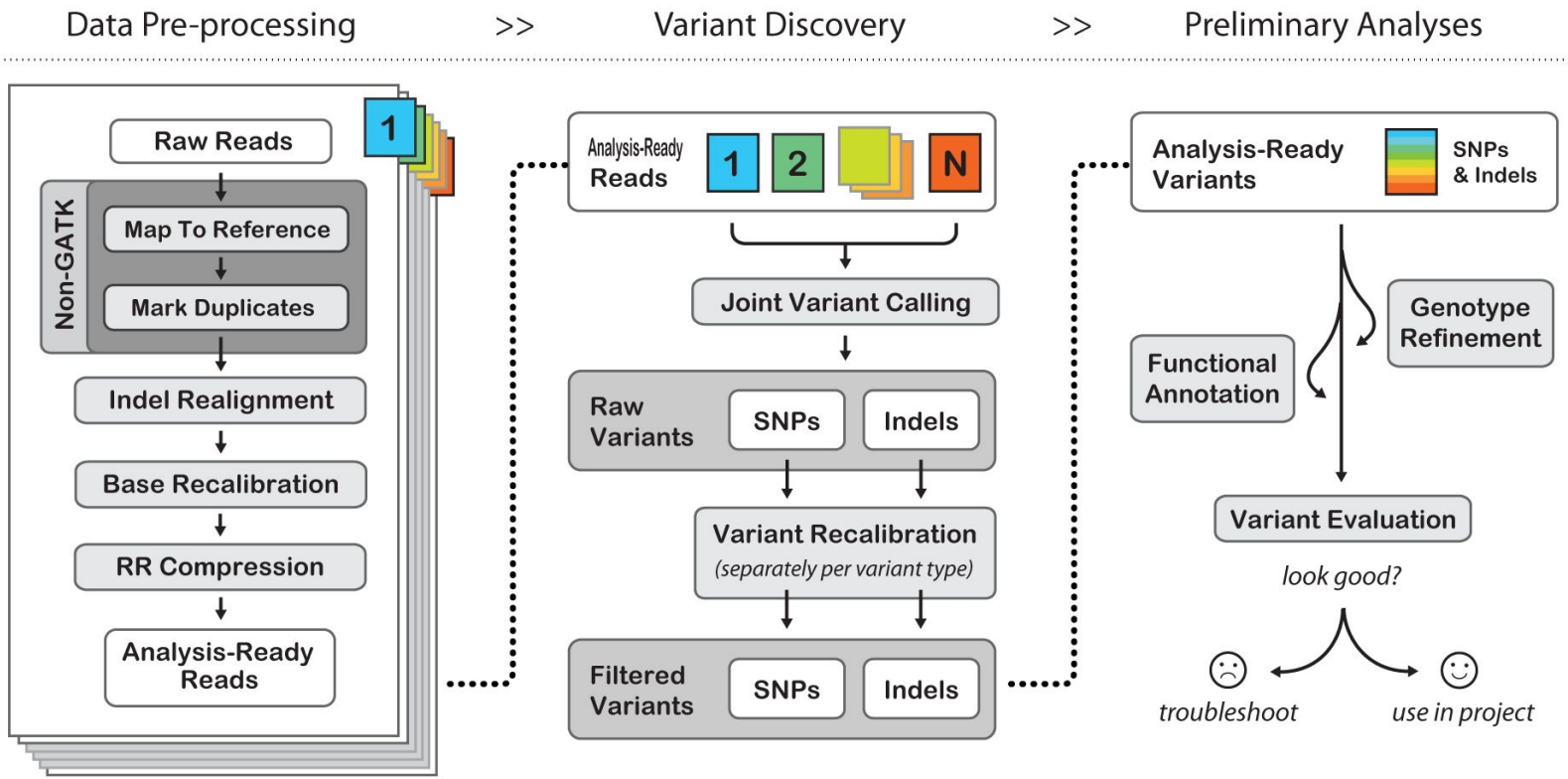
Intro to Workflow management systems

Brice Letcher & Paul Saary

EMBL-EBI

Workflows



GATK best practices:

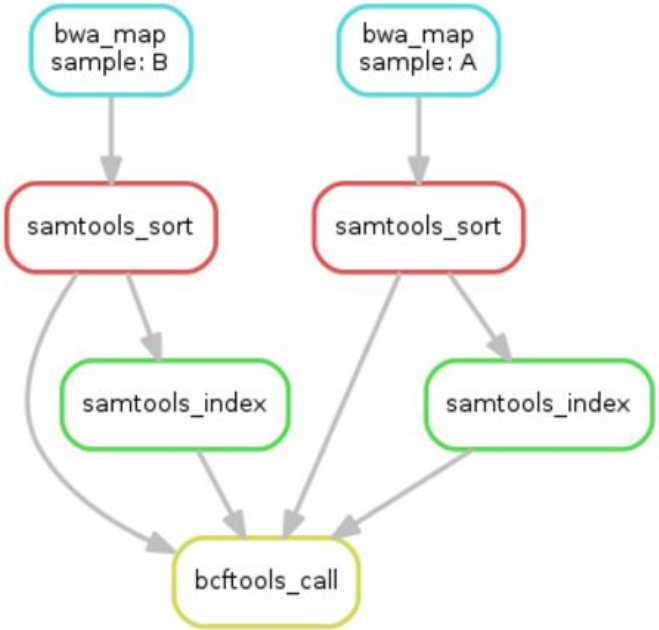
[DNA Variant Discovery]

