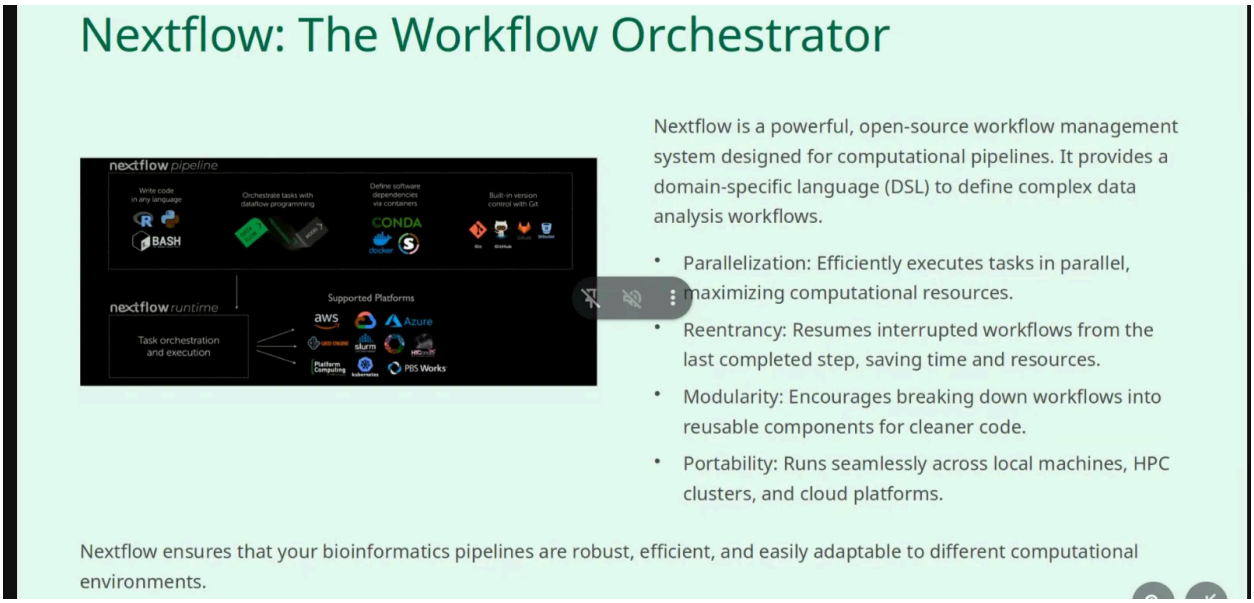
Nextflow: The Workflow Orchestrator



Why DSL matters in Nextflow?

- It defines how you write processes, workflows, channels, and parameters.
- Makes workflows:
  - Easy to reproduce
  - Portable across computing environments (e.g., local, cloud, HPC)
  - Modular and scalable

```
nextflow.enable.dsl=2

process sayHello {
input:
val name

output:
stdout

script:
"""
echo "Hello, $name!"
"""

}
```

```
workflow {
sayHello(name: 'Vibhanshu')
}
```

Seamless Software & Compute Integration

Nextflow and nf-core pipelines are designed for maximum compatibility across diverse computational environments, ensuring your analyses run consistently wherever you need them.

Containerization

Full support for Singularity and Docker, packaging all dependencies for consistent execution.

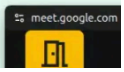
HPC Schedulers

Integrates with common High-Performance Computing (HPC) schedulers like SGE, PBS, LSF, and SLURM.

Cloud-Ready

Execute pipelines on major cloud platforms including Microsoft Azure, Amazon Web Services (AWS), and Google Cloud.

This extensive integration capability ensures your bioinformatics workflows are truly portable and scalable.



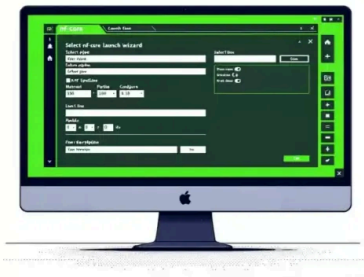
Launching Pipelines with Ease

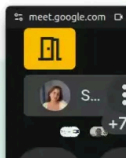
nf-core provides intuitive tools to simplify the process of launching and running complex bioinformatics pipelines.

- Web-Based Launch Wizard: The nf-co.re/launch tool offers a user-friendly interface to configure pipeline parameters without needing to touch the command line.
- Command-Line Efficiency: For advanced users, powerful command-line tools enable quick setup, execution, and customization of pipelines.

