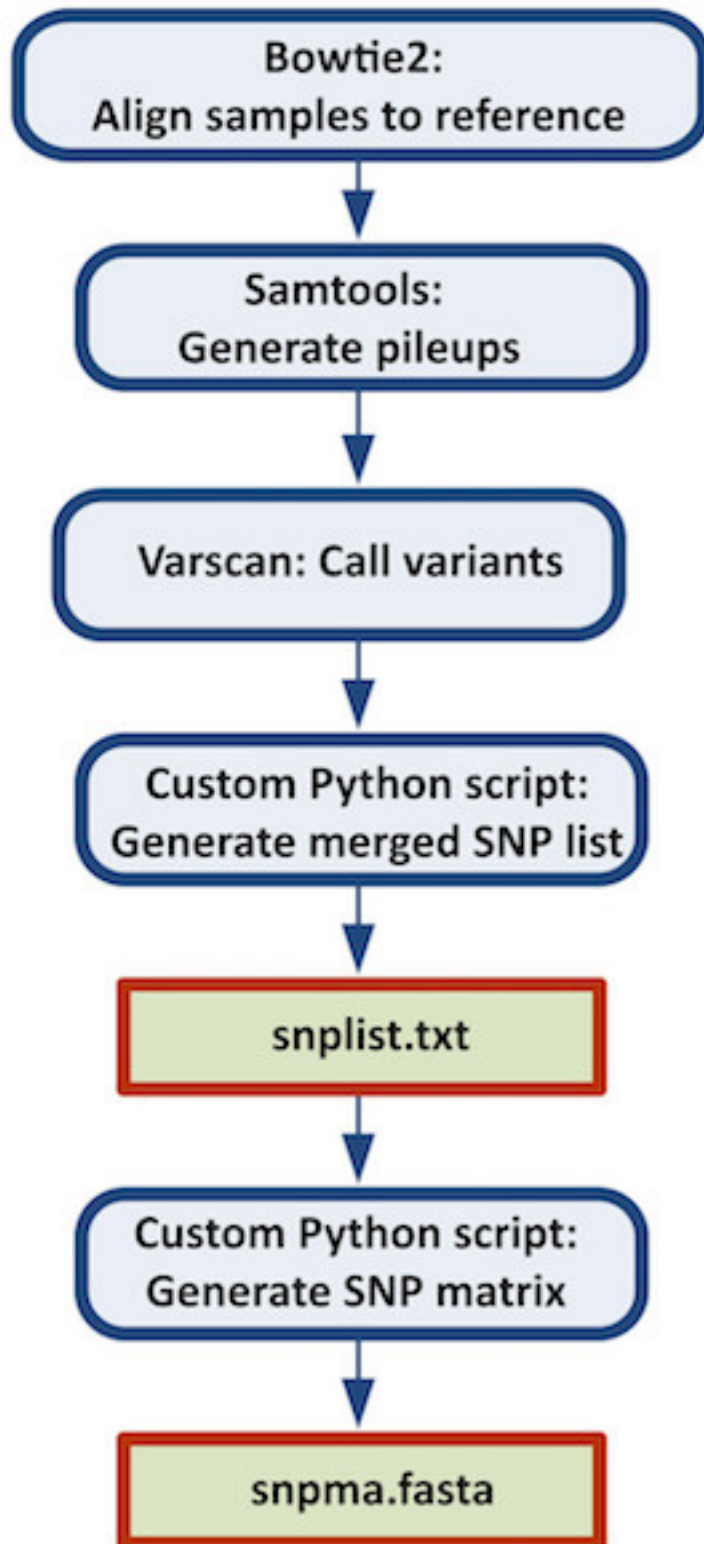


Figure 2.4: Figure from Davis S, Pettengill JB, Luo Y, Payne J, Shpuntoff A, Rand H, Strain E. (2015) CFSAN SNP Pipeline: an automated method for constructing SNP matrices from next-generation sequence data. *PeerJ Computer Science* 1:e20 <https://doi.org/10.7717/peerj-cs.20>

NGSPipes repository

The *NGSPipes repository* is a component of NGSPipes system that contains all the information related to the available tools which can be used when defining a pipeline. We provide a repository prototype that contains some tools to test our system, which can be found in <https://github.com/ngspipes/tools>. User made repositories can be used, as it will be explained in this section. This component has to supply the following information:

- a list of *tool names*;
- a list of *tool descriptors*;
- a list of *tool logotypes* (optional);
- a list of *configurators* of a given tool;
- a list of the names of the configurators available for a given tool.

For defining the *tool descriptors* and *configurators*, we have defined JSON schemas, as well as for specify all the tools that are available in the repository.

Tool names

The repository is composed by a list of tools. All the tools names that are available in a given repository, are described in a file with a JSON format designed by `Tools.json`. In NGSPipes repository example this file appears at the root of the repository (please, see the [tool's repository](#)). Moreover, the presented repository structure is one of the possible structures that is supported by the repository support library used in the NGSPipes framework. The format of the `Tools.json` file is given by the JSON schema presented in Listing 3.1.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "toolsName": {
      "type": "array",
      "items": { "type": "string" }
    },
  },
}
```

```
    "required": [ "toolsName" ]
  }
}
```

Listing 3.1: JSON schema for specifying the names of the tools included in the repository.

Tool descriptors

To each available tool in our framework, we have a *tool descriptor*, i.e., a JSON file responsible for supplying all the information needed about the tool, such as the memory needed to execute it, the commands and the arguments of each command. The format of this file is given by the JSON schema presented in Listing 3.2.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "author": { "type": "string" },
    "version": { "type": "string" },
    "description": { "type": "string" },
    "documentation": {
      "type": "array",
      "items": { "type": "string" }
    },
    "setup": {
      "type": "array",
      "items": { "type": "string" }
    },
    "toolType": {
      "type": "string",
      "enum": ["Unit", "splitting", "joinning", "listing"]
    },
    "requiredMemory": { "type": "integer" },
    "recommendedCpus": { "type": "integer" },
    "recommendedDiskSpace": { "type": "integer" },
    "commands": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": { "type": "string" },
          "command": { "type": "string" },
          "description": { "type": "string" },
          "priority": { "type": "integer" },
          "argumentsComposer": { "type": "string" },
          "arguments": {
            "type": "array",
            "items": {
              "type": "object",
              "properties": {
                "name": { "type": "string" },
                "argumentType": { "type": "string",
                  "enum": ["int", "file", "string", "double", "directory"] },
                "isRequired": { "type": { "enum": ["true", "false"] } },
                "description": { "type": "string" }
              }
            }
          }
        }
      }
    }
  }
}
```

```

        "required": ["name", "argumentType", "isRequired", "description"]
    },
    },
    "outputs": {
        "type": "array",
        "items": {
            "type": "object",
            "properties": {
                "name": { "type": "string" },
                "description": { "type": "string" },
                "outputType": { "type": {
                    "enum": ["directory_dependent", "file_dependent", "independent
↪"] } } },
                "argument_name": { "type": "string" },
                "value": { "type": "string" }
            },
            "required": ["name", "description", "outputType", "argument_name",
↪ "value"]
        }
    },
    "inputs": {
        "type": "array",
        "items": {
            "type": "object",
            "properties": {
                "name": { "type": "string" },
                "description": { "type": "string" },
                "inputType": { "type": "string",
                    "enum": [ "directory_dependent", "file_dependent",
↪ "independent" ] } },
                "value": { "type": "string" }
            },
            "required": [ "name", "description", "inputType", "argument_name",
↪ "value" ] } } },
        "required": ["name", "command", "description", "priority",
            "argumentsComposer", "arguments", "outputs", "inputs"]
    },
    },
    "required": ["name", "author", "version", "description",
        "documentation", "setup", "tool type",
        "requiredMemory", "recommendedDiskSpace",
        "recommendedCpus", "commands"]
}

```

Listing 3.2: JSON schema for specifying each tool included in the repository.

As an example, please see the tools descriptors that we have included in our tools' repository example, such as the [Velvet descriptor](#) and the [Trimmomatic descriptor](#) for Velvet and Trimmomatic tools, respectively. In our repository support library, each tool descriptor must be defined in a file named as `Descriptor.json`.

As defined on the previous JSON schema, a tool description must include its *name*, *author*, *version*, *description*, *documentation*, *setup*, *toolType*, *required memory*, *recommendedCpus*, *recommendedDiskSpace* and *commands*. The *version* property describes the version of the executable that is being considered by this descriptor. The *documentation* property allows to add a collection of links that contains documentation about the tool. The *setup* property contains all the **scripts** that must be executed before executing any command within the tool. For instance, for executing the Trimmomatic command, it must be previously installed the Java Runtime Environment. Thus, in the Trimmomatic descriptor, we include the setup presented in Example 3.1.

```
"setup" : [ "apt-get install -y default-jre" ]
```

Example 3.1: Trimmomatic command setup.

Command descriptions

commands is an array of JSON objects that describes each command within a tool. For instance, the Trimmomatic tool has only one command, but the Velvet tool has two commands, namely, *velvetg* and *velveth*. For each *command* in the array *commands* it must exist its *name*, the *command* itself, its *description*, its *priority*, its *arguments*, the *argumentComposer* and its *outputs*. The *priority* of each command within a tool is important for defining execution dependency among commands within the same tool. For instance, in the Velvet tool, although not explicitly defined as an argument, *velveth* uses files produced by *velvetg*. If the files are already produced, then it is not necessary to execute *velvetg* if data is the same. However, if data differs from the last execution or is not yet produced, it must be assured that *velvetg* is executed before *velveth*. Therefore, we have added the *priority* property to each command to assign an integer that reflect the execution order within commands of the same tool which do not have it explicitly, but which is needed. The *argumentsComposer* item is the responsible for knowing how to concatenate the arguments, namely if arguments are passed as *argName=argValue* OR *argName:argValue* OR *argName-argValue*. The many *argumentComposer* types supported by the NGSPipes repository support library are detailed in sub-section *ArgumentsComposer*. The *arguments* and the *outputs* are both arrays of JSON objects.

Argument descriptions

arguments is an array of JSON objects that describes each argument of a specific command. For each argument is required to define its *name*, its *argumentType*, if it is *isRequired* and its *description*. The type of each argument must be one of the following: integer number (*int*); file (*file*); text (*string*); real number (*double*) or a directory (*directory*). The *isRequired* property, which can be defined as *true* or *false*, indicates if is necessary to set a value to this argument or is an optional argument. As an example, consider the *trimmomatic* tool, which only has a command. For SINGLE END data, one output and input file are specified. Therefore, it is necessary to add to its descriptor the information specified in Example 3.2.

```
{
  "name" : "outputFile",
  "argumentType" : "file",
  "isRequired" : "false",
  "description" : "Specifies the name of output file."
},
{
  "name" : "inputFile",
  "argumentType" : "file",
  "isRequired" : "false",
  "description" : "Specifies the path to the fastq input file."
},
```

Example 3.2: arguments for Trimmomatic command

Both of the previous examples have the *isRequired* property set to *false* since for non SINGLE END data, trimmomatic execution uses pairs of input and output files, which are described in the tool descriptor by other arguments.

Output descriptions

output is an array of JSON objects that describes the outputs of each command. For each output is required to define its *name*, *outputType*, *description*, *argument_name* and *value*. Notice that the name passed as an argument to a command is not the name that is necessary to specify as a JSON property of the *output* JSON object. The

name property refers to the name of the JSON object, not to the name of the file that is produced by the execution of a given command. Depending on the command, the name of the file that is produced by a given command can be set as an argument by the user or be an internal decision of the executing command. Therefore, the `independent` `outputType` is used when an output value is specified inside of command and isn't affected by any argument. In this case, the `value` property of the JSON `output` object is set with the name that is internally generated by the corresponding command. An example of the output in descriptor file is depicted in Example 3.3.

```
{
  "name" : "output",
  "description" : "",
  "outputType" : "independent",
  "argument_name" : "",
  "value" : "output.txt"
}
```

Example 3.3: Example of an output descriptor.

In the previous case, the `argument_name` is the empty string since there is no corresponding argument defined in the tool descriptor to set the name of the produced output file.

The `outputType` can also be `file_dependent` or `directory_dependent`. An `outputType` is `file_dependent` if its value is specified in an argument and there is no specific directory that is created for keeping the generated output file. As an example, and taking into account the previous example of `trimmomatic` for SINGLE END data, the output is described in the tool description as presented in Example 3.4.

```
{
  "name" : "outputFile",
  "description" : "",
  "outputType" : "file_dependent",
  "value" : "",
  "argument_name" : "outputFile"
},
```

Example 3.4: Example of an output descriptor.

In this case, the `value` property is set to the empty string since the name of the output file is specified by the user. Moreover, the `argument_name` property defines the name of the JSON object that corresponds to the JSON object that defines the argument used for the specified the output file name.

The other type of output is `directory_dependent`, which is used when an output value is added to a specified directory that is generated within the command execution. In this case, the name of the directory is passed as an argument, but the name of the produced files are not passed as arguments. Instead, they are generated internally, within execution. As an example, consider the `velvet` tool, where the commands outputs are of this type because they will be written to a directory, the first argument of `velvetg` and `velveth`, when executing both commands. Therefore, since the output directory is a command argument, we have to specify in the tool descriptor a corresponding argument description, such as the one depicted in Example 3.5.

```
{
  "name" : "output_directory",
  "outputType" : "directory",
  "isRequired" : "true",
  "description" : "Directory where will be output files"
}
```

Example 3.5: Argument description in the case of a directory type

And thus, Example 3.6 illustrates of the output descriptor in the descriptor file, corresponding to the previous argument will be like:

```
{
  "name" : "stats",
  "description" : "",
  "argumentType" : "directory_dependent",
  "argument_name" : "output_directory",
  "value" : "stats.txt"
}
```

Example 3.6: Output description when is dependent of an argument with type directory

Notice that the file name `stats.txt` is not passed as an argument to `velveth` nor to `velvetg`. Instead, it is generated internally and is stored in the output directory whose name was passed as an argument.

Input descriptions

input is an array of JSON objects that describes the inputs of each command. They are similar to Output descriptions. They help inferring dependencies between pipeline tasks.

ArgumentsComposer

In this subsection is listed all the argumentsComposer that are already included in our repository support library. The existing argumentsComposer are (name of the argumentComposer-> [corresponding format]):

1. dummy -> []
2. valuesSeparatedBySpace -> [value value value]
3. nameValuesSeparatedByEqual -> [name=value name=value name=value]
4. nameValuesSeparatedByColon -> [name:value name:value name:value]
5. nameValuesSeparatedByHyphen -> [name-value name-value name-value]
6. nameValuesSeparatedBySpace -> [name value name value name value]
7. valuesSeparatedByColon -> [value:value:value]
8. valuesSeparatedByVerticalBar -> [value|value|value]
9. valuesSeparatedByHyphen -> [value-value-value]
10. valuesSeparatedBySlash -> [value/value/value]
11. valuesSeparatedByComma -> [value,value,value]
12. trimmomatic -> [TRIMMOMATIC STYLE ArgCategory:arg:arg:arg]
13. velvetG -> [VELVETG STYLE all arguments has format [name value] except output_directory that has format [value]]

Listing 3.3: Some argumentsComposer included in the repository support library.

Examples of the mapping of the arguments and output descriptions to command parameters.

As we can see in the [Velvet tool manual](#), a simple execution of the `velvetg` command in the command line (without the NGSPipes System) after producing the executable with the `make` command is described in Example 3.7.

```
./velvetg velvetDir -cov_cutoff 5
```

Example 3.7: Executing velvetg command on the command line.

Therefore, the description of `velvetg` command, within the descriptor of `velvet` tool, must include two arguments descriptions, namely, one for the directory argument and other for the option `_cov_cutoff`. As we can observe in `velvet` descriptor file (<https://github.com/ngspipes/tools/blob/master/Velvet/Descriptor.json>), the JSON object for defining the arguments of `velvetg` command starts with the definitions depicted in Example 3.8.

```
{
  "arguments" : [
    {
      "name" : "output_directory",
      "argumentType" : "directory",
      "isRequired" : "true",
      "description" : "Directory where will be output files"
    },
    {
      "name" : "-cov_cutoff",
      "argumentType" : "float",
      "isRequired" : "false",
      "description" : "remove coverage nodes
                      AFTER tour bus or allow the system to infer it (default no removal)"
    }
  ],
}
```

Example 3.8: Some velvetg arguments descriptions.

And, since the output directory produces output files, the produced output is `directory_dependent` as we can see in section “Output descriptions” within this section, the JSON object for defining the outputs of `velvetg` command starts with the descriptions depicted in Example 3.9.

```
"outputs" : [
  {
    "name" : "stats",
    "description" : "",
    "outputType" : "directory_dependent",
    "argument_name" : "output_directory",
    "value" : "stats.txt"
  },
  {
    "name" : "preGraph",
    "description" : "",
    "outputType" : "directory_dependent",
    "argument_name" : "output_directory",
    "value" : "PreGraph"
  }
],
```

Example 3.9: Some output descriptions for velvetg.

The values of these arguments (`velvetDir` and `5`, respectively) will be set in the pipeline specification. For more information about the pipeline specification, please consult (<https://github.com/ngspipes/dsl/wiki>).

Another example referred in this documentation is the `Trimmomatic` tool. As we can see in the [Trimmomatic manual](#), For single-ended data, one input and one output file are specified. The required processing steps (trimming, cropping, adapter clipping etc.) are specified as additional arguments after the input/output files. Thus, it appears in description presented in Example 3.10 how to execute this command.

```
java -jar <path to trimmomatic jar> SE
      [-threads <threads>] [-phred33 | -phred64] [-trimlog <logFile>]
      <input> <output> <step 1> ...
```

Example 3.10: Executing Trimmomatic in the command line for single-ended data.

For paired-end data, two input files, and 4 output files are specified, 2 for the ‘paired’ output where both reads survived the processing, and 2 for corresponding ‘unpaired’ output where a read survived, but the partner read did not. Thus, it appears in the description presented in Examl 3.11 how to executed this command in this version.

```
java -jar <path to trimmomatic.jar> PE
    [-threads <threads>] [-phred33 | -phred64] [-trimlog <logFile>] >]
    [-basein <inputBase> | <input 1> <input 2>]
    [-baseout <outputBase> | <unpaired output 1>
    <paired output 2> <unpaired output 2> <step 1> ...
```

Example 3.11: Executing Trimmomatic in the command line for paired-ended data.

Thus, considering the SINGLE END DATA, a possible execution in the command line could be like the following

```
java -jar local/trimmomatic/trimmomatic-0.33.jar SE -phred33 ERR406040.fastq
ERR406040.filtered.fastq ILLUMINACLIP:local/trimmomatic/adapters/TruSeq3-SE.fa:2:30:10
LEADING:3 TRAILING:3 SLIDINGWINDOW:4:15 MINLEN:36
```

****Example 3.12: ****

In this case the input file is ERR406040.fastq and the output file is ERR406040.filtered.fastq. Thus, in the Trimmomatic tool description, we have included as arguments descriptions the ones described in Example 3.13.

```
{
  "name" : "inputFile",
  "argumentType" : "file",
  "isRequired" : "false",
  "description" : "Specifies the path to the fastq input file."
},
{
  "name" : "outputFile",
  "argumentType" : "file",
  "isRequired" : "false",
  "description" : "Specifies the name of output file."
},
{
  "name" : "paired input 1",
  "argumentType" : "file",
  "isRequired" : "false",
  "description" : "Specifies the path to the input file 1 of paired mode."
},
{
  "name" : "paired input 2",
  "argumentType" : "file",
  "isRequired" : "false",
  "description" : "Specifies the path to the input file 2 of paired mode."
},
}
```

Example 3.13: Trimmomatic arguments.

In the case of Trimmomatic command (please notice that Trimmomatic tool has only one command, with the same name), since both arguments inputFile and outputFile are only required in the SINGLE END data, their property isRequired was set to false.

With respect to the outputs, the Trimmomatic command description has the outputs described as in Example 3.14.

```
{
  "name" : "outputFile",
```



```

    "description" : "",
    "argumentType" : "file_dependent",
    "value" : "",
    "argument_name" : "outputFile"
  },
  {
    "name" : "paired output 1",
    "description" : "",
    "argumentType" : "file_dependent",
    "value" : "",
    "argument_name" : "paired output 1"
  },
  {
    "name" : "unpaired output 1",
    "description" : "",
    "argumentType" : "file_dependent",
    "value" : "",
    "argument_name" : "unpaired output 1"
  },
  {
    "name" : "paired output 2",
    "description" : "",
    "argumentType" : "file_dependent",
    "value" : "",
    "argument_name" : "paired output 2"
  },
  {
    "name" : "unpaired output 2",
    "description" : "",
    "argumentType" : "file_dependent",
    "value" : "",
    "argument_name" : "unpaired output 2"
  }
}

```

Example 3.14: Outputs description of Trimmomatic command.

As mentioned before, in the arguments and outputs descriptions, the values to be set to the arguments are done in the pipeline specification, as can be seen in the example in <https://github.com/ngspipes/dsl/wiki>. Notice that the Trimmomatic outputs are all `file_dependent` which means that its value is also an argument and thus is set by the user in the pipeline specification.

Tool configurators

The repository must also include at least one configurator for each tool. A *tool configurator* is responsible for adding all the information needed to define the execution context for executing the tool and its respective commands. Each *tool configurator* for each tool is given as a JSON file. Thus, for knowing all the available configurators for a specific tool, it exists, for each tool, a JSON file that lists all the JSON files that correspond to tool configurators for that tool. In our repository example and thus in our support implementation, these files appear at the root of each tool directory (please, see the [tool directory example](#)). For instance, in `Blast` tool, it can be observed that there is only a tool configurator (<https://github.com/ngspipes/tools/blob/master/Blast/configurators.json>), and the same is given as (<https://github.com/ngspipes/tools/blob/master/Blast/DockerConfig.json>).

List of configurators of a tool

For each tool, the list of the tool configurators that are available in a given repository are described in a JSON format in a file designed by `Configurators.json`. The format of the file that lists all the tool configurators files is given by JSON schema defined in Listing 3.4.

```
"$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "configuratorsFileName": {
      "type": "array",
      "items": { "type": "string" }
    },
    "required": [ "configuratorsFileName" ]
  }
}
```

Listing 3.4: JSON schema for declaring the filenames of the configurators for a given tool.

Tool Configurators

As depicted in the previous schema, the file `Configurators.json` includes all the name of the files that corresponds to possible configurators for a given tool. Thus, for each file name included in `onfigurators.json` it exists a corresponding JSON file with the specific configuration. In our repository example and thus in our support implementation, the files for each specific configuration appears at the root of each tool directory (please, see the a [tool directory example](#)). The format of this file is given by the JSON schema defined in Listing 3.5.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "builder": { "type": "string" },
    "uri": { "type": "string" },
    "setup": {
      "type": "array",
      "items": { "type": "string" }
    }
  },
  "required": [ "name", "uri", "setup" ]
}
```

Listing 3.5: JSON schema for declarung each tool configurator.

Thus, a tool configuration is a JSON file with the following information: `name` of the file where is defined the execution context execution context (ex: `DockerConfig`); `builder` name of the execution context (ex: `Docker`); `setup`, i.e., the scripts that are necessary to execute to assure the existence of the execution context; and the `uri` where the tool is. Next example describes that the tool is on a docker image and thus is necessary to install docker in the execution context.