Why a tool for workflow management?

We need to handle complicated things like:

- **Forking** processes: running independent processes simultaneously
- **Rejoining** processes: combining the output from independent processes once they have completed
- Setting up process **environments** (eg access to tools, libraries), allocated **resources** (threads, RAM), **logging** to files.
- Creating **reports** showing, for *eg*, the time taken by each process.
- **Deploying** your pipeline on different platforms: Mac/Windows/Linux, different clusters, the cloud.
- **Sharing** your pipeline: readability & how easy it is to modify.
- **Restart** your pipeline where it last failed/stopped.

DAG representation



Workflow ⇔ Directed Acyclic Graph (DAG)

Nodes are processes

Edge: Node(A) -> Node(B) means A needs to complete before B can run.

Graph is acyclic: a process cannot depend on itself.

What a workflow manager needs to implement

- Declare processes
- Let data flow between processes
- Specify process dependency structure

Popular in bioinformatics :

Snakemake https://snakemake.readthedocs.io/en/stable/

Nextflow https://www.nextflow.io/

Snakemake

Python package which extends the Python language with syntax specific to workflows

- Processes are called **rules**
- Data flows between processes via files, always
- Process dependency structure achieved by linking output files of one rule with input files of another

Rule: Basic syntax

```
rule sort:
    input:
        "path/to/dataset.txt"
    output:
        "dataset.sorted.txt"
    shell:
        "sort {input} > {output}"
```

Each rule is a promise: If I find this input file, I will make this output file.

The promise should be fulfilled by running the shell code.

Example from:
http://slides.com/johanneskoester

Rules: Wildcards

```
rule sort:
    input:
        "path/to/{dataset}.txt"
    output:
        "{dataset}.sorted.txt"
    shell:
        "sort {input} > {output}"
```

Wildcards allow running a rule on multiple data.

Can be accessed under `wildcard` namespace (eg “{wildcards.dataset}”)

Example from:
http://slides.com/johanneskoester

Multiple inputs/outputs: access by index

```
rule sort:
    input:
        "path/to/{dataset}.txt"
        "path/to/annotation.txt"
    output:
        "{dataset}.sorted.txt"
    shell:
        "paste <(sort {input[0]}) {input[1]} > {output}"
```

Example from:
http://slides.com/johanneskoester

Multiple inputs/outputs: access by name

```
rule sort:
    input:
        a="path/to/{dataset}.txt"
        b="path/to/annotation.txt"
    output:
        b ="{dataset}.sorted.txt"
    shell:
        "paste <(sort {input.a}) {input.b} > {output}"
```

It might be easier to name your inputs and outputs.

This way you can keep better track of what is happening.

Example from:
http://slides.com/johanneskoester

Rules: Run python code

```
rule sort:
     input:
          a="path/to/{dataset}.txt"
     output:
          b ="{dataset}.sorted.txt"
     run:
          with open(output.b, “w”) as out:
               for l in sorted(open(input.a)):
                    print(l, file=out)
```

Instead of running bash code you can also use Python directly inside the rule’s run block.

Example from:
http://slides.com/johanneskoester

Rules: Execute a script

```
rule sort:
     input:
          a="path/to/{dataset}.txt"
     output:
          b ="{dataset}.sorted.txt"
     script:
          "scripts/myScript.R"
```

If you give a rule a script to execute, you can access snakemake-related environment variables (eg wildcards) from inside the script.