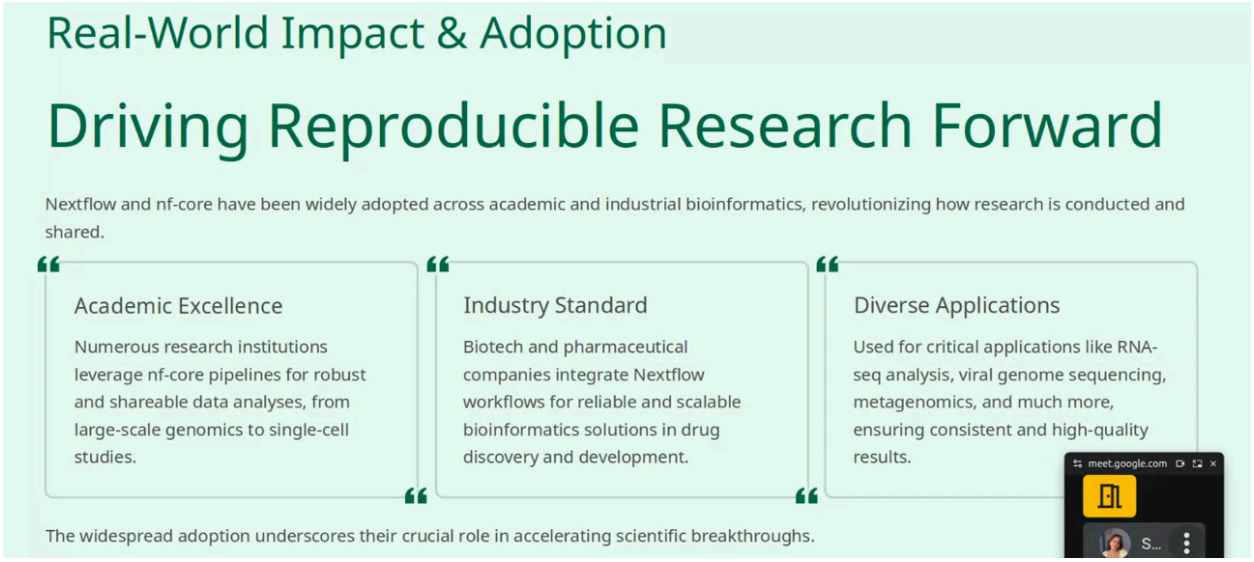
> Example: `nextflow run nf-core/rnaseq -params-file nf-params.`

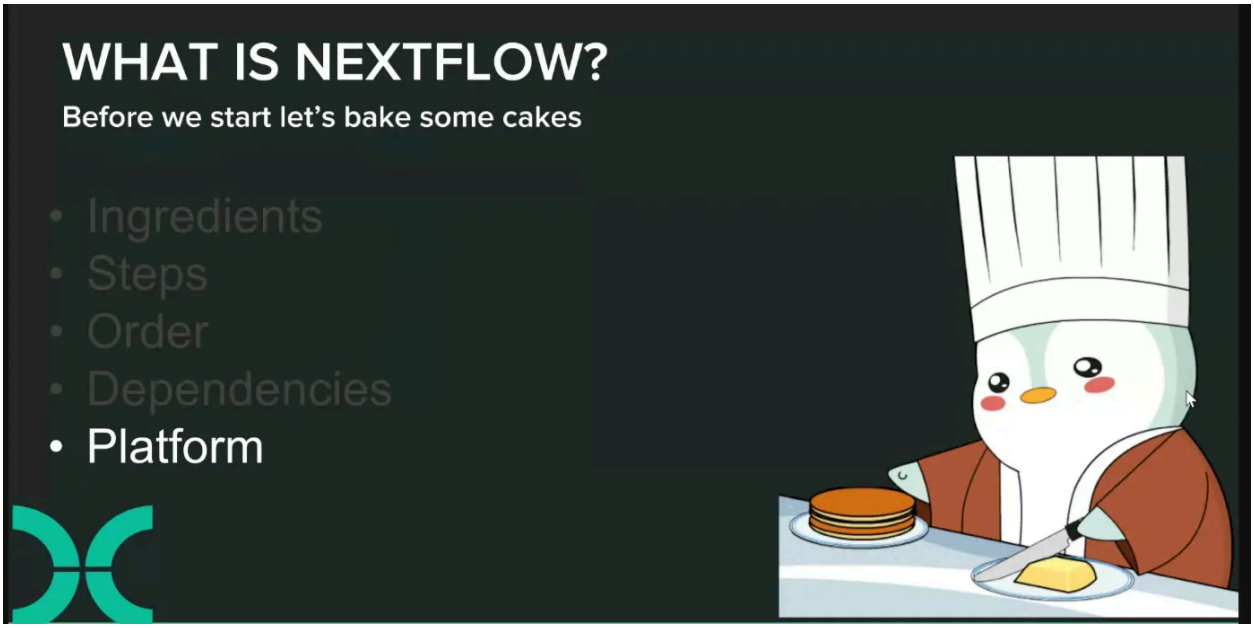
This command initiates the RNA-seq pipeline using parameters defined in a JSON file, showcasing the simplicity of execution.