```
{
  "name" : "DockerConfig",

  "builder": "Docker"

  "uri" : "simonalpha/ncbi-blast-docker",
}
```

```

"setup" : [
  "wget -qO- https://get.docker.com/ | sh"
]
}

```

Example 3.15: Example of a tool configurator for the blast tool.

Defining your own tool repository

Each user can define its own tool repository, locally or remotely and use NGSPipes support library. The simplest way to do this is to use an hierarchical directory system approach, either locally or remotely. For a different form of structuring data, it would probably be necessary to extend NGSPipes support library.

Using an hierarchical directory system approach

In this section it will be described how to use an hierarchical directory system approach for defining a new tool repository, locally and remotely. For the remote case, we will use github as an example. In both cases, it is necessary to create a directory for each tool. The directory name will be seen as the tool name (the tool identifier on the repository) and is exactly the same name that is used in the pipeline definition and in the file `Tools.json`. In the file `Tools.json` there will be a tool name for each available tool in the repository.

Each tool directory keeps all the information about that tool, namely its description, its logotype, its configurators and the file name of its configurators. As mentioned before, the tool descriptor, which includes all the metadata needed to describe a tool, is given in a JSON file. As a convention, each tool descriptor file name is `Descriptor.json`. With respect to the logo file it should be a png file named as `Logo.png`. The logo file is optional. The file where is kept the file name of the configurators for a tool is also given as a JSON file, always designed as `Configurators.json`. For each file name specified in this file, there must exist the respective configurator JSON file.

Define a new repository locally

For defining a new repository in our own computer we have first to create a directory that will be our tool repository (ex: named as `tools`). Then, add to `tools` directory the file `Tools.json` and for each tool name that appears in this file, which identifies a specific tool, create a new directory in `tools`. Each new created directory inside `tools` must have the corresponding name used in `Tools.json` to identify the tool. Each tool directory must contain the data described in the beginning of subsection “Defining your own tool repository”.

Define a new repository on github

After log-in in github, create a new repository (ex: named as `tools`). The endpoint of this new repository will be the tool repository. Then, after cloning your repository to your computer, it will appear a directory named as `tools`. Then, do the same steps of a section “Define a new repository locally” within this subsection. After that, synchronize the repository.

Tool Types

For supporting data partitioning in the engine for cloud, which will allow to executing in multiple machines partitions of data at the same time and thus increase process efficiency, the ‘tool type’ should be specified in a tool descriptor. Notice that this feature has only impact in the engine for cloud solution. There are four different tool types:

- data processing tools, *i.e.*, **unit**;
- listing tools, *i.e.*, **splitting**;
- splitting tools *i.e.*, **joining**;
- joining tools *i.e.*, **listing**.

Unlike unit processing tools, where each command is mapped into one task, a splitting command within a splitting tool generates one task corresponding to the splitting of the file plus N tasks per command, where N is the number of partitions of the file whereas each task processes a partition of the file. Data partitioning allows users to work with and process multiple files having to specify each command only once, while treating the files like a single file. It means that when users split a file in ten, for instance, they do not have to include the same tool ten times in the pipeline for every partition: the analyser will do that for them.

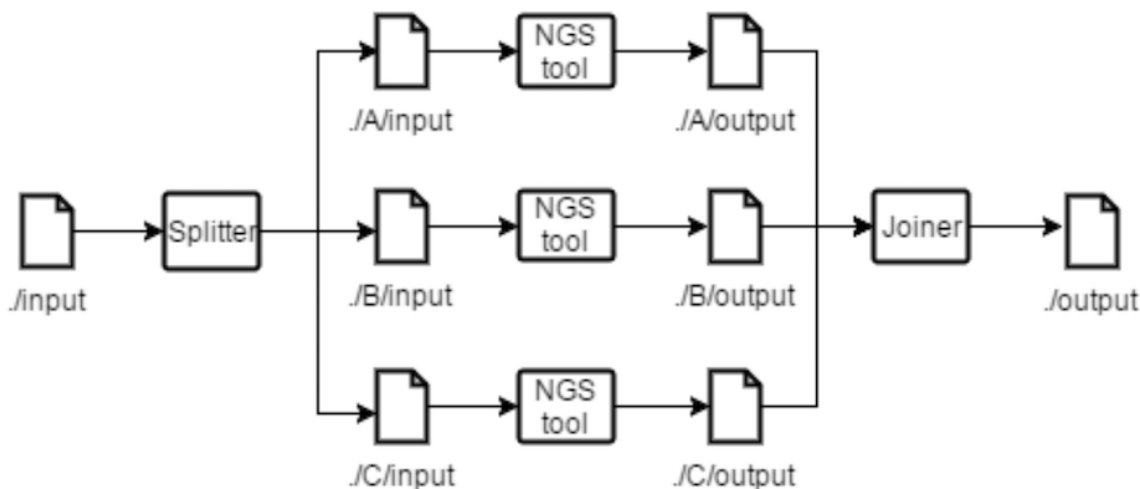


Figure 3.1 Splitting and Joining tools example.

Figure 3.1 shows how a file named input is split originating three different files with the same name, stored in directories with different names. For each partition the analyser will generate a directory where it stores the file partition with the same name it had before being partitioned. For every command specified in the pipeline description that uses the partitioned file, it is generated a task where the input path (partitioned file's path) is concatenated with the name of the directory where the partition of the file is stored. Multiple directories are created to avoid name collision between files generated. Joining tools generate a task to join the partitions with the name of the input, that are stored in analyser generated directories (through either splitting or listing) corresponding to that input. Commands whose input depend on the join output will no longer have their tasks multiplied per partition. In Figure 3.2 it is depicted an example where a user wants to process different files of the same type, using the same tools, without having to specify each command more than once. Listing tools move and rename the files to match the same pattern as the splitting tools. After files are listed, they can be treated as one, as if it had been a split. While the splitting tools are used to partition data files and apply the same command or set of commands to each partition, the listing tools allow users to

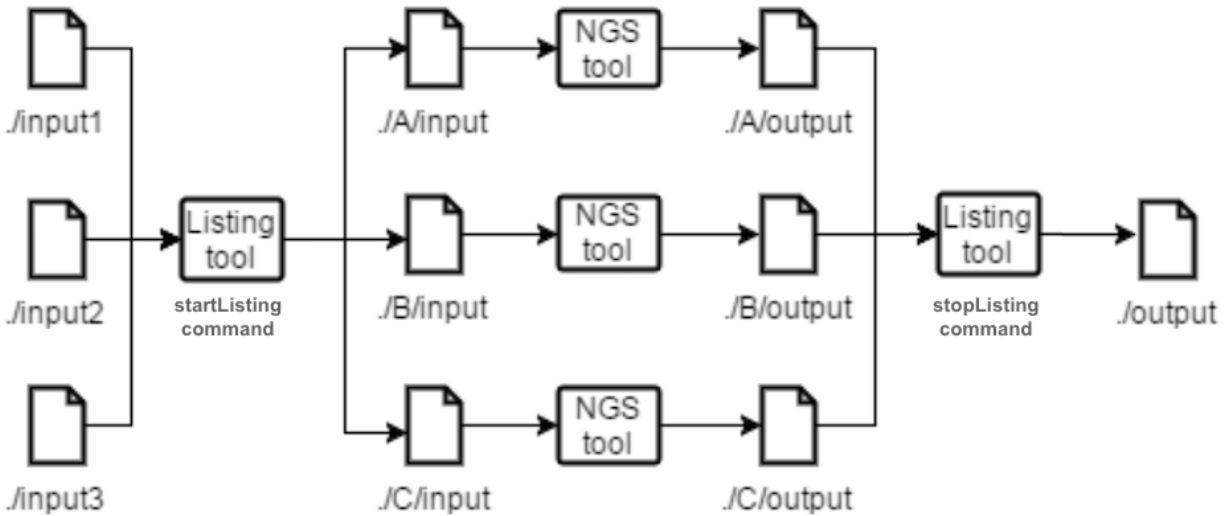


Figure 3.2 Listing and Joining tools example.

apply the same command or set of commands to a list of specified files. Listing tools generate a task to move the files to the newly generated directories and change the files names to match the name used in the `.pipes` file. Listing tools generate the same tasks as splitting and joining tools on commands that depend from them. The listing tools purpose is to allow a user to provide multiple files and treat them as one in the `.pipes` file. The listing tools have two commands, `startListing` and `stopListing` that are similar to `split` and `join` tools, respectively, but applied when the input are multiple files (passed as a zip file). The way data partitioning and dependencies inference is implemented, allows users to benefit from parallelization without adding complexity to the DSL and the process of writing a `.pipes` file. The analyser also skims all the outputs that will be produced in order to create a list of the directories that have to exist for the swift and correct execution of the pipeline. These directories are stored in an array on the intermediate representation, under the `directories` property. With the current version of the analyser, we have achieved a solution that allows users to split data and that allows to infer a topological graph from task dependencies, enabling the parallelization of the pipeline execution, without increasing the DSL complexity.

```

tool "Listing" "DockerConfig" {
  command "startListing" {
    argument "referenceName" "reads.sam"
    argument "filesList" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪reads1.sam snp-pipeline-master/snppipeline/data/lambdaVirusInputs/reads2.sam snp-
↪pipeline-master/snppipeline/data/lambdaVirusInputs/reads3.sam snp-pipeline-master/
↪snppipeline/data/lambdaVirusInputs/r
  }
}

```

Listing 3.6 An example of tool with type listing type.

More details on the type of tools can be found in this [report](#).

The NGSPipes Editor is a user-friendly editor for graphically define pipelines. Using this editor is very simple to define each processing step of the pipeline (i.e. a command) as well as the data to be used at each step (i.e., arguments).

The following sections show how to use the editor.

Download NGSPipes Editor

The NGSPipes Editor is a Java Application. To deploy this it in your system you need:

- Java Runtime Environment (JRE), version 8, which can be obtained from [here](#).

Download the editor from [here](#).

Execute NGSPipes Editor

To run the editor, and uncompress the downloaded file. Then you should have the following file tree:

```
|-- editor-1.0\  
    |-- bin\  
        |-- editor      (CUI OSX/Linux run script)  
        |-- editor.bat  (CUI Window run script)  
    |-- lib\  
    |-- ...
```

Then you can simple double click on corresponding script.

If you have OSX and you prefer the double click version to run the editor, it may appears, only the first time after you double click it, the following info:



Figure 4.1: Running a jar in a first time in OSX.

Then, go to “System Preferences” and choose “Security and Privacy”

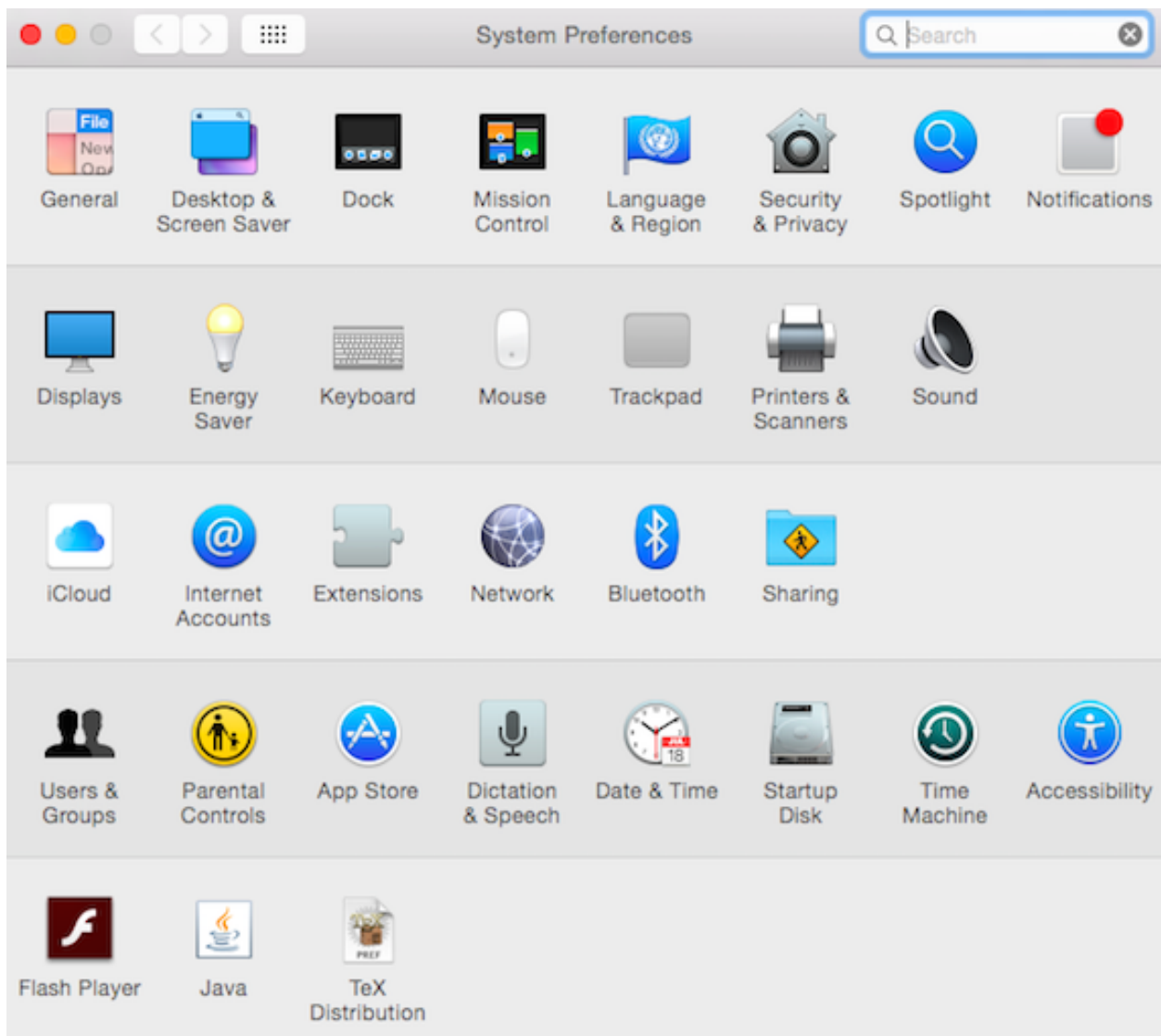
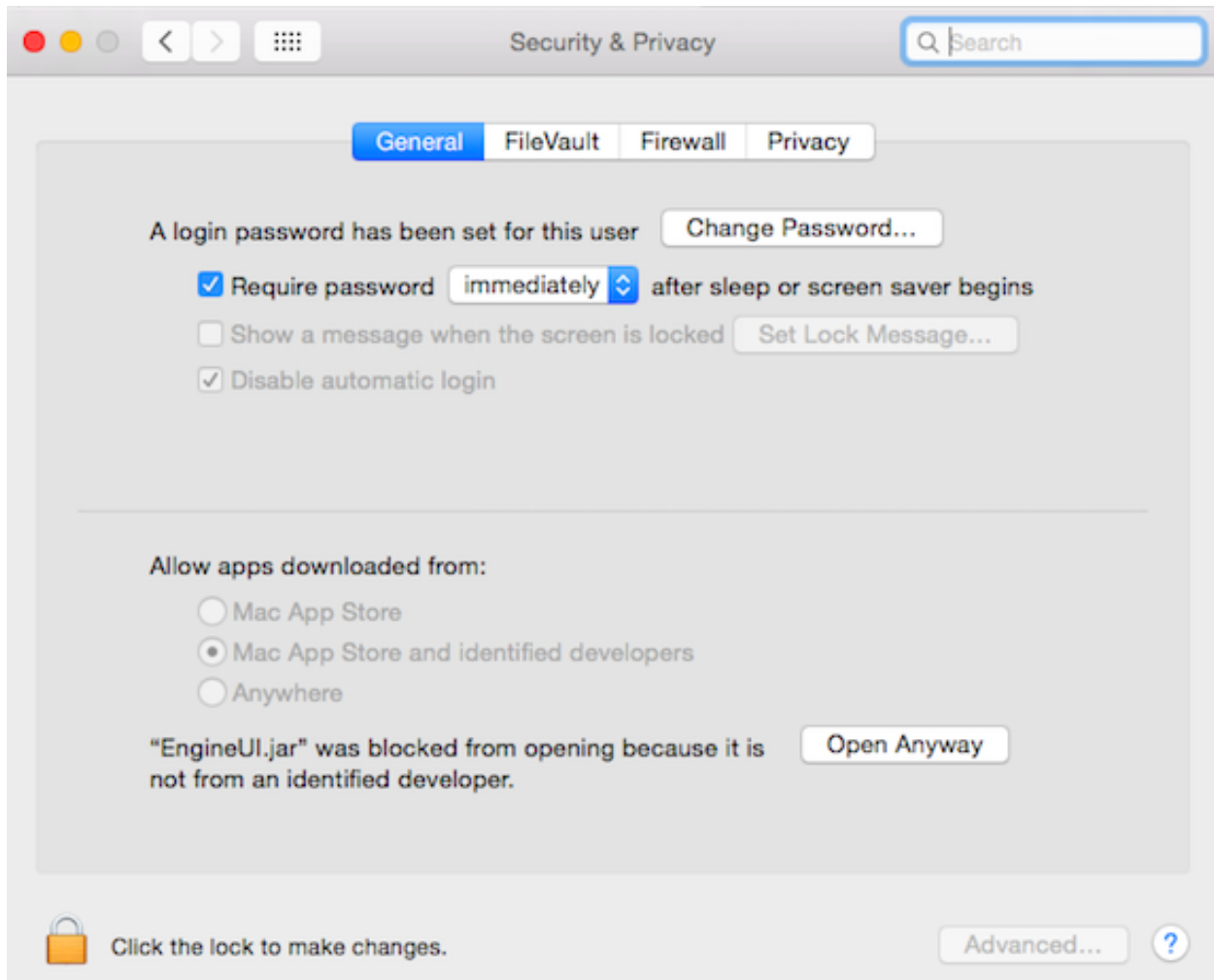


Figure 4.2: Selecting System Preferences in OSX.

Then select the button “Open anyway”



Notice that depending on the MAC OS version, it may be necessary to unlock to make changes and to select the option “Allow apps downloaded from Anywhere”

Figure 4.3: Allowing to run the jar file in OSX (just appears at the first time).

The initial GUI that appears from editor is the following:

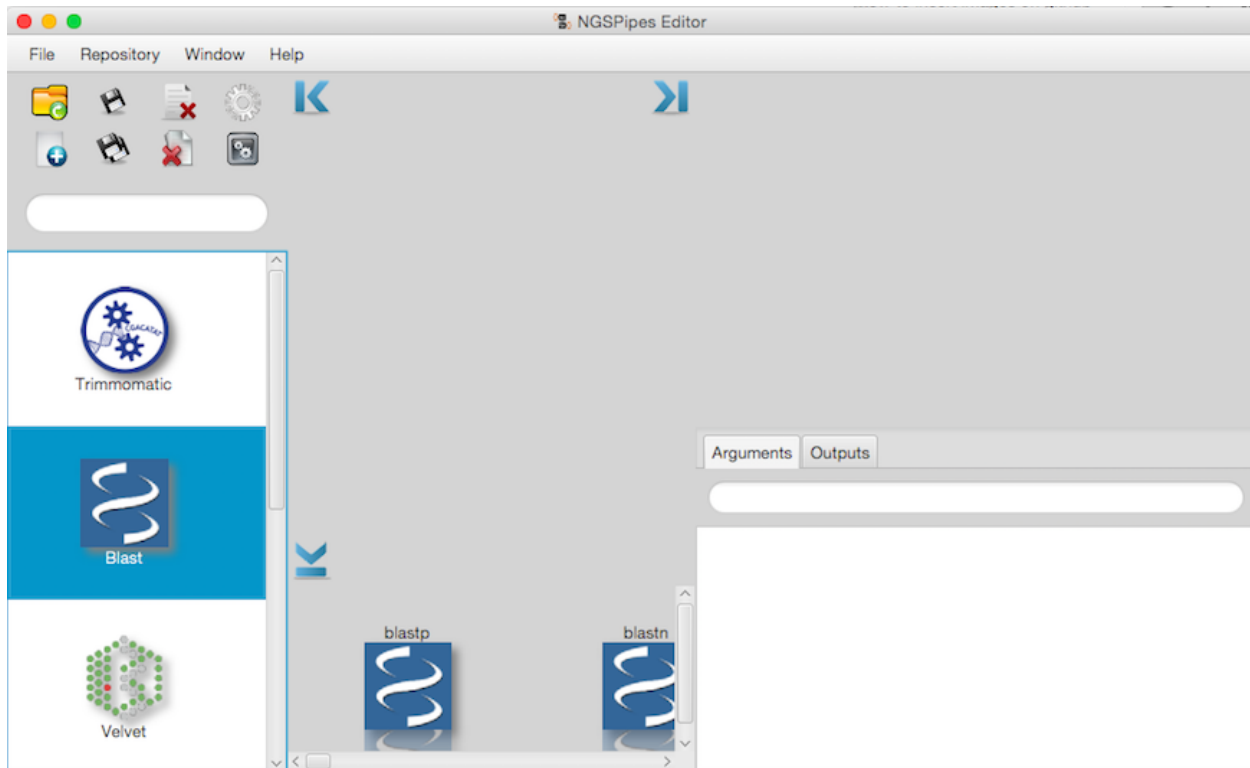


Figure 4.4: Editor initial screen.

In the following sections it will be explained how to use the editor. Moreover, in the editor's menu, selecting `help` and then the menu item `about`, it is possible to find some tutorial videos to help to use NGSPipes Editor.

NGSPipes Editor Sections

When defining a new pipeline (we will explain how to define a new pipeline in the Section *Creating a new pipeline*), the editor environment will appear similar to one of Figures 4.5 and 4.6 (it depends on the edition that is being performed).

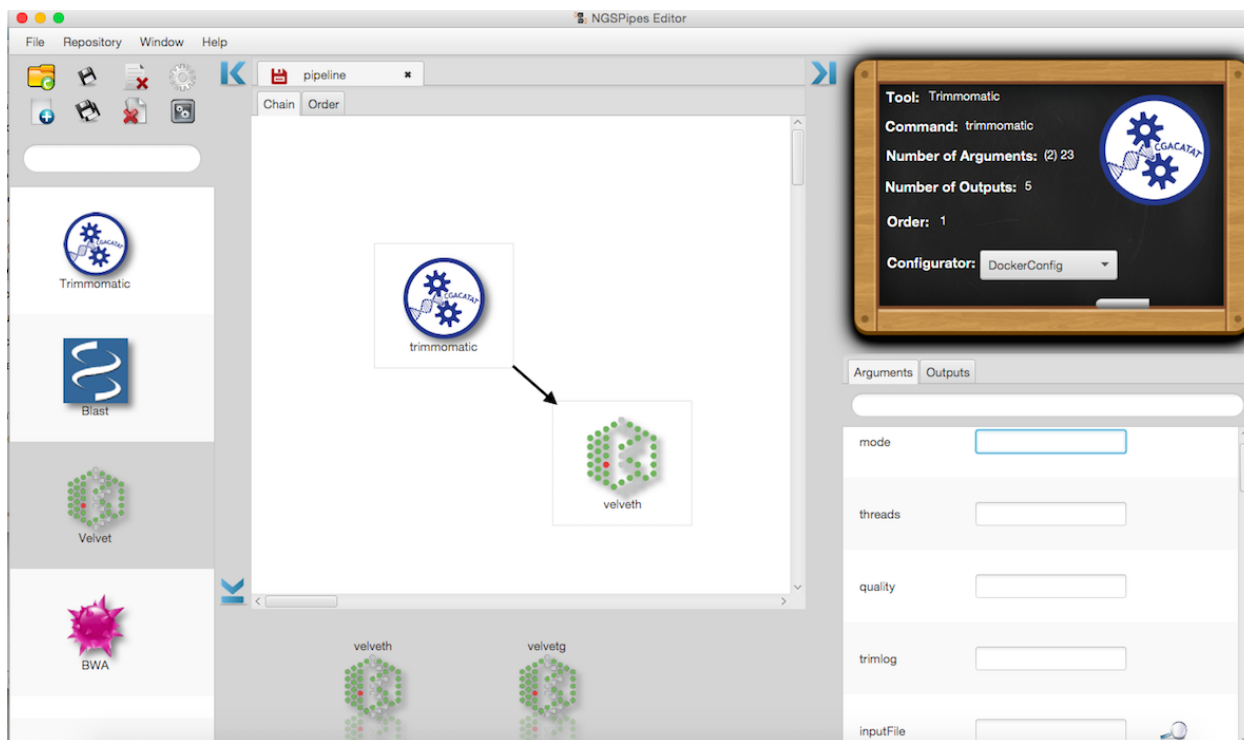


Figure 4.5: How NGSPipes editor looks like when defining a pipeline.

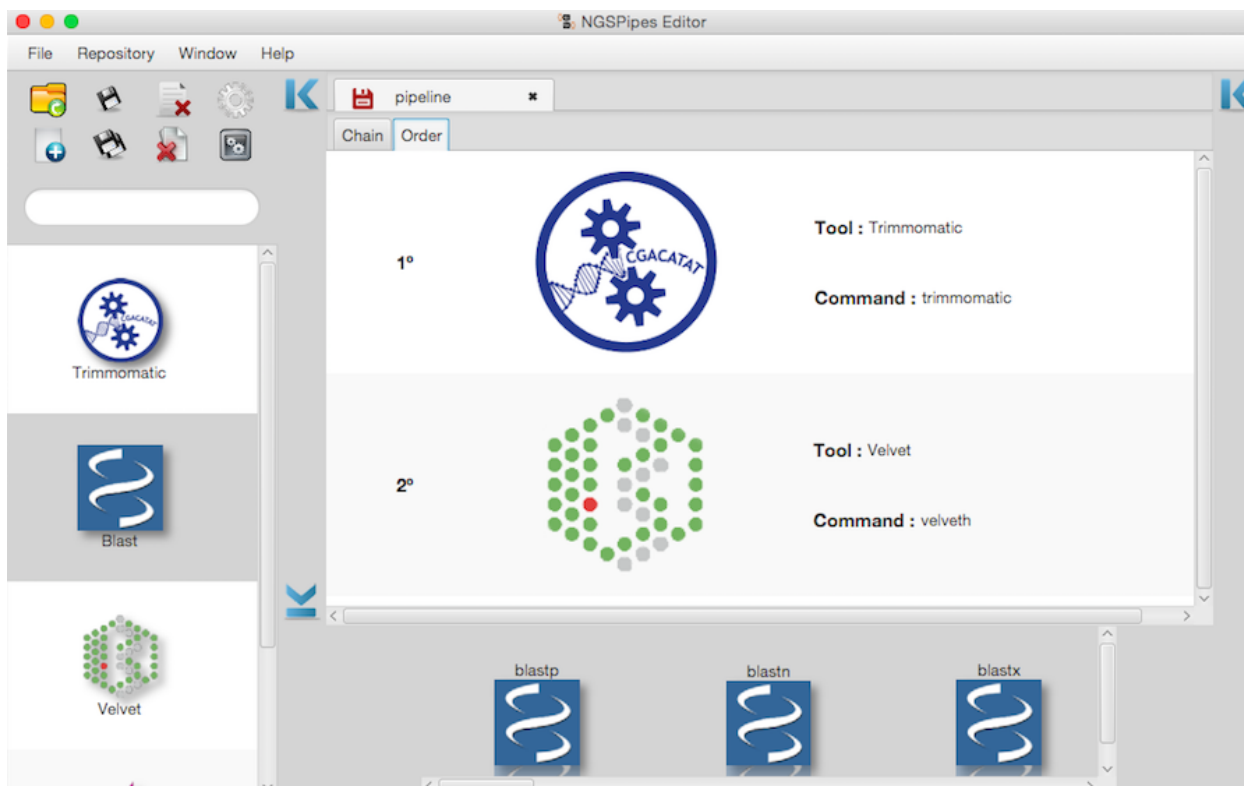


Figure 4.6: How NGSPipes editor looks like when consulting a possible execution order of the pipeline.

The NGSPipes Editor is composed of 5 sections: utilities; repository; tools; commands; pipeline and menu bar. These sections are pointed out in Figure 4.7.

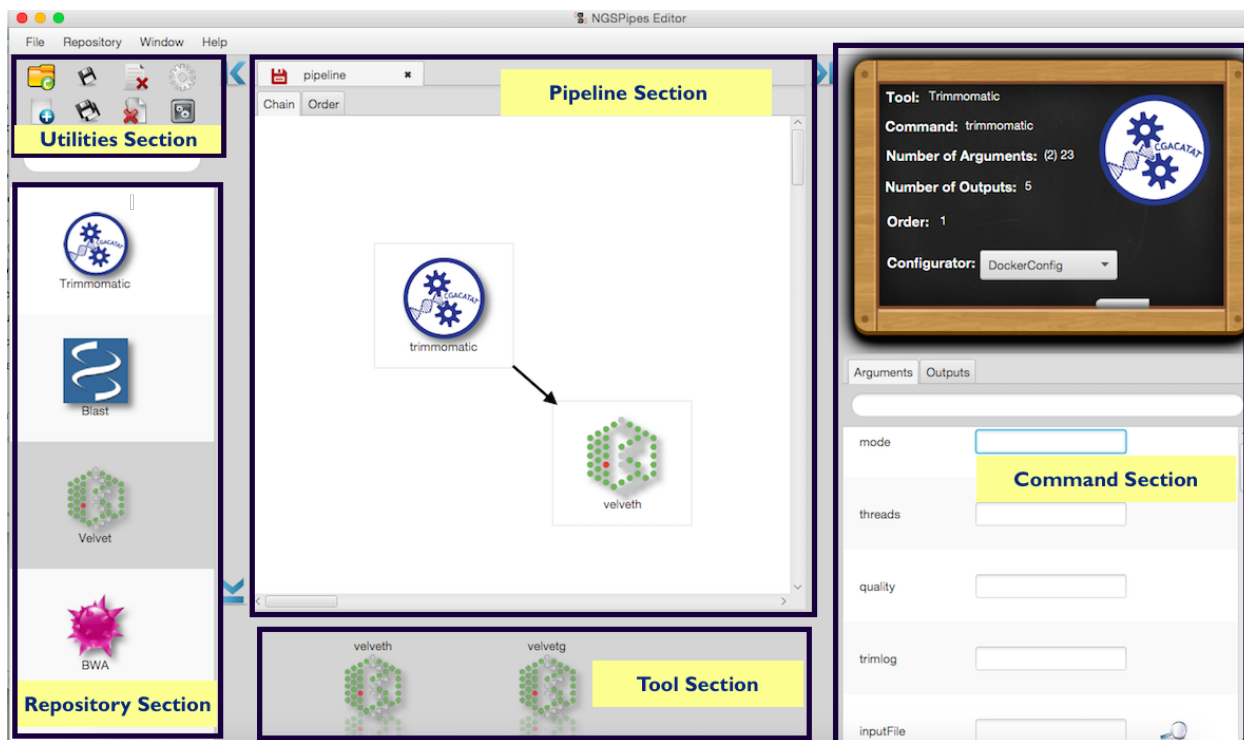


Figure 4.7: NGSPipes editor sections.

The **utilities** section includes all the buttons for executing the utilities actions, such as saving the active pipeline, closing it and generate the final version, when in this last case the user is asked to define the input and output directory ((see Section 6 for more information on pipeline generation). There are also in this section similar buttons for applying these actions for more than one workflow at the same time. Moreover, it is also in this section that exists a button for creating a new pipeline (see Section *Creating a new pipeline*) for more information).

More specifically,

- **New** button - Create a new pipeline;
- **Open** button - Open an existent pipeline;
- **Save** button - Save the active pipeline;
- **Save All** button - Save all the open pipelines;
- **Close** button - Close the active pipeline;
- **Close All** button - Close all the open pipelines;
- **Generate** button - Generate the pipeline in the NGSPipes language, *i.e.*, generate the file with extension `.pipes` for the active pipeline. This file is essential for executing the pipeline with the NGSPipes Engine (see <https://github.com/ngspipes/engine/wiki> for more information). With this action, the user is asked for defining the input and output directory. It is asked if it is allowed to copy the input files that are not already in the Input directory.
- **Generate All** button - Generate all the files with extension `.pipes` for all the active pipelines. For each pipeline it is applied the action of the **Generate** button.

Select the tools repository

When the Editor starts, it loads a local repository that is included within the tool. However, user can select other repository, either local or remote. To select other repository, go to the menu `Repository` and select `Change repository`. In the version 1 of the Editor, there are four types that are supported, as depicted in Figure 4.8.

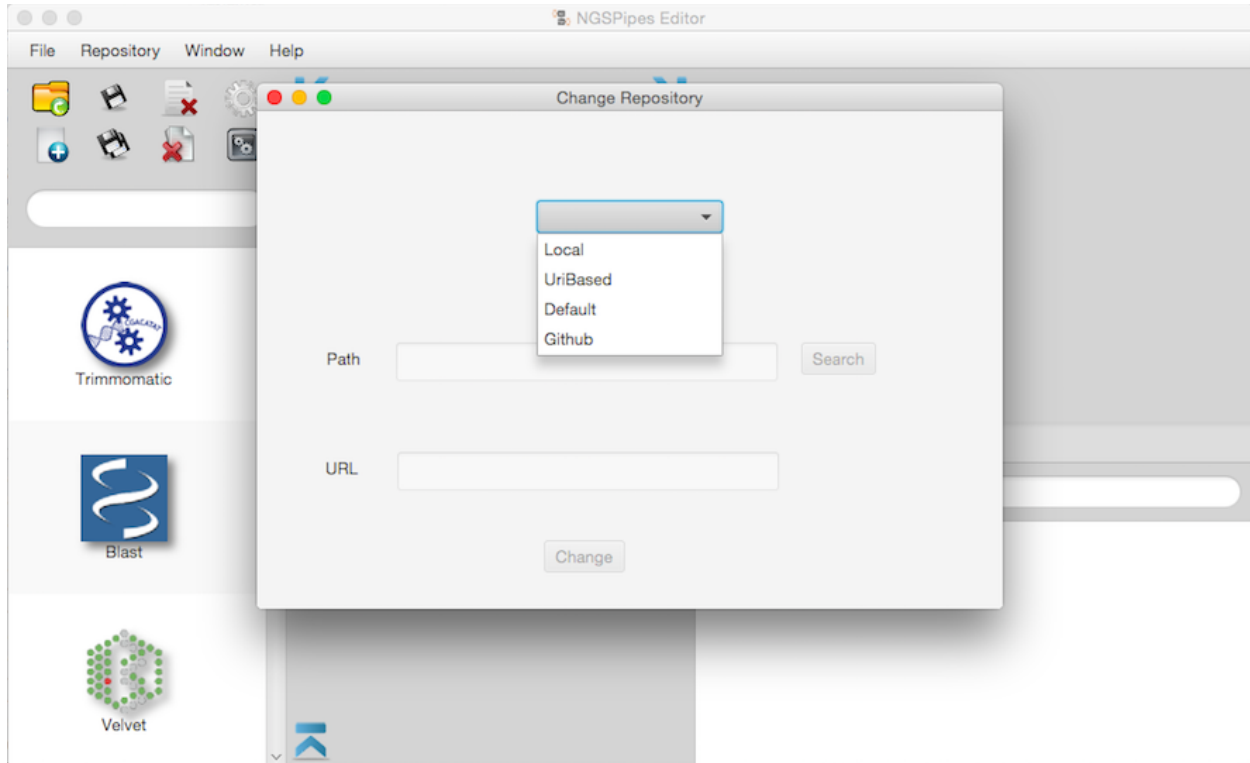


Figure 4.8: Setting the tool’s repository for the current pipeline.

The `Default` is a local repository that is included within the tool. To specify other local repositories, it is necessary to select the `Local` option and, with the `search` button, select the path to the repository. Instead, to specify a remote one, it is necessary to select one of the following options, `github` or `uribased`, depending on the type of the remote repository. For instance with the tools’ repository example, namely `https://github.com/ngspipes/tools`, please select the option `github`, with the URL `https://github.com/ngspipes/tools`.

Then, it is possible to observe the **repository section** on Figure 4.9.

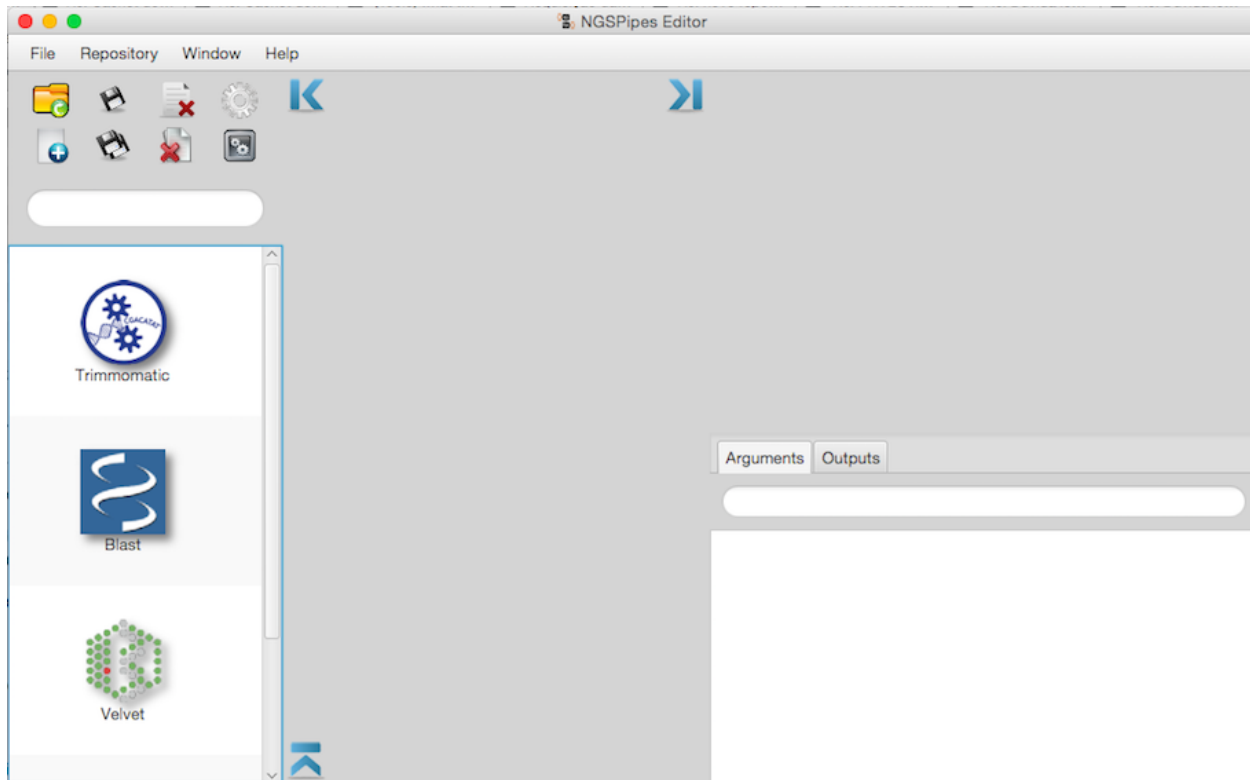


Figure 4.9: How NGSPipes Editor looks like when there is no pipeline loaded/created.

In the **repository section**, the user may explore all the tools that are available on the repository, as well as filter them by name. It is also possible to obtain the description of a tool if we place the mouse over the tool's logotype. Selecting a tool, will open the **tool section** (in Figure 4.5), Velvet is selected and the tool section is at the bottom, centered), where is possible to navigate over all the commands available within that tool. In some cases, the tool has only one command, as for instance the Trimmomatic tool. It is also possible to obtain the description of each command similarly as done for obtaining the tool description. In these sections, the user may obtain more information about a given tool, command, input or output, only by selecting a given item of one of these sections with the right button of the mouse and selecting the Description option. This option opens a new window with all the information available on the repository, as depicted in Figure 4.10.

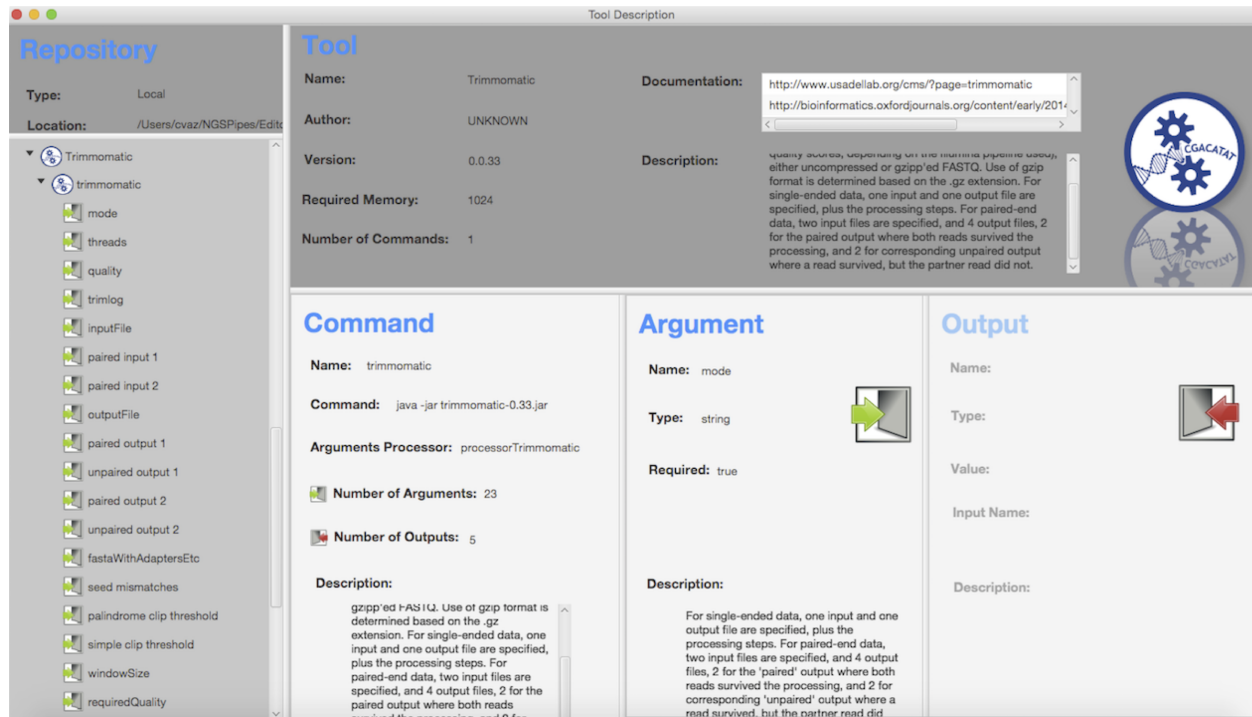


Figure 4.10: Description of all tools included in the selected repository.

The other sections, namely the **pipeline** and **command** sections will be detailed in next section.

Creating a new Pipeline

In order to create a new pipeline, after selecting the tools' repository, please select the button with a plus (in the utilities section) or go to the menu **File** and select the option **new**.

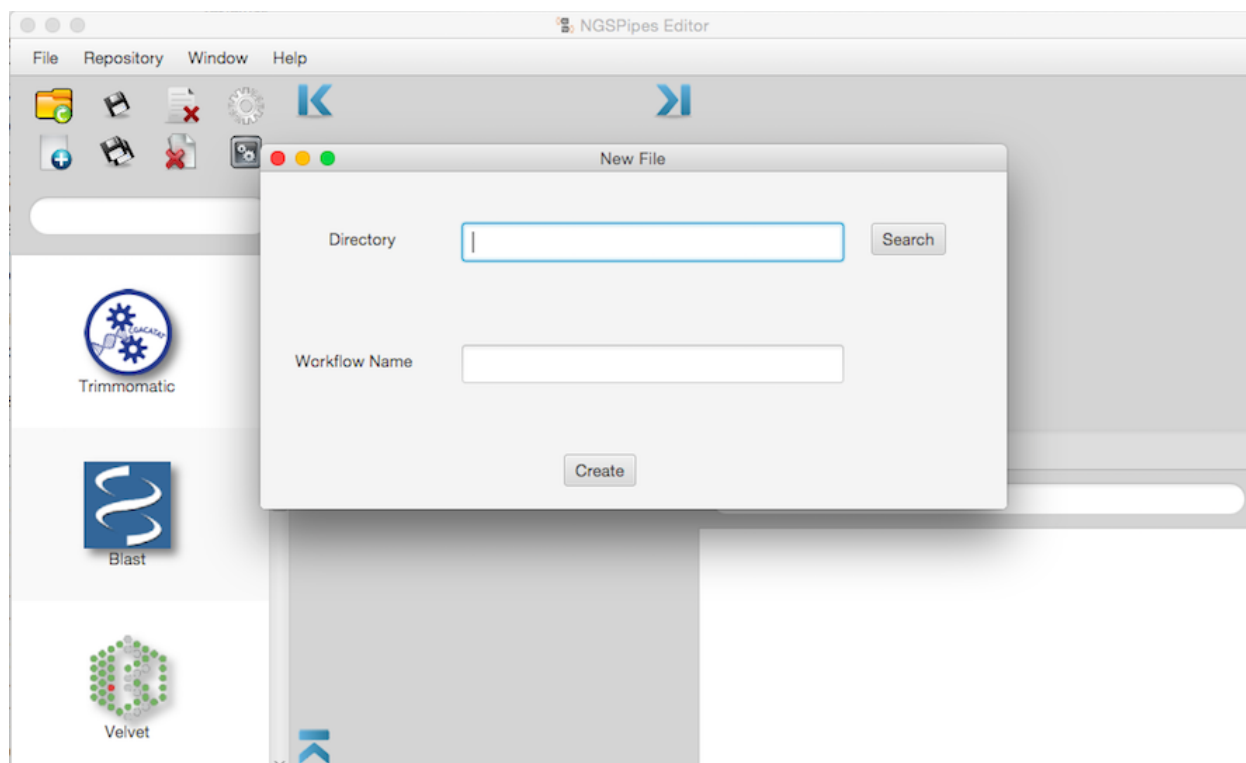


Figure 4.11: Creating a new pipeline in the editor.

After defining the directory where the pipeline is kept and the name of the pipeline, it will appear the **pipeline section**, as depicted in the following Figure.

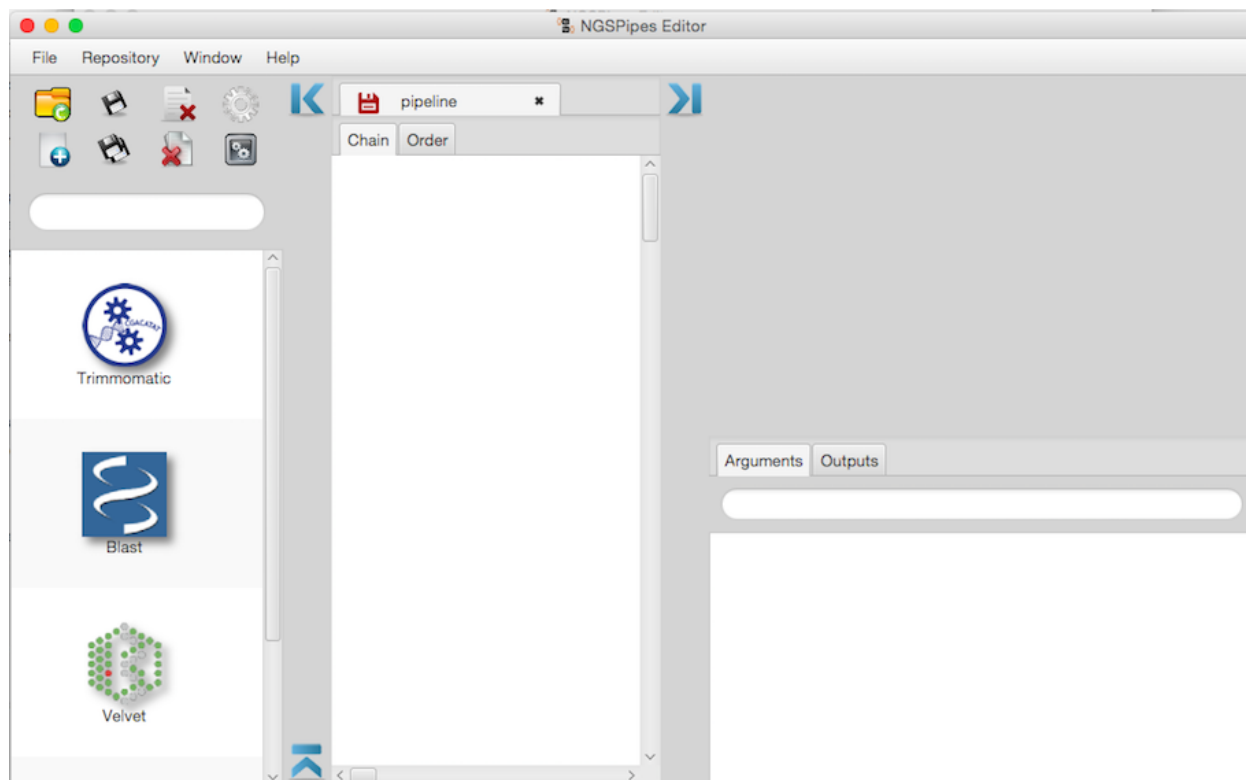


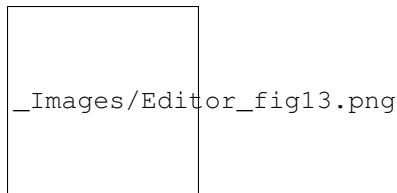
Figure 4.12 The pipeline section.

The pipeline creation generates a file with extension `.wf`. This file keeps all information of a pipeline within the editor, not only the commands as well as the visual positions of the pipeline within the editor.

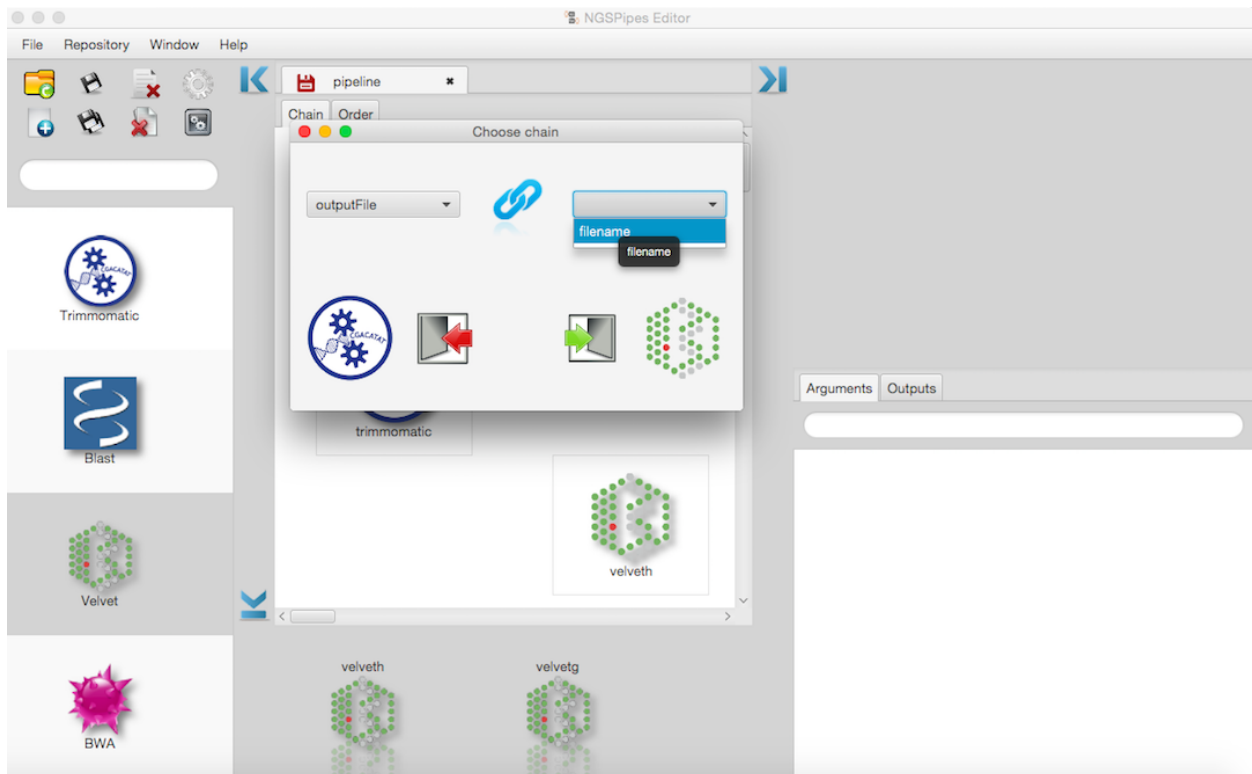
The **pipeline section** has two panels, the *chain* and the *order* panel. In the *chain* panel, the user can add or remove commands as well as set arguments and chains. For instance, in (Figure 4.5) it was defined a chain between the `trimmomatic` command and the `velveth` command since the output file of the first one is an input file of the second one. For more information about chains, please see <https://github.com/ngspipes/dsl/wiki>. Before defining the chain, it is necessary to add the commands to execute within the pipeline. Adding a new command to the *chain* panel (notice that is necessary to previously select the tool in the *repository* section and then in the *tool* section select the command) is simply done by a *drag and drop* action.

After adding a command and double clicking on it, it appears the **command section** (see the right size of Figure 4.5). This section only appears when the user does a double click on a command over the *chain* panel. In the **command section**, the user can set the arguments of the selected commands as well as to confirm its generated output file names.

For defining the chain between two commands, it is necessary to drag the **blue icon** that appears in the command image within the *chain* panel, after a double click.

**Figure 4.13: Command selection.**

Then it is necessary to select the **blue icon** and drag it to the command to which is to do the chain operation. After that, it will appear the following figure:

**Figure 4.14: The chain panel.**

Here it is necessary to select the output to be chained as an argument to the other tool. After that, it is necessary to click on the *blue icon* and the chain action will be set and a black arrow will appear between both tools (see next Figure).

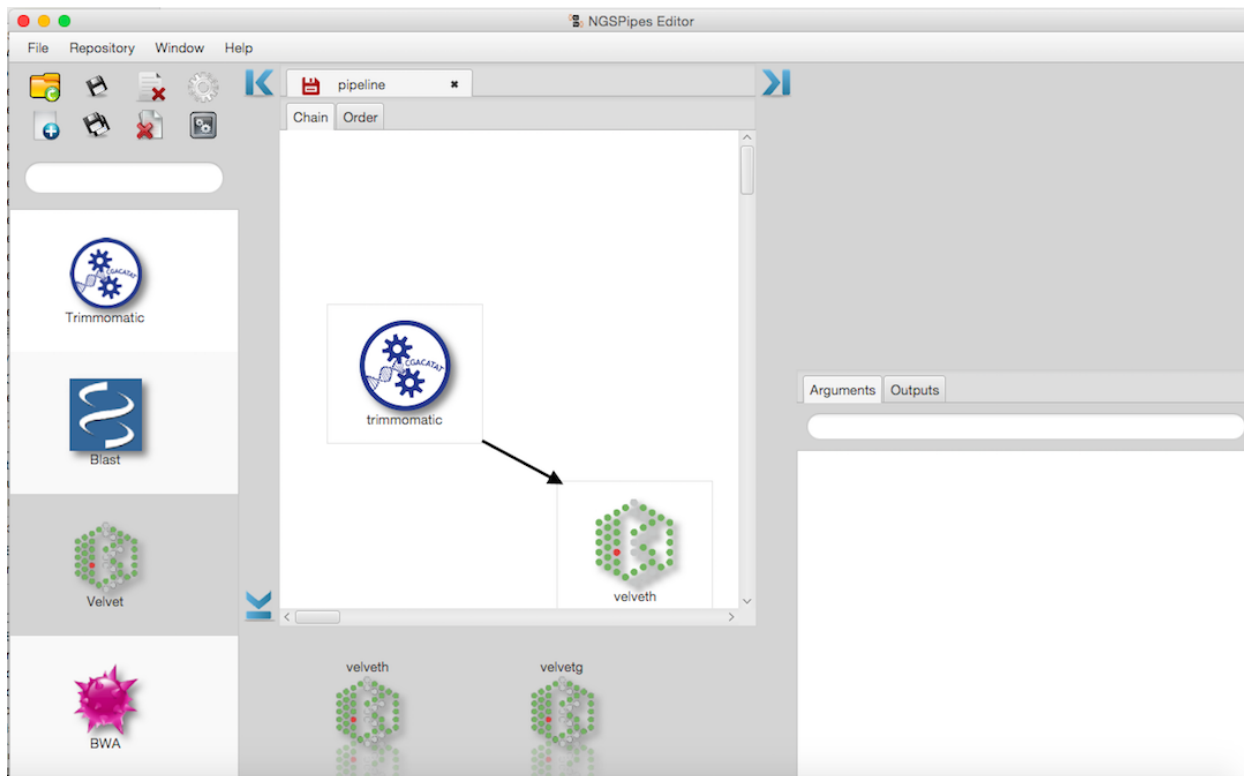


Figure 4.15: After setting a chain action between two commands.

After setting the required arguments of all commands added to the pipeline and setting all the chains, the user can observe in the *order* panel one of the possible inferred orders of the pipeline execution (see Figure 4.6). This order is given, assuring that command dependency and priority are preserved. For more information about command dependency and priority, please see NGSPipes repository section.

Moreover, in the editor's menu, selecting *help* and then the menu item *about*, it is possible to find some tutorial videos to help to use NGSPipes Editor.

Generate the final pipeline version to execute

As mentioned before, creating a new pipeline generates a file with extension `.wf` and with the name chosen by the user. This file keeps all information of a pipeline within the editor, not only the commands as well as its visual positions.

However, if the user wants to execute the pipeline in the NGSPipes Engine (<https://github.com/ngspipes/engine/wiki>), it is necessary to produce a file with extension `.pipes`. This file is written using the NGSPipes language (<https://github.com/ngspipes/dsl/wiki>) and thus does not have visual information. For producing the file with extension `.pipes` it is necessary to select the generate button or go to `File -> Generate pipeline`.

Loading an existing pipeline

To load an existing pipeline, it is necessary to go to `File -> Open` and choose a pipeline file (with extension `.wf`) or select the open button.

Multiple loaded pipelines

It is possible to have multiple loaded pipelines, but just one is active.

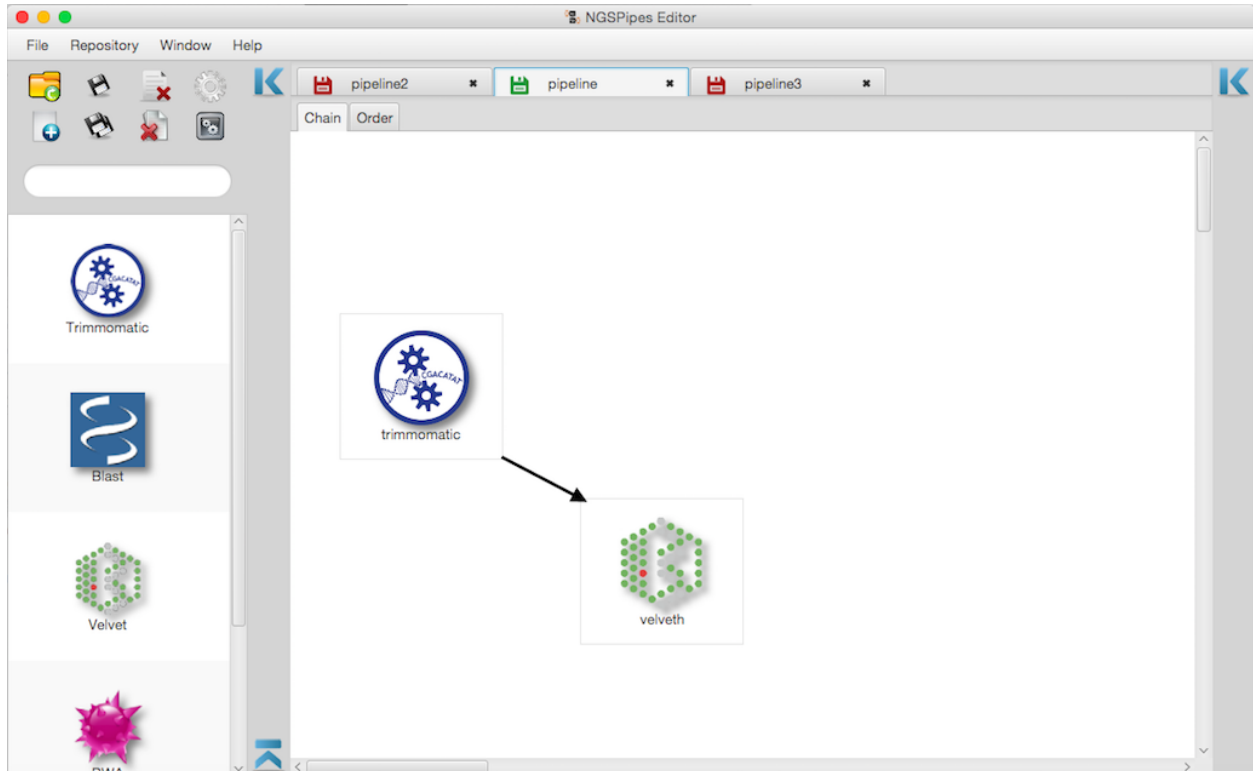


Figure 4.16: Multiple loaded pipelines.

Error Reporting

Each argument of a command of a given tool has a type and may be or not optional. The required arguments (not optional) appears in a red box. The type of each argument must be one of the following: integer number; file; text; real number or a directory. When the user assigns an incompatible type to a given argument, the editor will generate an error message to report that situation. An example of this situation is depicted in the following figure:

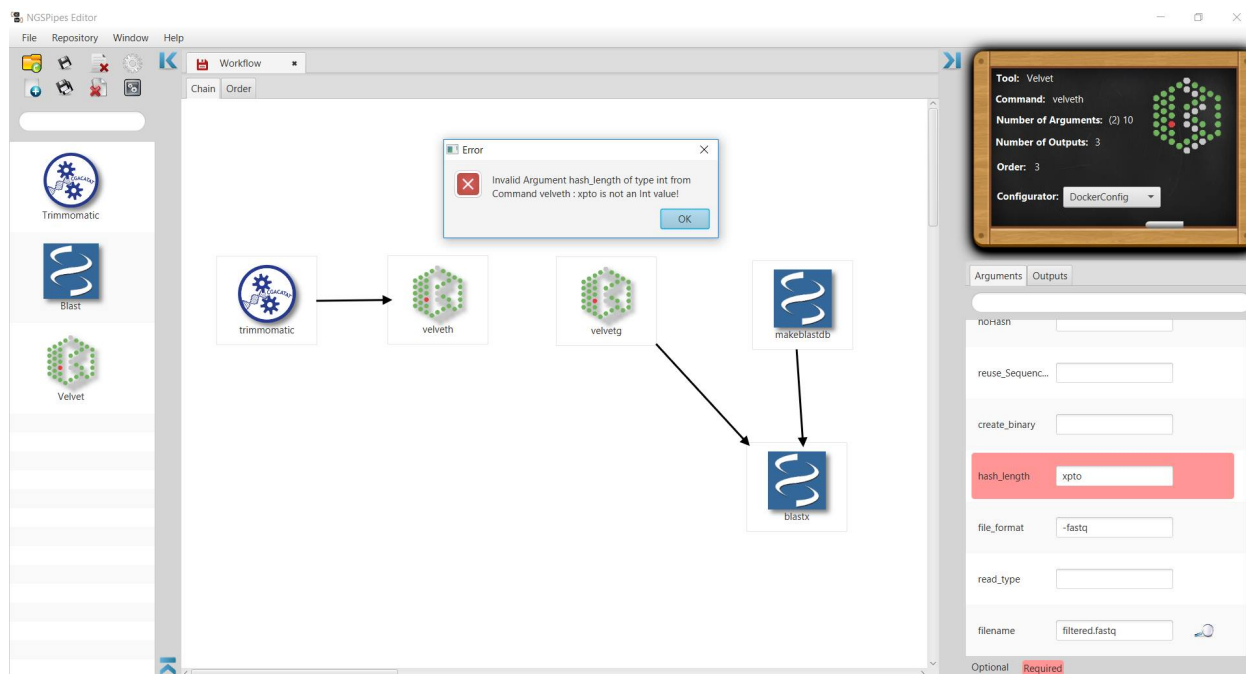


Figure 4.17: Incompatible value type for the selected argument.

If the user does not set a compatible value to a required argument, the editor will also generate an error message to report that situation, when generating the pipeline specification (pressing the generate button).

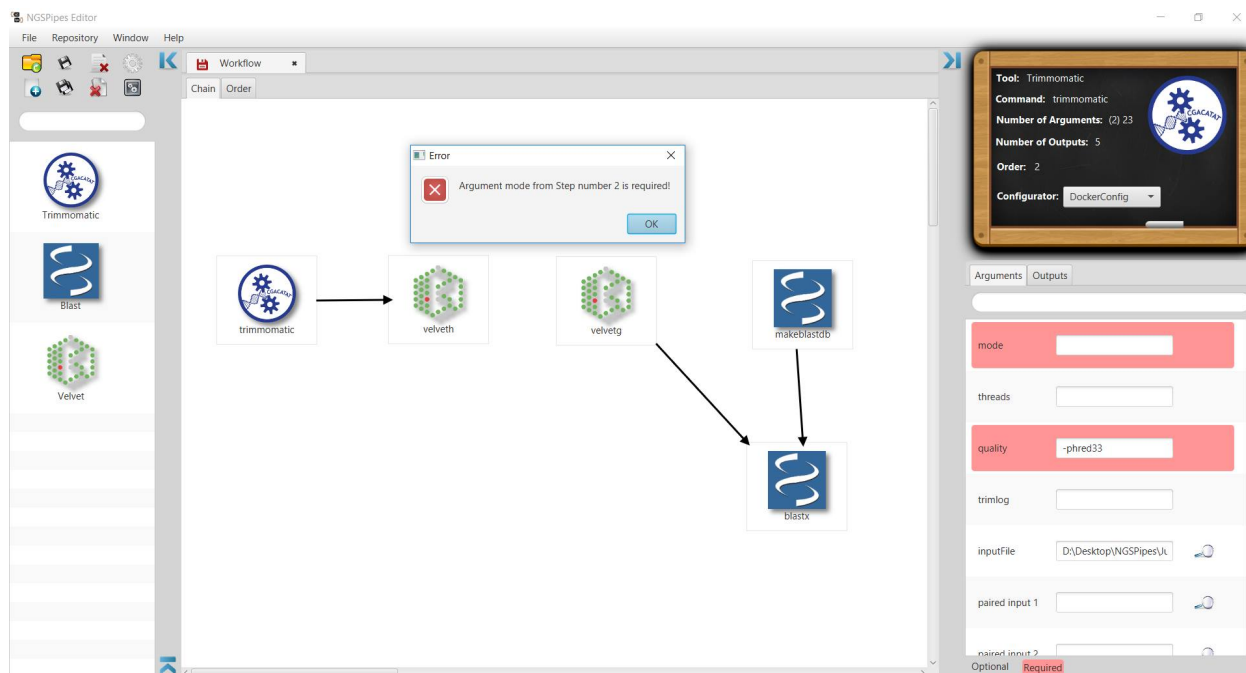


Figure 4.18: Missing a value for a required argument.

Multiple inputs

The tools can produce multiple outputs. These tools also might require multiple inputs coming from different tools or the same tool. When the multiple inputs are from the same tool, NGSPipes supports it by adding a numbered label on the arrow. This functionality is depicted in the following figure:

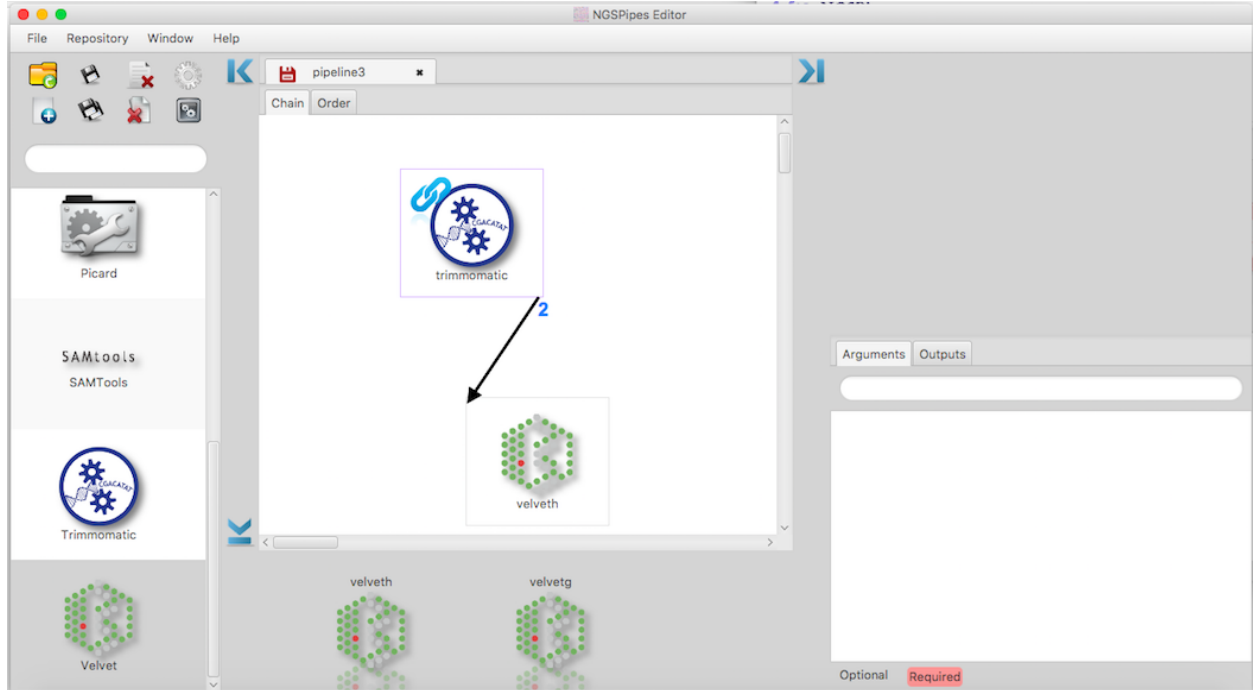


Figure 4.19: Multiple inputs from a previous command.

Notice that this label is only visible when one of the tools is selected.

Our framework offers two engines. Based on the same definition and the same tool's repository, a pipeline can be put to run either on the workstation or on a compatible cloud environment. Both engines analyse the pipeline description and transformation it to an executable format, determining resource requirements of each tool based on the tool configuration present in the repository.

Engine for workstation

The *NGSPipes engine for workstation* starts with a pipeline description and transform it into a sequence of calls to the designated tools. After this, the *engine* automatically configures and executes each tool in isolation from the remaining system environment (using a dedicated virtual machine).

The *NGSPipes engine for workstation* is available in two flavors: a command line user interface (CUI) and a graphic user interface (GUI). The first is ideal to use when running on remote servers (either physical or deployed as virtual machines in the cloud), although it can also run locally. The second one can be used in systems where a graphical display is available.

The following sections shows how to run the engine. To build from source code please follow the instructions in subsection "Instructions to build NGS Pipes Engine from source code".

Requirements to run the engine for workstation

The machine where *engine* is to be executed needs the following tools:

- Java 8 Development Kit (<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>).
- VirtualBox version ≥ 5.0 (<https://www.virtualbox.org/wiki/Downloads>). NOTE: Ensure the command `VBoxManage` can be found by the *command line* of your operating system.

Install engine for workstation

The *engine* is made of a regular Java application and a VirtualBox's compliant image (also identified as *executor*). To deploy this in your system:

1. Download [engine-2.0-zip](#) from our file server and uncompress to a working directory (WD)
2. Download the *executor* image from [here](#) and uncompress to your work directory (WD\engine-1.0\)
3. Follow the instructions bellow to either run in a system in a *console* or with a *graphical interface*.

After these steps you should have the following file tree:

```
WD
|-- engine-1.0\
    |-- NGSPipesEngineExecutor\
        |-- NGSPipesEngineExecutor.vbox
        |-- NGSPipesEngineExecutor.vdi
    |-- bin\
        |-- engine          (CUI OSX/Linux run script)
        |-- engine.bat      (CUI Window run script)
        |-- engine-ui       (GUI OSX/Linux run script)
        |-- engine-ui.bat   (GUI Window run script)
    |-- lib\
        |-- ...
    |-- (other files, e.g. the pipeline description)
```

Run the Engine for workstation

The engine is provided as a console application or a graphical user interface application.

command line tool

This is a regular Java application packed as a JAR file. To run, use the following command line at the working directory (WD):

Windows:

```
c:\WD>engine-1.0\bin\engine.bat <mandatory arguments> [<optional arguments>]
```

OSX/Linux

```
user@machine:/home/WD$ engine-1.0/bin/engine <mandatory arguments> [<optional arguments>]
```

Parameters

- The command line tool has the following mandatory parameters :

-pipes Relative ou absolute path of the pipeline description (mandatory). This file must be a .pipes extension file, where the pipeline is written using the NGSPipes language.

-in Absolute path at the user machine where the input data files are located (mandatory).

-out Absolute path at the user machine where the output data file will be placed (mandatory).

- The command line tool has the following *optional* parameters :

-cpus Assigned cores. Default is 2 CPUs.

-mem Assigned memory (in Gigabytes). If not present, the number of gigabytes allocated to the engine will be inferred by analyzing the tools in the pipeline.

-from Initial pipeline step. If not present, the first step will be executed.

-to Final pipeline step. If not present, the pipeline will execute all the steps.

-executor Executor image name. If not present, uses the image located at WD/NGSPipesEngineExecutor.

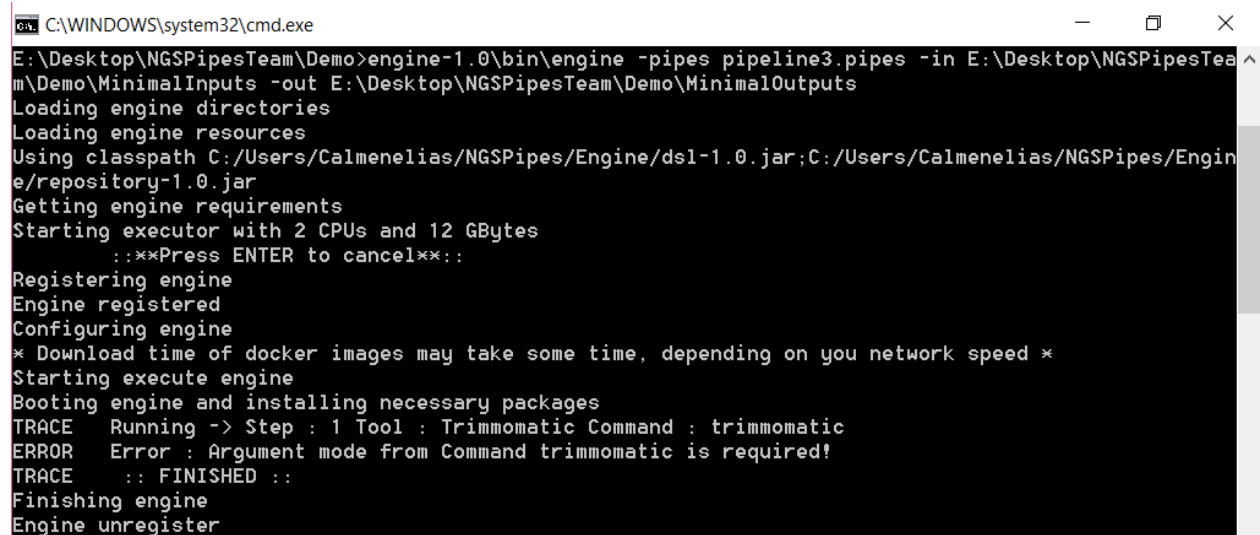
Example

A small example with only mandatory arguments (a more complete description is presented in the subsection “Use case”).