Python:

```
outputfile = snakemake.output['b']
```

R:

```
outputfile <- snakemake@output$b
```

Example from:
http://slides.com/johanneskoester

Snakemake is ‘output-oriented’:

It looks for what the workflow end product is, and works backwards from there.

```
rule all:
    input:
        [f“final_outputs/{i}.txt for i in range(4)”]
```

First rule usually specifies the final output and is called `all`

Working workflow

```
DATASETS = ["D1", "D2", "D3"] # Native python array

rule all:
     input:
          expand("{dataset}.sorted.txt", dataset=DATASETS)

rule sort:
     input:
          "path/to/{dataset}.txt"
     output:
          "{dataset}.sorted.txt"
     shell:
          "sort {input} > {output}”
```

Example from:
http://slides.com/johanneskoester

Snakefile execution

```
# execute the workflow with target D1.sorted.txt
snakemake D1.sorted.txt

# execute the workflow without target: first rule defines target
snakemake

# dry-run
snakemake -n

# dry-run, print shell commands
snakemake -n -p

# dry-run, print execution reason for each job
snakemake -n -r
```

Examples from:
http://slides.com/johanneskoester

Mental map

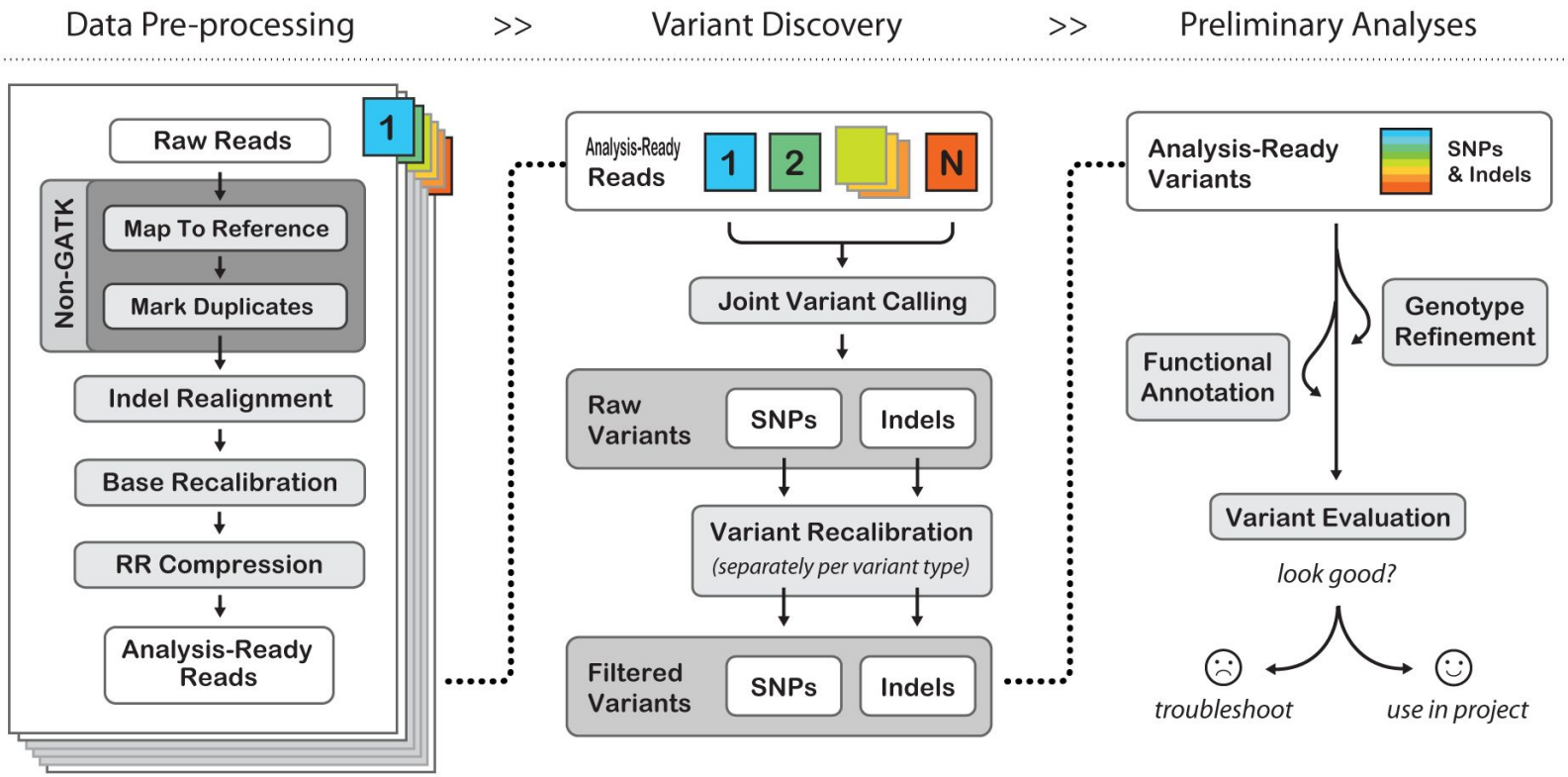
How you write: *a* leads to *b* which leads to *c*

→ Write out the DAG before writing the workflow

How Snakemake reads: *c* needs *b*'s output to run which needs *a*'s output to run

→ Guides writing your rules and debugging

Workflows



GATK best practices:

[Snakemake implementation]

What’s next?

The rest of this workshop is on the webpage:

https://bricoletc.github.io/WMS_teaching