```
engine-1.0/bin/engine -in /home/ngs/inputs -out /home/ngs/outputs -pipes pipeline.
↳ pipes
```

Error reporting

The next figure shows an error report from the pipeline engine when executing a pipeline where a mandatory argument is not specified.



```
C:\WINDOWS\system32\cmd.exe
E:\Desktop\NGSPipesTeam\Demo>engine-1.0\bin\engine -pipes pipeline3.pipes -in E:\Desktop\NGSPipesTeam\Demo\MinimalInputs -out E:\Desktop\NGSPipesTeam\Demo\MinimalOutputs
Loading engine directories
Loading engine resources
Using classpath C:/Users/Calmenelias/NGSPipes/Engine/dsl-1.0.jar;C:/Users/Calmenelias/NGSPipes/Engine/repository-1.0.jar
Getting engine requirements
Starting executor with 2 CPUs and 12 GBytes
::**Press ENTER to cancel**::
Registering engine
Engine registered
Configuring engine
* Download time of docker images may take some time, depending on you network speed *
Starting execute engine
Booting engine and installing necessary packages
TRACE   Running -> Step : 1 Tool : Trimmomatic Command : trimmomatic
ERROR   Error : Argument mode from Command trimmomatic is required!
TRACE   :: FINISHED ::
Finishing engine
Engine unregister
```

The next figure shows an error report from the pipeline engine when executing a pipeline where an argument is used with a non-compatible value type, according to the tool’s specification.

```

C:\WINDOWS\system32\cmd.exe

E:\Desktop\NGSPipesTeam\Demo>engine-1.0\bin\engine -pipes pipeline4.pipes -in E:\Desktop\NGSPipesTeam\Demo\MinimalInputs -out E:\Desktop\NGSPipesTeam\Demo\MinimalOutputs
Loading engine directories
Loading engine resources
Using classpath C:/Users/Calmenelias/NGSPipes/Engine/dsl-1.0.jar;C:/Users/Calmenelias/NGSPipes/Engine/repository-1.0.jar
Getting engine requirements
Starting executor with 2 CPUs and 12 GBytes
::**Press ENTER to cancel**::
Registering engine
Engine registered
Configuring engine
* Download time of docker images may take some time, depending on you network speed *
Starting execute engine
Booting engine and installing necessary packages
TRACE Running -> Step : 1 Tool : Trimmomatic Command : trimmomatic
INFO Executing : sudo docker run -v /home/ngspipes/Inputs:/shareInputs:/rw -v /home/ngspipes/Outputs:/shareOutputs:/rw ngspipes/trimmomatic0.33 java -jar trimmomatic-0.33.jar SE -phred33 /shareInputs/ERR406040.fastq /shareOutputs/ERR406040.filtered.fastq ILLUMINACLIP:/shareInputs/TruSeq3-SE.fa:2:30:10 SLIDINGWINDOW:4:15 LEADING:3 TRAILING:3 MINLEN:36
INFO TrimmomaticSE: Started with arguments: -phred33 /shareInputs/ERR406040.fastq /shareOutputs/ERR406040.filtered.fastq ILLUMINACLIP:/shareInputs/TruSeq3-SE.fa:2:30:10 SLIDINGWINDOW:4:15 LEADING:3 TRAILING:3 MINLEN:36
INFO Automatically using 2 threads
INFO Using Long Clipping Sequence: 'AGATCGGAAGAGCGTCGTAGGGAAGAGTGTA'
INFO Using Long Clipping Sequence: 'AGATCGGAAGAGCACACGTCTGAACTCCAGTCAC'
INFO ILLUMINACLIP: Using 0 prefix pairs, 2 forward/reverse sequences, 0 forward only sequences, 0 reverse only sequences
INFO Exception in thread "Thread-0" java.lang.NullPointerException
INFO at org.usadellab.trimmomatic.fastq.FastqRecord.<init>(FastqRecord.java:24)
INFO at org.usadellab.trimmomatic.fastq.FastqParser.parseOne(FastqParser.java:81)
INFO at org.usadellab.trimmomatic.fastq.FastqParser.next(FastqParser.java:171)
INFO at org.usadellab.trimmomatic.threading.ParserWorker.run(ParserWorker.java:42)
INFO at java.lang.Thread.run(Thread.java:745)
INFO Input Reads: 212000 Surviving: 203476 (95.98%) Dropped: 8524 (4.02%)
INFO TrimmomaticSE: Completed successfully
INFO Command sudo docker run -v /home/ngspipes/Inputs:/shareInputs:/rw -v /home/ngspipes/Outputs:/shareOutputs:/rw ngspipes/trimmomatic0.33 java -jar trimmomatic-0.33.jar SE -phred33 /shareInputs/ERR406040.fastq /shareOutputs/ERR406040.filtered.fastq ILLUMINACLIP:/shareInputs/TruSeq3-SE.fa:2:30:10 SLIDINGWINDOW:4:15 LEADING:3 TRAILING:3 MINLEN:36 finished with Exit Code = 0
TRACE Running -> Step : 2 Tool : Velvet Command : velvet
ERROR Error : Invalid Argument hash_length of type int from Command velvet : xpto is not an Int value!
TRACE :: FINISHED ::
Finishing engine
Engine unregister
E:\Desktop\NGSPipesTeam\Demo>

```

User interface tool

The GUI version of the *NGSPipes Engine for workstation* allows the same operations but using a graphical interface. When installed at a working directory (WD), the tool can be executed in the file explorer of your operating system:

Windows

WD\engine-1.0\bin\engine-ui.bat

OSX/Linux

WD/engine-1.0/bin/engine-ui

If you have OSX and you prefer the double click version to run the editor, it may appear, only the first time after you double click it, the following info:

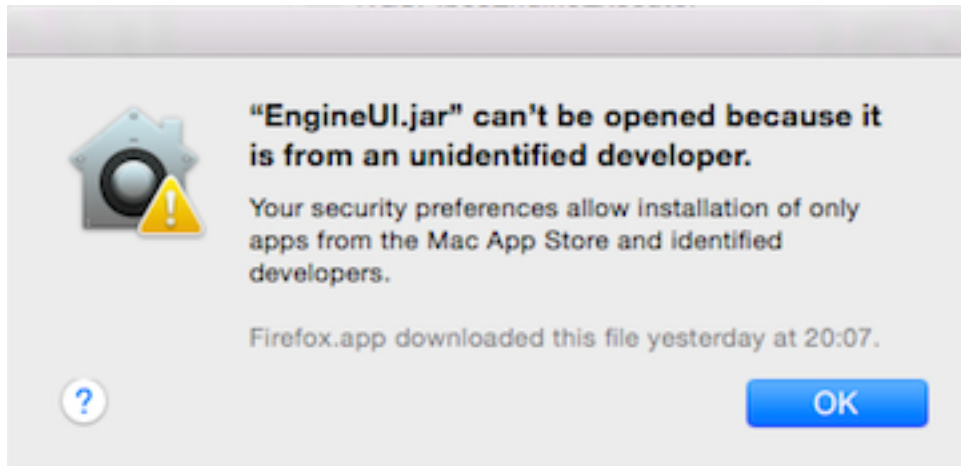


Figure 1

Then, go to "System Preferences" and choose "Security and Privacy"

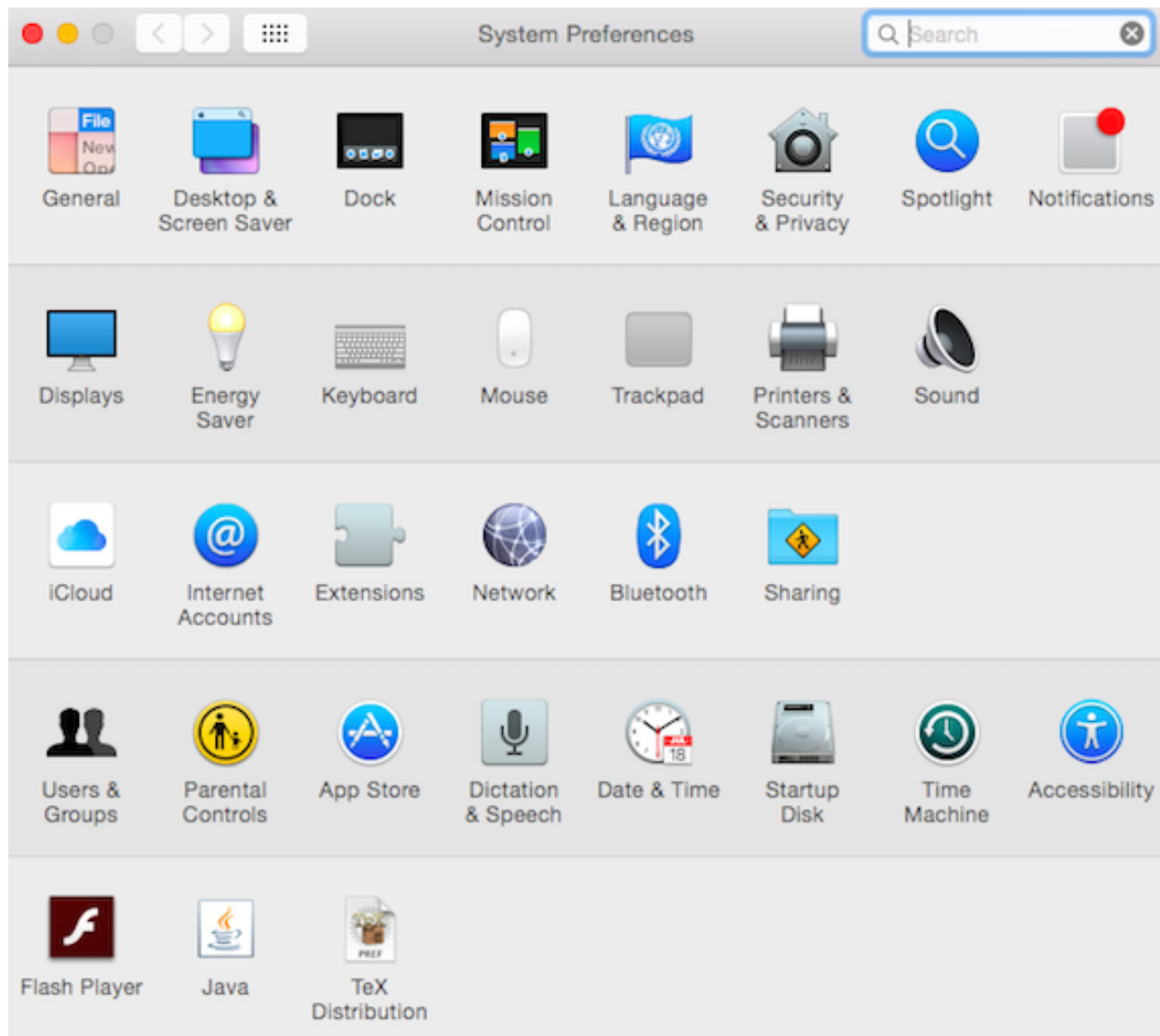


Figure 2

Then select the button “Open anyway”

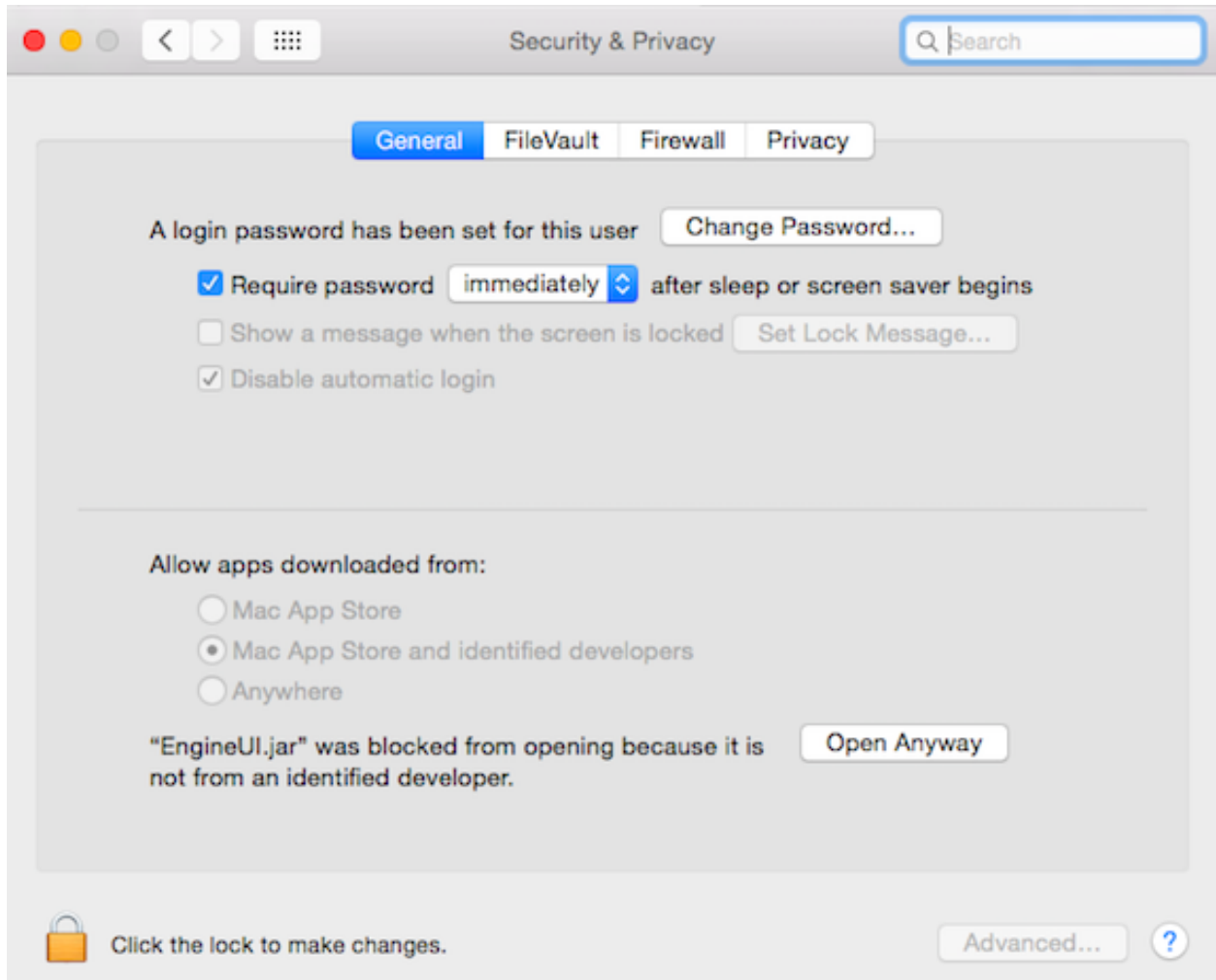
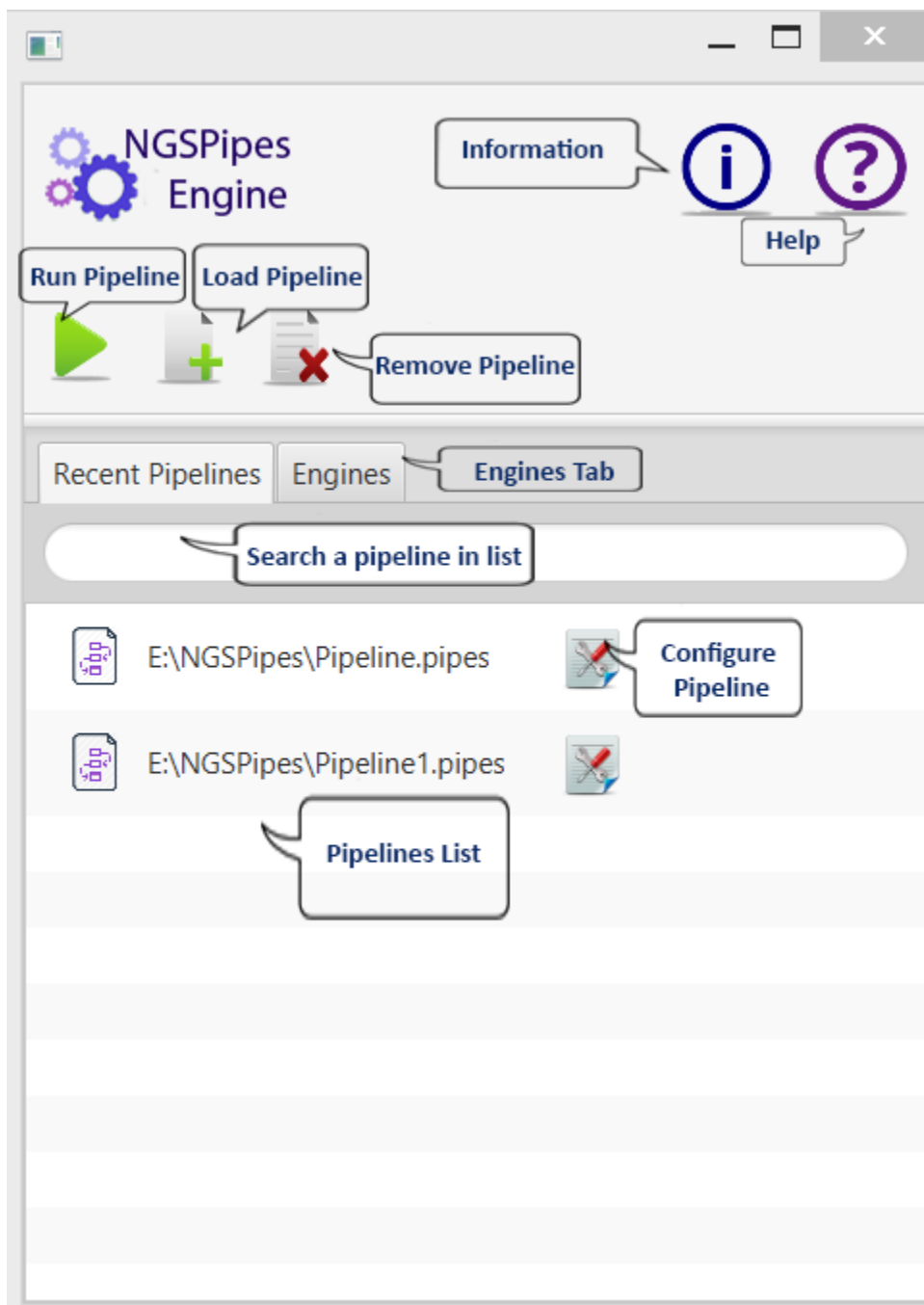


Figure 3

Notice that depending on the MAC OS version, it may be necessary to unlock to make changes and to select the option "Allow apps downloaded from Anywhere"

Screen shots

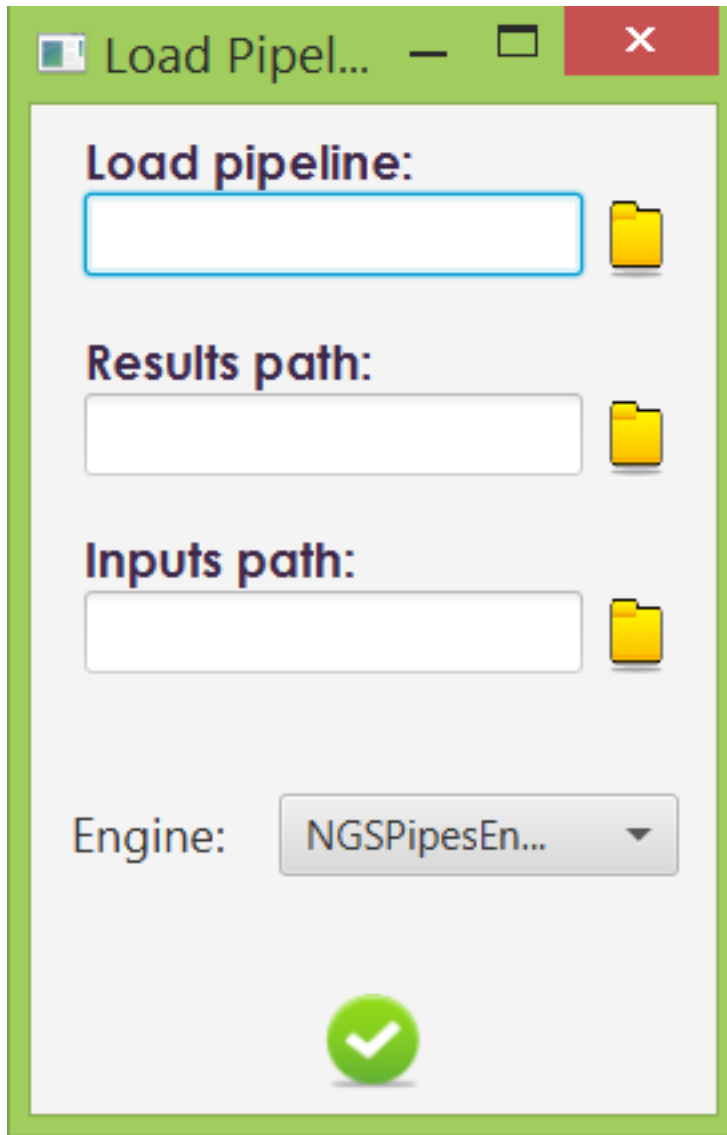
The following image shows a screenshot of the main windows and a short description of each button.



There are two main tabs: **Recent pipelines** and **Engines**. The **Recent pipelines** tab lists the last pipelines loaded by the application. It also allows the configuration of parameters for a selected pipeline. The **Engines** tab shows the previously used instances. In each engine, different tools can already be installed. The user can choose which instance to execute based on his knowledge of the pipeline.

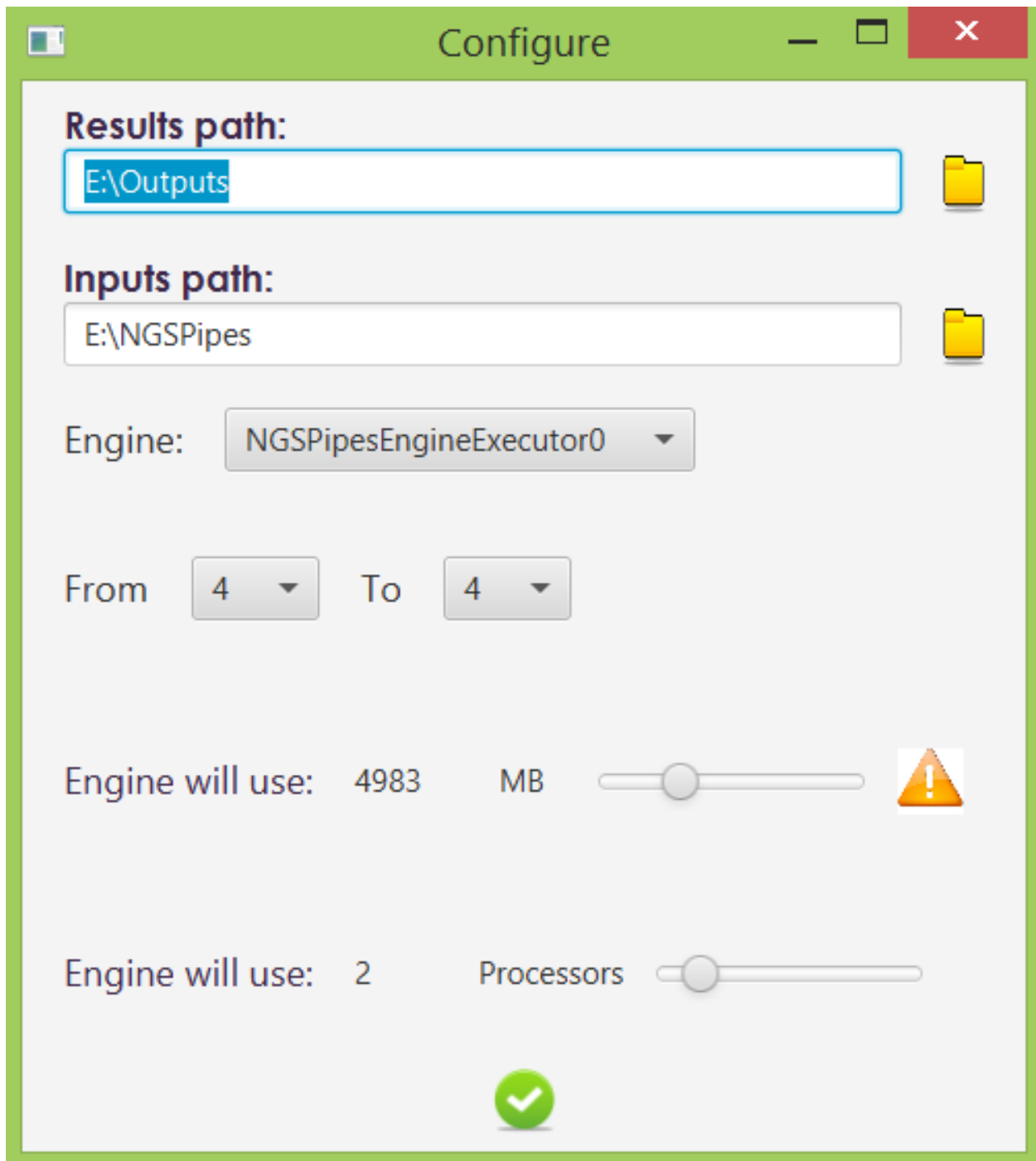
Load pipeline

When loading a pipeline the user chooses the file with the pipeline description, the directory at the user's computer where the results are to be written and the path from where the data is to be loaded. The user can also choose which engine instance to use.



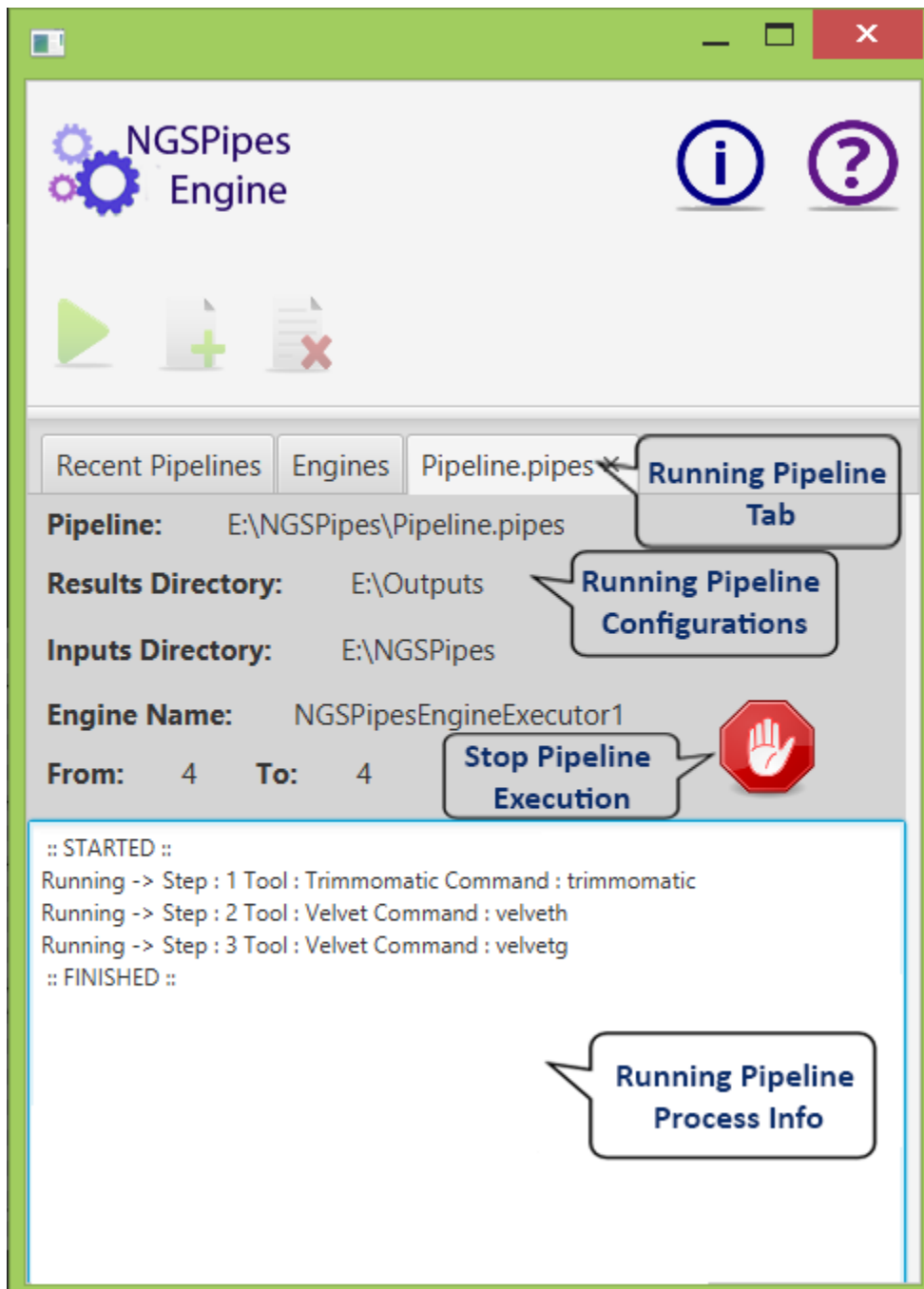
Configure pipeline

When a pipeline description is already loaded by the UI, several execution parameters can be changed: paths, engine instance, memory and number of cores.



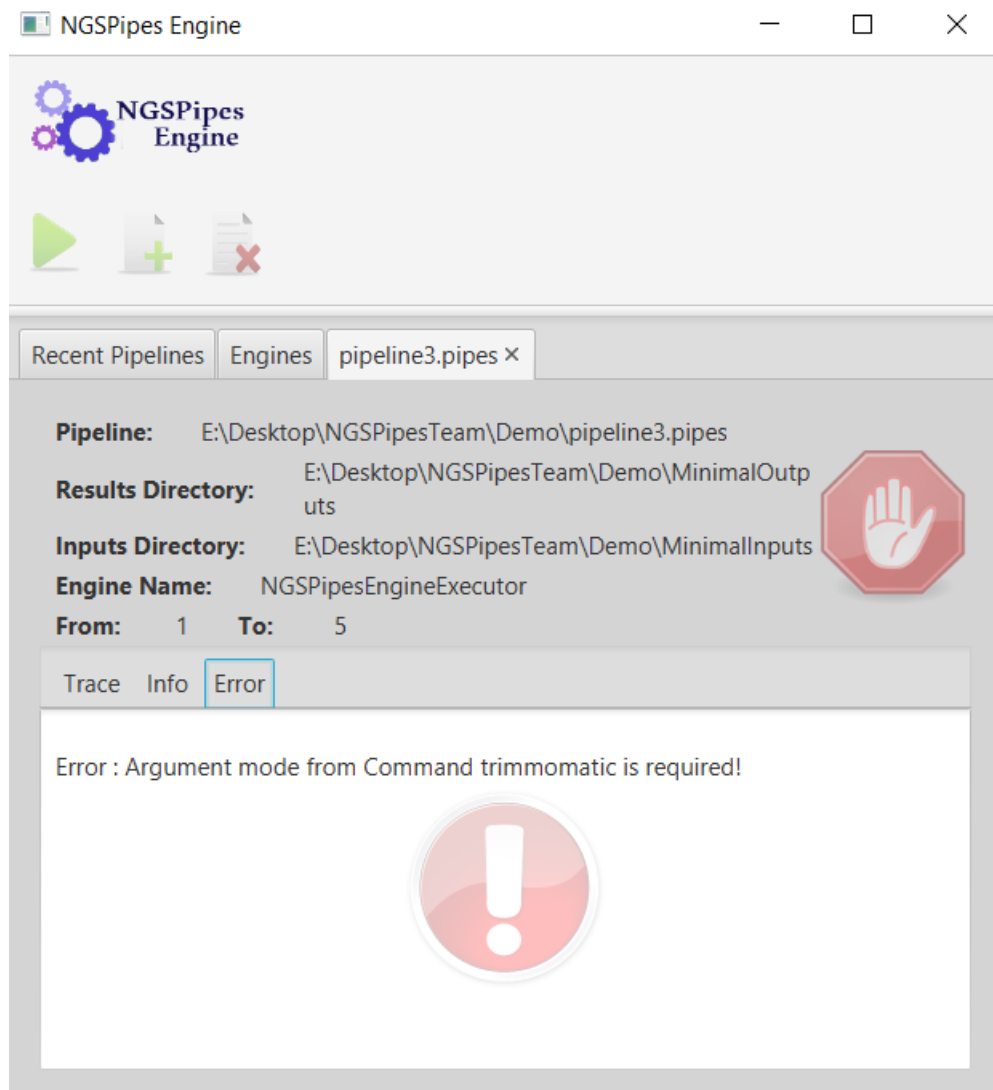
Execute pipeline

When an engine instance is selected and the “Run Pipeline” button is pressed, the UI will show the following window, where output information regarding the execution steps are presented.

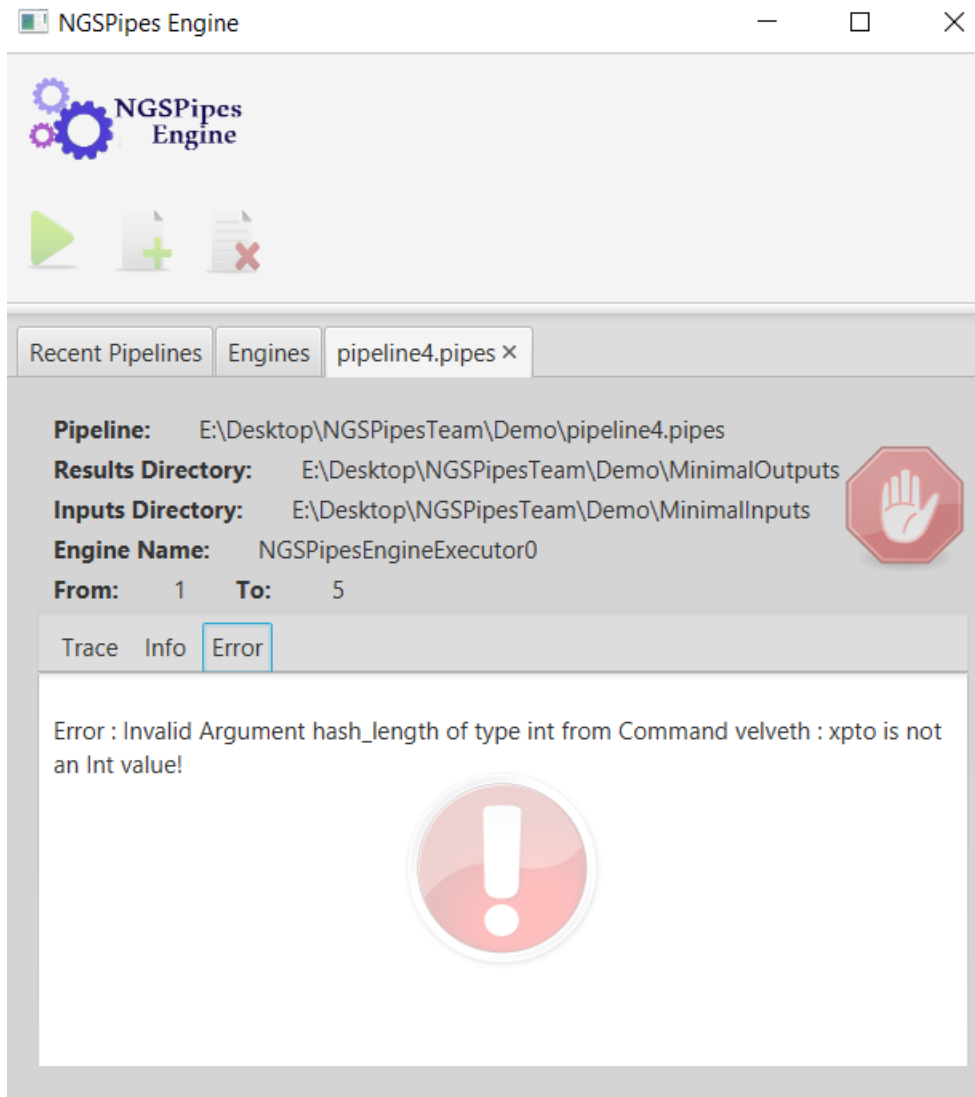


Error reporting

Errors can occur during the execution of a pipeline. For example, the next figure shows an error related to a mandatory argument that is not specified.



The next figure shows an error related to a mismatch between the type of value used in the pipeline and the type of parameter that is present in the specification of the tool.



Use case

The following use case executes the pipeline described in the [DSL section](#) using the console version of *NGSEngine*. The tools' repository used in the pipeline is <https://github.com/ngspipes/repository>. It has metadata for the tools *Trimmomatic*, *Velvet* and *Blast*.

- Check if requirements are met and that the engine and executor image are installed.
- Download the pipeline [here](#) and save it as `pipeline.pipes` to the working directory. The following examples assume the working directory is `c:\ngspipes`.
- Download the input data [sample](#) and place it at `inputs` directory (other name can be chosen). This data set comes from the [NCBI SRA](#), being part of a project on *deep sequencing within the Streptococcus pneumoniae antibiotic resistant pandemic clone PMEN1*. Extra information on how this data was obtained can be obtained [here](#).

After unzipping file `ERR406040.fastq.zip` the directory content will look like:

```
c:\ngspipes\inputs
|-- allrefs.fna.pro
|-- ERR406040.fastq
|-- NexteraPE-PE.fa
|-- TruSeq2-PE.fa
|-- TruSeq2-SE.fa
|-- TruSeq3-SE.fa
|-- TruSeq3-PE-2.fa
|-- TruSeq3-PE.fa
|-- TruSeq3-SE.fa
```

- Create the `outputs` directory (`c:\ngspipes\outputs`)
- Execute the *engine* at your working directory using the following command line:

Windows

```
c:\ngspipes>engine-1.0\bin\engine.bat -in c:\ngspipes\inputs -out
c:\ngspipes\outputs -pipes c:\ngspipes\pipeline.pipes
```

OSX/Linux

```
ngs@server:/home/ngspipes$engine-1.0/bin/engine -in /home/ngspipes/inputs -out
/home/ngspipes/outputs -pipes /home/ngspipes/pipeline.pipes
```

Example and description of output messages

Initial steps of the output will look like this:

```
Loading engine directories
Loading engine resources
Using classpath C:/Users/user/NGSPipes/Engine/dsl-1.0.jar;
                C:/Users/user/NGSPipes/Engine/repository-1.0.jar
Getting engine requirements
Getting clone engine
Clonning engine
..... Clonning engine
..... Clonning engine
..... Clonning engine
..... Clonning engine
..... Clonning engine
..... Clonning engine
Configurating engine
Starting execute engine
Booting engine and installing necessary packages
...
```

Note that the cloning step only happens in the first execution of the engine. On the other hand, when a tool is used for the first in any *pipeline*, the engine will automatically download and install the corresponding Docker image. An example of output for when this is necessary is presented for the *Trimmomatic* tool:

```
...
TRACE      :: STARTED ::
TRACE      Running -> Step : 1 Tool : Trimmomatic Command : trimmomatic
```

```

INFO      Executing : sudo docker run -v /home/ngspipes/Inputs/:/shareInputs/:rw -v
                        /home/ngspipes/Outputs/:/shareOutputs/:rw
                        ngspipes/trimmomatic0.33 java -jar trimmomatic-0.33.jar SE
                        -phred33 /shareInputs/ERR406040.fastq /shareOutputs
                        ERR406040.filtered.fastq
                        ILLUMINACLIP:/shareInputs/adapters/TruSeq3-SE.fa:2:30:10
                        SLIDINGWINDOW:4:15 LEADING:3 TRAILING:3 MINLEN:36
INFO      Unable to find image 'ngspipes/trimmomatic0.33:latest' locally
INFO      latest: Pulling from ngspipes/trimmomatic0.33
INFO      511136ea3c5a: Pulling fs layer
INFO      e977d53b9210: Pulling fs layer
INFO      c9fa20ecce88: Pulling fs layer
...
INFO      6cf3f4911f80: Download complete
INFO      Digest:
↳ sha256:44f1dea760903cdce1d75c4c9b2bd37803be2e0fbbb9e960cd8ff27048cbb997
INFO      Status: Downloaded newer image for ngspipes/trimmomatic0.33:latest
INFO      TrimmomaticSE: Started with arguments: -phred33 /shareInputs/ERR406040.fastq
                        / shareOutputs/ERR406040.filtered.fastq
                        ILLUMINACLIP:/shareInputs/adapters/TruSeq3-SE.fa:2:30:10
                        SLIDINGWINDOW:4:15 LEADING:3 TRAILING:3 MINLEN:36
...

```

Note that this tool was previously *dockerized* by the NGSPipes team. For other tools, such as Velvet or Blast, there is already public Docker images which the example pipeline uses.

When the execution finish, the following files will be at the working directory:

```

c:\ngspipes\outputs
|-- allrefs.phr
|-- allrefs.pin
|-- allrefs.psq
|-- blast.out
|-- filtered.fastq
|-- velvetdir/
    |-- Log
    |-- Roadmaps
    |-- Sequences
    |-- contigs.fa
    |-- LastGrpah
    |-- stats.txt

```

Execution times

The above example was executed using several hardware configurations and operating systems. The pipeline was executed with the command line:

```

ngs@server:/home/ngspipes$ engine-1.0/bin/engine -in /home/ngspipes/inputs
                        -out /home/ngspipes/outputs -pipes /home/ngspipes/pipeline.
↳ pipes

```

The following table shows execution times measured on *cold* and *warm* start situations. Cold start happens when the engine is executed the first time after installation. Warm start represents the situation when a pipeline is being re-executed and no updates are necessary to the tools.

OS	CPU	RAM (GB)	Disk	Cold start	Warm start
Windows 10	Intel i5 @ 2.53 Ghz	8	SSD	39 min.	35 min.
Windows 10	Intel i7 @ 3.5 Ghz	16	HDD		

39 min. | 30 min. | | OSX | Intel(TM) i5 1.8Ghz | 8 | SSD | 41 min. | 38 min. |

(*) <http://www.speedtest.net/>

As expected, a cold start takes an extra time because of initial setup and download of the tools used in the [pipeline](#). Warm start is the common execution scenario. These values can vary depending on:

- the size of the input data;
- the number of tools and commands used in pipeline;
- the input data and resources assigned to the execution image (CPU (`-cpus`) and memory (`-mem`)).

Instructions to build NGS Pipes Engine for workstation from source code

Requirements

No specific tools must be installed to build the *NGSPipes engine for workstation*. The code is available at git hub repository, which can be downloaded as a zip.

The source code repository can also be *cloned*. In the last case, the [git version control tool](#) must be installed first.

Build commands

To build the *NGSPipes engine for workstation* follow these steps:

- Download the zip or clone the [git repo](#) to your working directory. The following command will build all the components – [DSL](#), [Tools repository](#) and the [Engine](#) (both console and UI version).
 - `cd main`
 - `git submodule init`
 - `git submodule update`
 - `gradlew build` (This will install [Gradle](#), if necessary)
- To generate the `engine-1.0.zip` and `editor-1.0.zip` distributions files, run `gradlew distZip`, at `main` directory. The files will be located at the respective `build/distributions` directory.

Engine for cloud

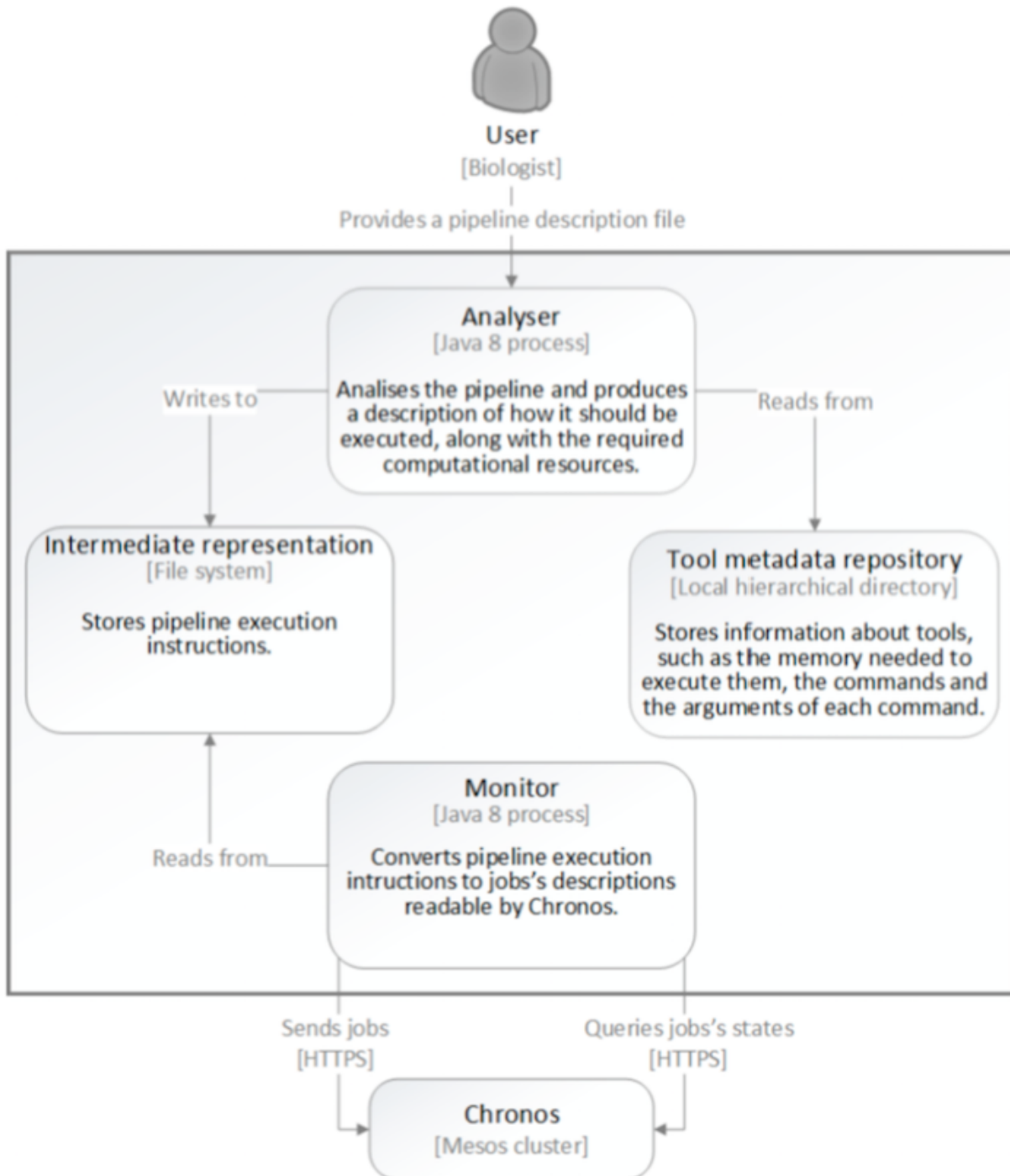
The Engine for cloud solution consists of two tools: the **analyser** and the **monitor**.

The **analyser** inspects the given pipeline and, using the information stored in the tool meta-data repository, produces the instructions to execute the pipeline, along with the required computational resources for its execution. These tool produces internally a graph of tasks, which reflects the dependency among the tasks and allows to infer what can be executed in parallel and what can only be executed in serie. These instructions are then written to a file and locally stored, giving origin to an intermediate representation. The analyser tool can be simple executed in a workstation or in a cloud environment.

Given the pipeline representation produced by the Analyser tool, the **monitor** can be launched. The monitor tool is suitable for running in a cloud environment. It converts the intermediate representation into jobs' descriptions readable by [Chronos](#) and, using Chronos's REST API, schedules them for execution. The tasks are deployed in a **cluster of machines** governed by the [Mesos](#) batch job scheduling framework Chronos. Having launched the pipeline, the user can now query the system to know its current state. This results in a series of requests from the monitor to Chronos. When the pipeline finishes the execution, the user can make a request to the monitor to download its outputs.

We provide a **jar file with the analyser tool** and a **virtual machine with the monitor tool** for experimental purposes, where users can simulate a cluster. With this machine, pipelines can be tested without requiring a big amount of computational resources or an account in a cloud provider. Thus our solution can be tested in this virtual machine or within a cloud provider, as described in the subsection “Install engine for cloud”.

Next figure describes the main components of this engine and their interactions. More details on this engine can be found in this [report](#).



Requirements to run the engine for cloud

To run the analyser tool you will need:

- to have installed [JRE 8](#).

To run the **monitor (within the virtual machine that we supply for testing)** you will need:

- to have installed [JRE 8](#).
- virtualization software which supports vmdk files, like [VMware](#) or [VirtualBox](#).
- To emulate the image is required 8GB of RAM and 1 CPU

To run the **monitor (without the virtual machine that we supply for testing)** you will need:

- a cluster with Chronos installed with the following requirements:
 - an endpoint to Chronos REST API;
 - Support for the execution of Docker jobs on all Mesos-Agents (Slaves);
 - A NFS accessible on all Mesos-Agents
 - SSH access to a cluster machine that can interact with the cluster's NFS;

Install the engine for cloud

Install the analyser

To deploy this in your system:

- Create a new directory named Analyser and download the [executable](#) to there.
- After the installation, you should have the following tree file:

```
WorkingDirectory
|-- Analyser\
    |-- ngs4cloud-analyser-1.0-SNAPSHOT\
        |-- bin
            |--ngs4cloud-analyser
            |--ngs4cloud-analyser.bat (CUI Window run script)
    |-- Monitor\
        |-- monitor.jar
    |-- (other files,...)
```

Install the monitor

- To deploy this in your system:
- Create a new directory named Monitor and download the [jar](#) to there.

Normally to execute [monitor.jar](#) one would need a cluster with the specifications mentioned above. **However to facilitate executing, for testing purposes, we provide a [Virtual Image](#) which emulates an appropriate cluster to execute [monitor.jar](#).** In a cloud environment, the setting will be similar, with the corresponding credentials.

- Emulate the image of the cluster using a virtualization software which supports vmdk files, like [VMware](#) or [VirtualBox](#). To emulate the image **is required 8GB of RAM and 1 CPU**.
- In order to emulate the image (**steps in Virtual Box**), for instance, we should:
 - Select the option New Virtual Machine. We can choose, for instance, Debian.

- Select the option Using an existing virtual hard disk file.
- Select the option Create (for creating a virtual machine).
- After creating, it is necessary to configure SSH between The host system and virtual box guest.
- Then, add a NAT Network in Virtual Box menu, namely, Virtual Box --> Preferences.
- Then, add vboxnet0 in Virtual Box menu, namely, Virtual Box --> Preferences.
- Then, select the created virtual machine and select this settings. In its settings:
 - * Check if the IO APIC is enabled, namely in Settings-->System;
 - * Select to attach to HostOnlyAdapter the name vboxnet0, namely in Settings-->Adapter 2
 - * Select to attach NAT Network the name NAT Network, namely in Settings-->Adapter 1
- Launch (Start) the created virtual machine.
- After launching the virtual machine, wait for the graphical environment and log in the desktop using the following credentials:

```
User: ngs4cloud
Pass: cloud123
```

- Open a terminal and type the command

```
/sbin/ifconfig
```

and get the ip of Virtual Machine.

- If necessary, you may have to configure the ip of the Virtual Machine. For example `sudo ifconfig eth1 192.168.56.2`
- Now back to the host OS. If you try to execute the monitor the following message will be shown +

```
"The environment variable NGS4_CLOUD_MONITOR_CONFIGS
needs to be defined with the path of the configuration file"
```

- Therefore we need to first setup the configurations so that we can execute the monitor. In same directory where `monitor.jar` is, create a file named `configs` and open it. Now write on the file the following configurations:

```
SSH_HOST = "The ip of the virtual machine"
SSH_PORT = 22
SSH_USER = ngs4cloud
SSH_PASS = cloud123
CHRONOS_HOST = "The ip of the virtual machine"
CHRONOS_PORT = 4400
PIPELINE_OWNER = example@example.com
CLUSTER_SHARED_DIR_PATH = /home/ngs4cloud/pipes
WGET_DOCKER_IMAGE = jnforja/wget
P7ZIP_DOCKER_IMAGE = jnforja/7zip
```

- An example of the `configs.txt` is

```
SSH_HOST = 192.168.56.2
SSH_PORT = 22
SSH_USER = ngs4cloud
SSH_PASS = cloud123
CHRONOS_HOST = 192.168.56.2
```

```
CHRONOS_PORT = 4400
PIPELINE_OWNER = example@example.com
CLUSTER_SHARED_DIR_PATH = /home/ngs4cloud/pipes
WGET_DOCKER_IMAGE = jnforja/wget
P7ZIP_DOCKER_IMAGE = jnforja/7zip
```

- After defining the `config.txt` file, it is necessary to set the environment variable.
- For instance, in MAC OS, for setting the environment variable is to open a terminal in the current Monitor directory and then do `export NGS4_CLOUD_MONITOR_CONFIGS=configs.txt` if `config.txt` is also in the Monitor directory.
- In the monitor directory create another directory called `repo` and add the following entry to the configs file:

```
EXECUTION_TRACKER_REPO_PATH = "The repo directory path"
```

- As an example, configs file should look like:

```
SSH_HOST = 192.168.56.2
SSH_PORT = 22
SSH_USER = ngs4cloud
SSH_PASS = cloud123
CHRONOS_HOST = 192.168.56.2
CHRONOS_PORT = 4400
PIPELINE_OWNER = example@example.com
CLUSTER_SHARED_DIR_PATH = /home/ngs4cloud/pipes
WGET_DOCKER_IMAGE = jnforja/wget
P7ZIP_DOCKER_IMAGE = jnforja/7zip
EXECUTION_TRACKER_REPO_PATH = /Users/cvaz/Documents/pipelines/Monitor/repo
```

Save the `configs` file and close it. Now to finish configuring the monitor just add a environment variable named `NGS4_CLOUD_MONITOR_CONFIGS` with the path of the configs file as value.

Open a terminal and execute **monitor.jar**, a message explaining all the commands the monitor can execute should appear.

Run the Engine for cloud

Run the Analyser

This a regular Java application package as a JAR file. To run, open a terminal and execute the command **analyse** at the working directory.

```
user@machine:/home/workingDirectory$ ./bin/ngs4cloud-analyser analyse
analyse <mandatory arguments>
```

Parameters

The command line tool has the following *mandatory* parameters:

- `-pipes` Relative ou absolute path of the pipeline description (mandatory). This file must be a `.pipes` extension file, where the pipeline is written using the NGSPipes language.
- `-ir` Intermediate representation file produced by the analyser. This filepath should be provided by the user.
- This file will be given as input to the monitor tool.
- `-input` The URI with the location of the inputs for the pipeline.

- `-outputs $space_separated_list_of_outputs`.

In this current version of the Analyser tool, the input must always be an URI. And since usually we have more than one input for each pipeline, the same can be zipped. If the input is not an URI already, we can create a shared link in dropbox, for instance, and use that URI.

Example

Here is an example:

```
./bin/ngs4cloud-analyser analyse -pipes ./pipelines/pipeline.pipes
      -ir ./ir/ir.json
      -input https://github.com/CFSAN-Biostatistics/snp-pipeline/archive/
↪master.zip
      -outputs snp-pipeline-master/snppipeline/data/lambdaVirusInputs/snplist.
↪txt
      snp-pipeline-master/snppipeline/data/lambdaVirusInputs/snpma.
↪fasta
```

This will analyse and process the file `-pipes`, store the IR in the file `-ir`, set the input and outputs for `-ir` and `-output`.

To execute the pipeline on a simulated environment, see subsection . . .

Run the monitor

In this section, we describe how to **run the monitor with the virtual image provided for testing**, after setting up the virtual imagem, as explained before. In a cloud environment, the setting will be similar, with the corresponding credentials.

Open a terminal and execute **monitor.jar**, a message explaining all the commands the monitor can execute should appear. The application monitor is a regular Java application package as a JAR file. To run, open a terminal and execute the command **monitor** at the working directory.

This application has 3 *commands*:

- `launch`, to launch the execution of a pipeline. This command is followed with the filepath of a IR file (a file produced by the analyser tool).
- This command returns an integer, which is the id of the launched pipeline.
- `status`, to check the state of the pipeline execution. This command is followed by the ID returned by a execution of the previous command
- `outputs`, to download the outputs of a pipeline execution after it has finished. This command is followed by the ID of the pipeline. Notice that this will only download the outputs that were previously specified by the analyser tool to be downloaded for this pipeline.

Example

Here is an example:

```
"java -jar monitor.jar launch ir.json"
```

The pipeline represented in the file `ir.json`, which in this example is assumed to be in the same directory as the `monitor.jar`, is launched for execution. You will now see in the console messages regarding the upload of the input file. Once the upload is finished a message like this will appear `"ID: 1"`, this is the ID attributed to the launched pipeline.

To check the state of the pipeline execution type the following command:

```
"java -jar monitor.jar status 1"
```

notice that “1” is the id of pipeline, which was given in the previous step. If the pipeline has finished executing this message will appear

```
"The pipeline execution has finished with success."
```

When the pipeline execution has finished type the following commands to download its outputs:

```
"java -jar monitor.jar outputs 1 ."
```

This will download the outputs of the pipeline to the directory where the monitor is being executed.

Here is a video for demonstrating the previous steps:

Running Examples

A pipeline used on epidemiological surveillance

In this section we present a pipeline used on epidemiological surveillance. The aim is to characterize bacterial strains through allelic profiles. When sequencing a bacterial strain by paired end methods with desired depth of coverage of 100x (in average each position in the genome will be covered by 100 reads), the output from the sequencer will be two FASTQ files containing the reads. Each read typically will have 90-250 nucleotides length, using Illumina technology. The first data processing step is to trim the reads for removing the adapters used in the sequencing process and any tags used to identify the experiment in a run.

In de novo assembly, software such as Velvet is used to obtain a draft genome composed of contigs, longer DNA sequences resulting from assembling multiple reads. The draft genome can be compared to databases of gene alleles for multiple loci using BLAST. Given BLAST results we can create an allelic profile characterizing the strain.

```
Pipeline "Github" "https://github.com/ngspipes/tools" {
  tool "Trimmomatic" "DockerConfig" {
    command "trimmomatic" {
      argument "mode" "SE"
      argument "quality" "-phred33"
      argument "inputFile" "study1/ERR406040.fastq"
      argument "outputFile" "ERR406040.filtered.fastq"
      argument "fastaWithAdaptersEtc" "study1/TruSeq3-SE.fa"
      argument "seed mismatches" "2"
      argument "palindrome clip threshold" "30"
      argument "simple clip threshold" "10"
      argument "windowSize" "4"
      argument "requiredQuality" "15"
      argument "leading quality" "3"
      argument "trailing quality" "3"
      argument "minlen length" "36"
    }
  }
}
tool "Velvet" "DockerConfig" {
  command "velveth" {
```

```
    argument "output_directory" "velvetdir"
    argument "hash_length" "21"
    argument "file_format" "-fastq"
    chain "filename" "outputFile"
  }
  command "velvetg" {
    argument "output_directory" "velvetdir"
    argument "-cov_cutoff" "5"
  }
}
tool "Blast" "DockerConfig" {
  command "makeblastdb" {
    argument "-dbtype" "prot"
    argument "-out" "allrefs"
    argument "-title" "allrefs"
    argument "-in" "study1/allrefs.fna.pro"
  }
  command "blastx" {
    chain "-db" "-out"
    chain "-query" "Velvet" "velvetg" "contigs_fa"
    argument "-out" "blast.out"
  }
}
}
```

Example 6.1: A pipeline used on epidemiological surveillance.

A visual representation of this pipeline described in Example 6.1 is presented in the Figure 6.1. Moreover, in this figure is also possible to observe other execution orders that are feasible to execute this pipeline in the engine for workstation.

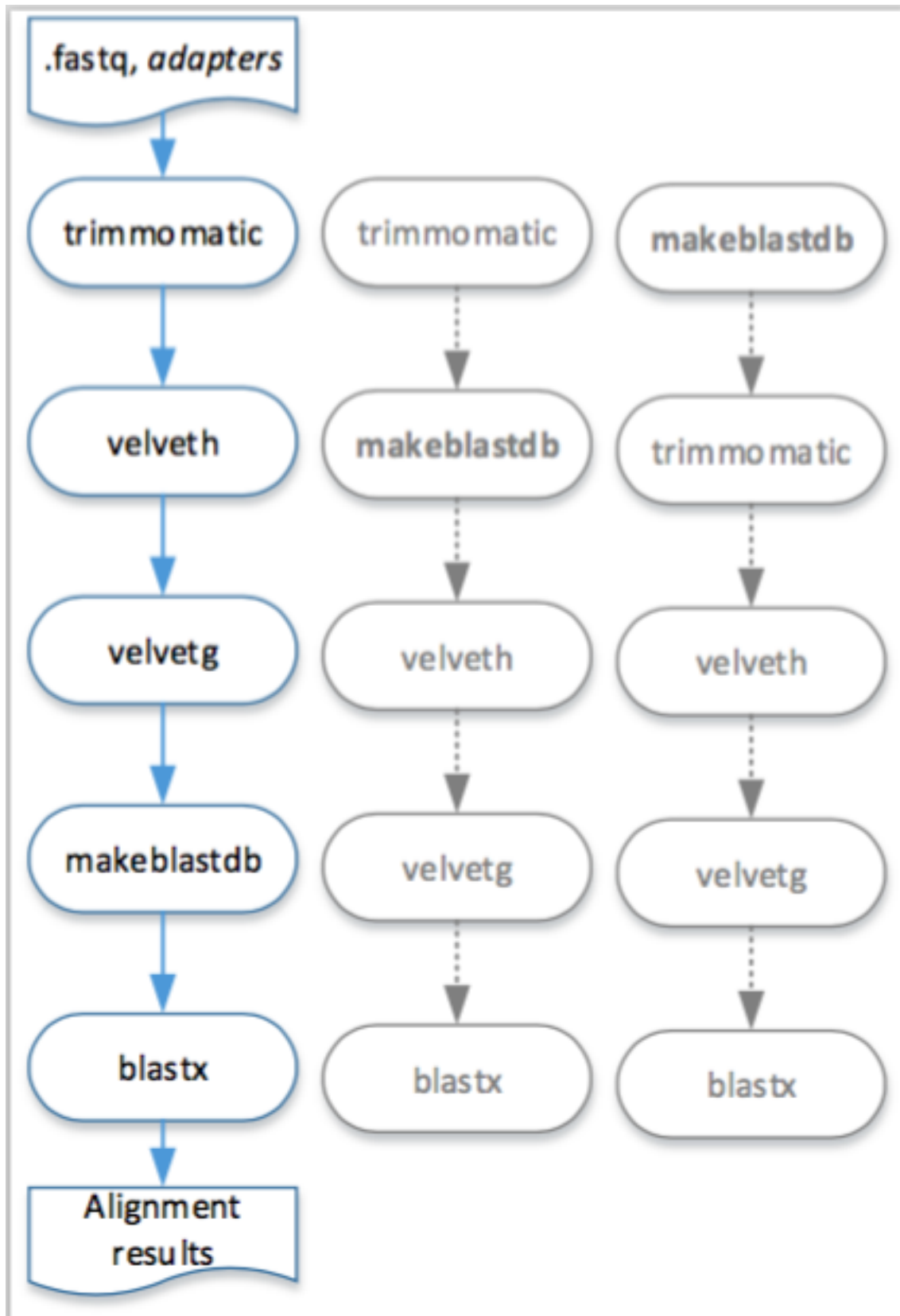
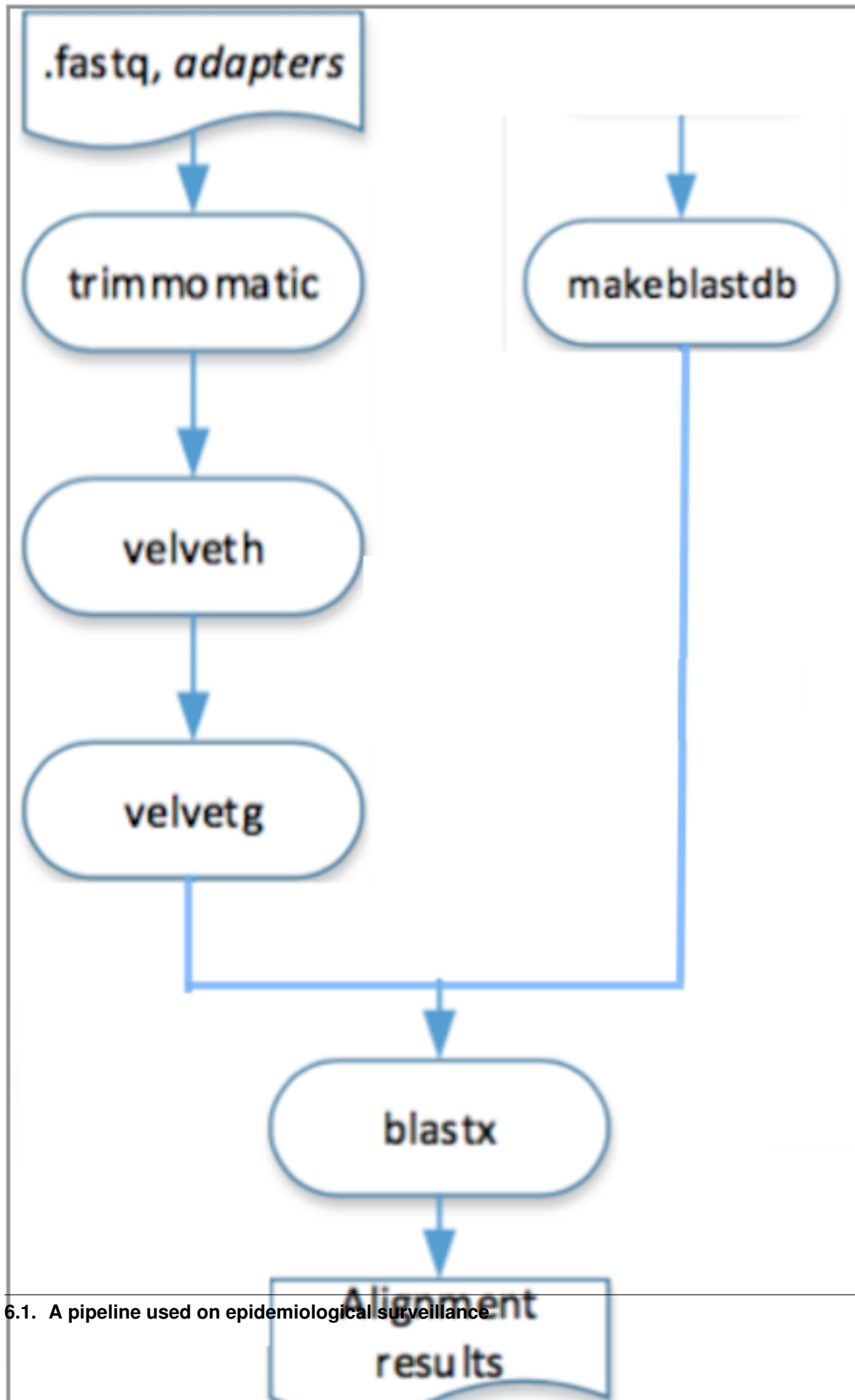


Figure 6.1: Visual representation of the execution, in the engine for workstation, of the pipeline described in

Example 6.1.

In the engine for cloud, different steps of the pipeline can be executed in different machines, it is only necessary to respect its dependencies, as it is shown in the Figure 2.2.



6.1. A pipeline used on epidemiological surveillance

Figure 6.2: Visual representation of the execution, in the engine for cloud, of the pipeline described in Example 6.1.

Input data is available [here](#)

Running this example in Engine for workstation

Note Please, be sure that the Engine for Workstation is already installed. For this, follow the steps that are in section:

Engine->Engine for Workstation-> Install engine for workstation.

Since the engine for workstation is provided as a console application or a graphical user interface application, we will describe how to do with the console application (for more information on how to user the graphical user interface, please look at the section: Engine->Engine for Workstation-> Run engine for workstation.

- After the installation, you should have the following tree file:

```
WD
|-- engine-1.0\
    |-- NGSPipesEngineExecutor\
        |-- NGSPipesEngineExecutor.vbox
        |-- NGSPipesEngineExecutor.vdi
    |-- bin\
        |-- engine          (CUI OSX/Linux run script)
        |-- engine.bat      (CUI Window run script)
        |-- engine-ui       (GUI OSX/Linux run script)
        |-- engine-ui.bat   (GUI Window run script)
    |-- lib\
        |-- ...
    |-- (other files, ...)
```

- Download the data available [here](#)
- After unzipping, the directory content look like, for instance,

```
/home/ngspipes/study1
|-- allrefs.fna.pro
|-- ERR406040.fastq
|-- NexteraPE-PE.fa
|-- TruSeq2-PE.fa
|-- TruSeq2-SE.fa
|-- TruSeq3-SE.fa
|-- TruSeq3-PE-2.fa
|-- TruSeq3-PE.fa
|-- TruSeq3-SE.fa
```

- Create a file `casestudy1.pipes` (.pipes is the extension containing the pipeline previously described in Figure 6.1. Assume that, on the following `casestudy1.pipes` is inside the directory `study1`.
- Create the `outputs` directory (`/home/ngspipes/outputs`)
- Execute the *engine* at your working directory using the following command line:

Windows

```
c:\ngspipes>engine-1.0\bin\engine.bat -in c:\ngspipes\study1 -out
c:\ngspipes\outputs -pipes c:\ngspipes\casestudy1.pipes
```

OSX/Linux

```
ngs@server:/home/ngspipes$engine-1.0/bin/engine -in /home/ngspipes/inputs -out
/home/ngspipes/outputs -pipes /home/ngspipes/casestudy1.pipes
```

Example and description of output messages

Initial steps of the output will look like this:

```
Loading engine directories
Loading engine resources
Using classpath C:/Users/user/NGSPipes/Engine/dsl-1.0.jar;
                C:/Users/user/NGSPipes/Engine/repository-1.0.jar
Getting engine requirements
Getting clone engine
Clonning engine
..... Clonning engine
..... Clonning engine
..... Clonning engine
..... Clonning engine
..... Clonning engine
..... Clonning engine
Configurating engine
Starting execute engine
Booting engine and installing necessary packages
...
```

Note that the cloning step only happens in the first execution of the engine. On the other hand, when a tool is used for the first in any *pipeline*, the engine will automatically download and install the corresponding Docker image. An example of output for when this is necessary is presented for the *Trimmomatic* tool:

```
...
TRACE    :: STARTED ::
TRACE    Running -> Step : 1 Tool : Trimmomatic Command : trimmomatic
INFO     Executing : sudo docker run -v /home/ngspipes/Inputs:/shareInputs:rw -v
                /home/ngspipes/Outputs:/shareOutputs:rw
                ngspipes/trimmomatic0.33 java -jar trimmomatic-0.33.jar SE
                -phred33 /shareInputs/ERR406040.fastq /shareOutputs
                ERR406040.filtered.fastq
                ILLUMINACLIP:/shareInputs/adapters/TruSeq3-SE.fa:2:30:10
                SLIDINGWINDOW:4:15 LEADING:3 TRAILING:3 MINLEN:36
INFO     Unable to find image 'ngspipes/trimmomatic0.33:latest' locally
INFO     latest: Pulling from ngspipes/trimmomatic0.33
INFO     511136ea3c5a: Pulling fs layer
INFO     e977d53b9210: Pulling fs layer
INFO     c9fa20ecce88: Pulling fs layer
...
INFO     6cf3f4911f80: Download complete
INFO     Digest:
↪sha256:44f1dea760903cdce1d75c4c9b2bd37803be2e0fbbb9e960cd8ff27048cbb997
INFO     Status: Downloaded newer image for ngspipes/trimmomatic0.33:latest
INFO     TrimmomaticSE: Started with arguments: -phred33 /shareInputs/ERR406040.fastq
                / shareOutputs/ERR406040.filtered.fastq
                ILLUMINACLIP:/shareInputs/adapters/TruSeq3-SE.fa:2:30:10
                SLIDINGWINDOW:4:15 LEADING:3 TRAILING:3 MINLEN:36
...
```

Note that this tool was previously *dockerized* by the NGSPipes team. For other tools, such as Velvet or Blast, there is already public Docker images which the example pipeline uses.

When the execution finish, the following files will be at the working directory:

```
home/ngspipes/outputs
|-- allrefs.phr
|-- allrefs.pin
|-- allrefs.psq
|-- blast.out
|-- filtered.fastq
|-- velvetdir/
    |-- Log
    |-- Roadmaps
    |-- Sequences
    |-- contigs.fa
    |-- LastGrpah
    |-- stats.txt
```

Running this example in Engine for Cloud

Note Please, be sure that the Engine for Cloud is already installed. For this, follow the steps that are in section:

Engine->Engine for Cloud-> Install engine for cloud.

If previously installed, please ensure that:

- the ip of the virtual machine is configured
- the environment variable is stablished on the terminal that you are executing the monitor. For managing these settings, please also consult the section:

Engine->Engine for Cloud-> Install engine for cloud.

After the installation, you should have the following tree file:

```
WorkingDirectory
|-- Analyser\
    |-- ngs4cloud-analyser-1.0-SNAPSHOT\
        |-- bin
            |--ngs4cloud-analyser
            |--ngs4cloud-analyser.bat (CUI Window run script)
    |-- Monitor\
        |-- monitor.jar
    |-- (other files,...)
```

- **Input data is available [here](#)**, but is not necessary to download. Input data in Engine for Cloud engine is always passed as an URI.
- Create a file `casestudy1.pipes(.pipesis the extension containing the pipeline previously described in Figure 6.1. Assume that, on the following, casestudy1.pipes is inside the directory ngs4cloud-analyser-1.0-SNAPSHOT.`
- Start by execution the analyser tool, in order to produce an file with `json`extension.

OSX/Linux

```
ngs@server:ngs4cloud-analyser-1.0-SNAPSHOT$ ./bin/ngs4cloud-analyser analyse
-pipes casestudy1.pipes
-ir irl.json
-input https://www.dropbox.com/s/h8e8t3prt9f0gq3/study1.zip?dl=0
-outputs blast.out velvetdir/contigs.fa
```

- This execution will produce the file `irl.json`.
- Then, copy the `irl.json` inside to directory `Monitor`
- Before executing the `Monitor`, please assure that the Virtual Machine with the cluster image given for test purposes is launched and correctly settled (please, see the section

```
Engine->Engine for cloud->
  Install the engine for cloud -> Install the monitor
```

- Launch the pipeline into the cluster through the `monitor` command

```
ngs@server:Monitor$ java -jar monitor.jar launch irl.json
```

- The previous command will generate a pipeline id. Assume in this example that the id is 1.
- Consult the status of the pipeline by its id

```
ngs@server:Monitor$ java -jar monitor.jar status 1
```

- After pipeline is finished, it is possible to download its results from the cluster to a previously defined directory inside the `Monitor` directory.

```
ngs@server:Monitor$ java -jar monitor.jar outputs 1 resultsDirectory
```

- `resultsDirectory` is the directory that contains a copy of the outputs that were previously specified by the analyser that should be copied; 1 is the pipeline id

For more information about the `analyser` and `monitor` commands and its parameters, please see section

```
Engine->Engine for cloud->Run the engine for cloud
```

A pipeline used on ChIP-Seq analysis

In this section we present a pipeline used on ChIP-Seq analysis. This pipeline includes mapping with `bowtie2`, converting the output to `bam` format, sorting the `bam` file, creating a `bam` index file, running `flagstat` command, and removing duplicates with `picard`. So, this pipeline can be used in a ChIP-Seq pipeline that uses the resulting `bam` file for peak calling and creating heatmaps. Since those steps are generic that can be used for ATAC-Seq analysis too.

```
Pipeline "Github" "https://github.com/ngspipes/tools" {
  tool "Bowtie2" "DockerConfig" {
    command "bowtie2-build" {
      argument "reference_in" "study2/sequence.fasta"
      argument "bt2_base" "sequence"
    }
  }
  tool "Bowtie2" "DockerConfig" {
    command "bowtie2" {
```

```
        argument "-U" "study2/SRR386886.fastq"
        argument "-x" "sequence"
        argument "--trim3" "1"
        argument "-S" "eg2.sam"
    }
}
tool "Samtools" "DockerConfig" {
    command "view" {
        argument "-b" "NA"
        argument "-o" "eg2.bam"
        chain "input" "-S"
    }
}
tool "Samtools" "DockerConfig" {
    command "sort" {
        argument "-o" "eg2.sorted.bam"
        chain "input" "-o"
    }
}
tool "Picard" "DockerConfig" {
    command "MarkDuplicates" {
        chain "INPUT" "-o"
        argument "OUTPUT" "marked_duplicates.bam"
        argument "REMOVE_DUPLICATES" "true"
        argument "METRICS_FILE" "metrics.txt"
    }
}
}
```

Example 6.2: A pipeline used on ChIP-Seq analysis.

A visual representation of this pipeline is presented in the next figure.

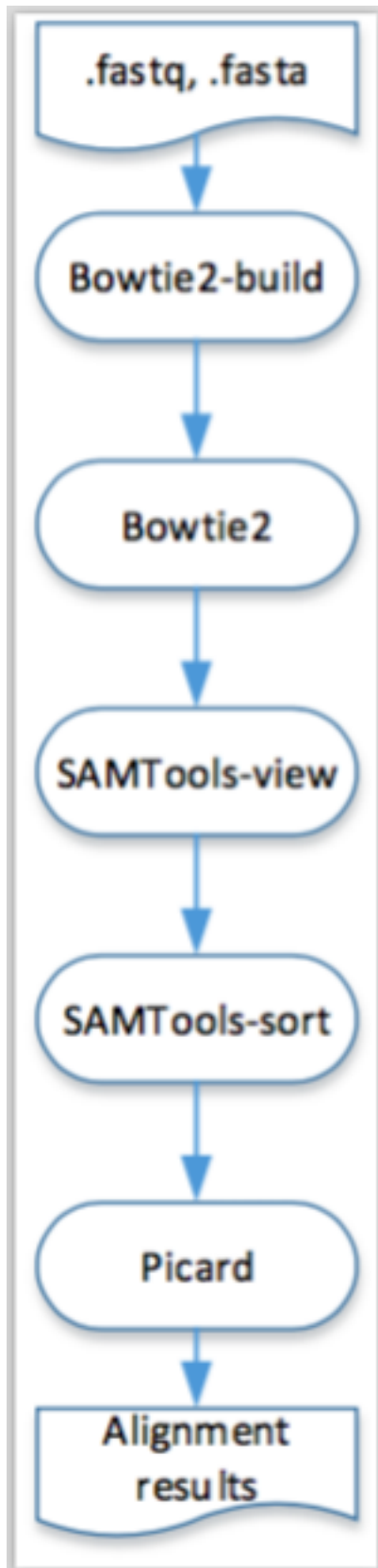


Figure 6.2: Visual representation of the execution, in both engines, of the pipeline described in Example 6.2.

Running this example in Engine for workstation

Similar to the previous example.

Running this example in Engine for Cloud

It is similar to the previous example.

Note Please, be sure that the Engine for Cloud is already installed. For this, follow the steps that are in section:

Engine->Engine for Cloud-> Install engine for cloud.

If previously installed, please ensure that:

- the ip of the virtual machine is configured
- the environment variable is stablished on the terminal that you are executing the monitor. For managing these settings, please also consult the section:

Engine->Engine for Cloud-> Install engine for cloud.

After the installation, you should have the following tree file:

```
WorkingDirectory
|-- Analyser\
    |-- ngs4cloud-analyser-1.0-SNAPSHOT\
        |-- bin
            |--ngs4cloud-analyser
            |--ngs4cloud-analyser.bat (CUI Window run script)
    |-- Monitor\
        |-- monitor.jar
    |-- (other files,...)
```

- **Input data is available [here](#)**, but is not necessary to download. Input data in Engine for Cloud engine is always passed as an URI.
- Create a file `casestudy2.pipes(.pipes` is the extension containing the pipeline previously described in Figure 6.2. Assume that, on the following, `casestudy2.pipes` is inside the directory `ngs4cloud-analyser-1.0-SNAPSHOT`.
- Start by execution the analyser tool, in order to produce an file with `json` extension.

OSX/Linux

```
ngs@server:ngs4cloud-analyser-1.0-SNAPSHOT$ ./bin/ngs4cloud-analyser analyse
-pipes casestudy2.pipes
-ir ir2.json
-input https://www.dropbox.com/s/filps3qavvhjta7/study2.zip?dl=0
-outputs metrics.txt
```

- This execution will produce the file `ir2.json`.
- Then, copy the `ir2.json` inside to directory `Monitor`
- Before executing the Monitor, please assure that the Virtual Machine with the cluster image given for test purposes is lauched and correctly settled (please, see the section

```
Engine->Engine for cloud->
    Install the engine for cloud -> Install the monitor
```


- Launch the pipeline into the cluster through the monitor command

```
ngs@server:Monitor$ java -jar monitor.jar launch ir2.json
```

- The previous command will generate a pipeline id. Assume in this example that the id is 2.
- Consult the status of the pipeline by its id

```
ngs@server:Monitor$ java -jar monitor.jar status 2
```

- After pipeline is finished, it is possible to download its results from the cluster to a previously defined directory inside the Monitor directory.

```
ngs@server:Monitor$ java -jar monitor.jar outputs 2 resultsDirectory2
```

- `resultsDirectory` is the directory that contains a copy of the outputs that were previously specified by the analyser that should be copied; 2 is the pipeline id

For more information about the `analyser` and `monitor` commands and its parameters, please see section

```
Engine->Engine for cloud->Run the engine for cloud
```

A pipeline using listing tools

A specific use of NGS data in public health is the determination of the relationship between samples potentially associated with a foodborne pathogen outbreak. This relationship can be determined from the phylogenetic analysis of a DNA sequence alignment containing only variable positions, which we refer to as a SNP matrix. The applications of such a matrix include inferring a phylogeny for systematic studies and determining within traceback investigations whether a clinical sample is significantly different from environmental/product samples.

This case study is a pipeline which combines all the steps necessary to construct a reference-based SNP matrix from an NGS sample data set. The pipeline starts with the mapping of NGS reads to a reference genome using Bowtie2, then it continues with the processing of those mapping (BAM) files using SAMtools, identification of variant sites using VarScan3, and ends with the production of a SNP matrix using custom Python scripts (calling of SNPs at each variant site, combining the SNPs into a SNP matrix). The Python scripts are reused from the *CFSAN SNP Pipeline: an automated method for constructing SNP matrices from next-generation sequence data*. *PeerJ Computer Science 1:e20* <https://doi.org/10.7717/peerj-cs.20>. As it can be observed in this data set, there are four samples, whose dataflow process is more detailed in the [documentation page](#) of this pipeline.

```
Pipeline "Github" "https://github.com/Vacalexix/tools" {
  tool "snp-pipeline" "DockerConfig" {
    command "create_sample_dirs" {
      argument "-d" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪samples/*"
      argument "--output" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/sampleDirectories.txt"
    }
  }

  tool "Bowtie2" "DockerConfig" {
    command "bowtie2-build" {
      argument "reference_in" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/reference/lambda_virus.fasta"
      argument "bt2_base" "reference"
    }
    command "bowtie2" {
```

```

        argument "-p" "1"
        argument "-q" "-q"
        argument "-x" "reference"
        argument "-1" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪samples/sample1/sample1_1.fastq"
        argument "-2" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪samples/sample1/sample1_2.fastq"
        argument "-S" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪reads1.sam"
    }
    command "bowtie2" {
        argument "-p" "1"
        argument "-q" "-q"
        argument "-x" "reference"
        argument "-1" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪samples/sample2/sample2_1.fastq"
        argument "-2" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪samples/sample2/sample2_2.fastq"
        argument "-S" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪reads2.sam"
    }
    command "bowtie2" {
        argument "-p" "1"
        argument "-q" "-q"
        argument "-x" "reference"
        argument "-1" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪samples/sample3/sample3_1.fastq"
        argument "-2" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪samples/sample3/sample3_2.fastq"
        argument "-S" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪reads3.sam"
    }
    command "bowtie2" {
        argument "-p" "1"
        argument "-q" "-q"
        argument "-x" "reference"
        argument "-1" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪samples/sample4/sample4_1.fastq"
        argument "-2" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪samples/sample4/sample4_2.fastq"
        argument "-S" "snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪reads4.sam"
    }
}
tool "Listing" "DockerConfig" {
    command "startListing" {
        argument "referenceName" "reads.sam"
        argument "filesList" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/reads1.sam snp-pipeline-master/snppipeline/data/lambdaVirusInputs/
↪reads2.sam snp-pipeline-master/snppipeline/data/lambdaVirusInputs/reads3.sam snp-
↪pipeline-master/snppipeline/data/lambdaVirusInputs/reads4.sam"
    }
}
tool "Samtools" "DockerConfig" {
    command "view" {
        argument "-b" "-b"
    }
}

```

```

        argument "-S" "-S"
        argument "-F" "4"
        argument "-o" "reads.unsorted.bam"
        argument "input" "reads.sam"
    }

    command "sort" {
        argument "-o" "reads.sorted.bam"
        argument "input" "reads.unsorted.bam"
    }

    command "mpileup" {
        argument "--fasta-ref" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/reference/lambda_virus.fasta"
        argument "input" "reads.sorted.bam"
        argument "--output" "reads.pileup"
    }
}

tool "VarScan" "DockerConfig" {
    command "mpileup2snp" {
        argument "mpileupFile" "reads.pileup"
        argument "--min-var-freq" "0.90"
        argument "--output-vcf" "1"
        argument "output" "var.flt.vcf"
    }
}

tool "Listing" "DockerConfig" {
    command "stopListing" {
        argument "referenceName" "var.flt.vcf"
        argument "destinationFiles" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/samples/sample1/var.flt.vcf snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/samples/sample2/var.flt.vcf snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/samples/sample3/var.flt.vcf snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/samples/sample4/var.flt.vcf"
    }
}

tool "snp-pipeline" "DockerConfig" {
    command "create_snp_list" {
        argument "--vcfname" "var.flt.vcf"
        argument "--output" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/snplist.txt"
        argument "sampleDirsFile" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/sampleDirectories.txt"
    }
}

tool "Listing" "DockerConfig" {
    command "restartListing" {
        argument "referenceName" "reads.pileup"
    }
}

tool "snp-pipeline" "DockerConfig" {
    command "call_consensus" {
        argument "--snpListFile" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/snplist.txt"
        argument "--output" "consensus.fasta"
        argument "--vcfFileName" "consensus.vcf"
        argument "allPileupFile" "reads.pileup"
    }
}

tool "Listing" "DockerConfig" {

```

```
command "stopListing" {
    argument "referenceName" "consensus.fasta"
    argument "destinationFiles" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/samples/sample1/consensus.fasta snp-pipeline-master/snppipeline/
↪data/lambdaVirusInputs/samples/sample2/consensus.fasta snp-pipeline-master/
↪snppipeline/data/lambdaVirusInputs/samples/sample3/consensus.fasta snp-pipeline-
↪master/snppipeline/data/lambdaVirusInputs/samples/sample4/consensus.fasta"
}
}
tool "snp-pipeline" "DockerConfig" {
    command "create_snp_matrix" {
        argument "sampleDirsFile" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/sampleDirectories.txt"
        argument "--consFileName" "consensus.fasta"
        argument "--output" "snp-pipeline-master/snppipeline/data/
↪lambdaVirusInputs/snpma.fasta"
    }
}
}
```

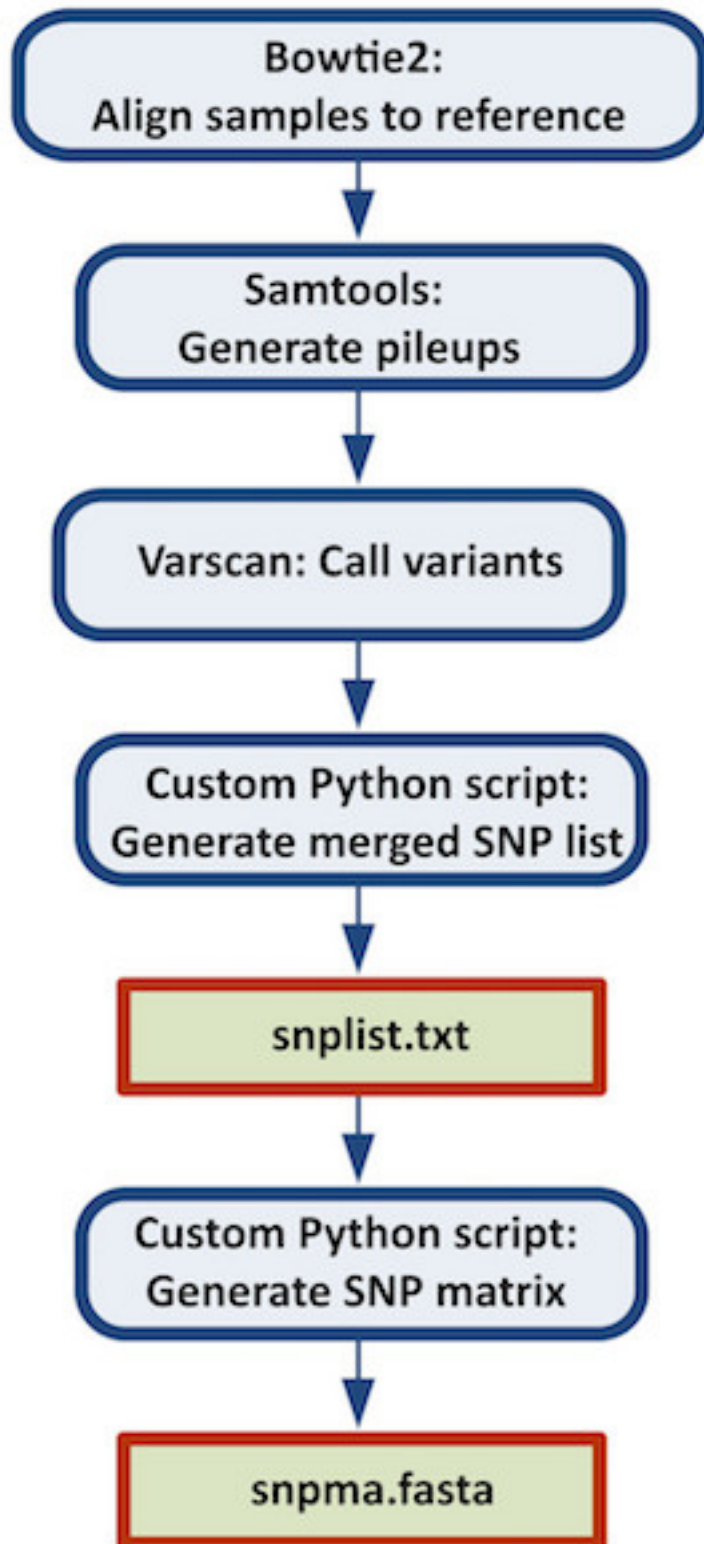


Figure 6.3: Figure from Davis S, Pettengill JB, Luo Y, Payne J, Shpuntoff A, Rand H, Strain E. (2015) CFSAN SNP Pipeline: an automated method for constructing SNP matrices from next-generation sequence data. *PeerJ Computer Science* 1:e20 <https://doi.org/10.7717/peerj-cs.20>

Running this example in Engine for Cloud

It is similar to the previous example. **Note that this tool types are only available for running in the Engine for Cloud.**

Note Please, be sure that the Engine for Cloud is already installed. For this, follow the steps that are in section:

Engine->Engine for Cloud-> Install engine for cloud.

If previously installed, please ensure that:

- the ip of the virtual machine is configured
- the environment variable is established on the terminal that you are executing the monitor. For managing these settings, please also consult the section:

Engine->Engine for Cloud-> Install engine for cloud.

After the installation, you should have the following tree file:

```
WorkingDirectory
|-- Analyser\
    |-- ngs4cloud-analyser-1.0-SNAPSHOT\
        |-- bin
            |--ngs4cloud-analyser
            |--ngs4cloud-analyser.bat (CUI Window run script)
    |-- Monitor\
        |-- monitor.jar
    |-- (other files,...)
```

- **Input data is available [here](#)**, but is not necessary to download. Input data in Engine for Cloud engine is always passed as an URI.
- Create a file `casestudy3.pipes` (the extension containing the pipeline previously described in Figure 6.3. Assume that, on the following, `casestudy3.pipes` is inside the directory `ngs4cloud-analyser-1.0-SNAPSHOT`).
- Start by execution the analyser tool, in order to produce an file with `json` extension.

OSX/Linux

```
ngs@server:ngs4cloud-analyser-1.0-SNAPSHOT$ ./bin/ngs4cloud-analyser analyse
-pipes casestudy3.pipes
-ir ir3.json
-input https://github.com/CFSAN-Biostatistics/snp-pipeline/archive/master.
↪zip
-outputs snp-pipeline-master/snppipeline/data/lambdaVirusInputs/snplist.txt
↪snp-pipeline-master/snppipeline/data/lambdaVirusInputs/snpma.fasta
```

- This execution will produce the file `ir3.json`.
- Then, copy the `ir3.json` inside to directory `Monitor`
- Before executing the Monitor, please assure that the Virtual Machine with the cluster image given for test purposes is lauched and correctly settled (please, see the section

```
Engine->Engine for cloud->
    Install the engine for cloud -> Install the monitor
```

- Launch the pipeline into the cluster through the monitor command

```
ngs@server:Monitor$ java -jar monitor.jar launch ir3.json
```

- The previous command will generate a pipeline id. Assume in this example that the id is 3.
- Consult the status of the pipeline by its id

```
ngs@server:Monitor$ java -jar monitor.jar status 3
```

- After pipeline is finished, it is possible to download its results from the cluster to a previously defined directory inside the Monitor directory.

```
ngs@server:Monitor$ java -jar monitor.jar outputs 3 resultsDirectory3
```

- `resultsDirectory` is the directory that contains a copy of the outputs that were previously specified by the analyser that should be copied; 3 is the pipeline id

For more information about the `analyser` and `monitor` commands and its parameters, please see section

```
Engine->Engine for cloud->Run the engine for cloud
```

//: # (##A pipeline using split and join tools (for executing only with Engine for Cloud))