Getting started with complex bioinformatics analyses has never been more straightforward.





1. **Ingredients** :

   → Analogous to input data, tools, and software components you need to run a workflow.

2. **Steps** :

   → Represents individual computational tasks or processes in the workflow.

3. **Order** :

   → Highlights that steps must execute in a particular sequence.

4. **Dependencies** :

   → Suggests that some steps depend on the results of others.

5. **Platform** :

   → Likely emphasizes Nextflow's ability to run on multiple computing environments (local, HPC clusters, cloud services like AWS, Google Cloud, etc.)

Simple Example of Nextflow Process:

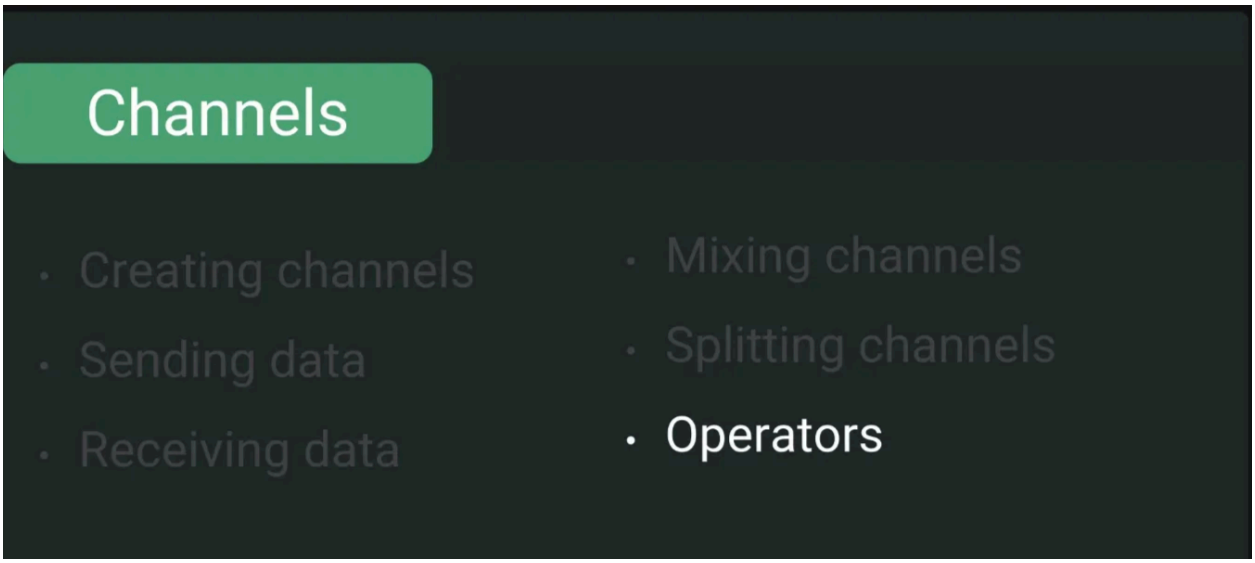
```
process sayHello {
input:
val name

output:
stdout

script:
"""
echo "Hello, $name!"
"""
}
```

The **four key components of a Nextflow process** are usually:

1. **Process definition** → Defines the block of code as a process.
2. **Input** → Data or channels coming into the process.
3. **Output** → Results or channels produced by the process.
4. **Script (or Command block)** → The actual commands or scripts executed by the process.



- **Creating channels** :

  → How to create channels in Nextflow using commands like `Channel.fromPath`, `Channel.from`, etc.

- **Sending data** :

  → How processes send data into channels to be consumed by other processes.

- **Receiving data** :

  → How data flows from channels into processes as inputs.

- **Mixing channels** :

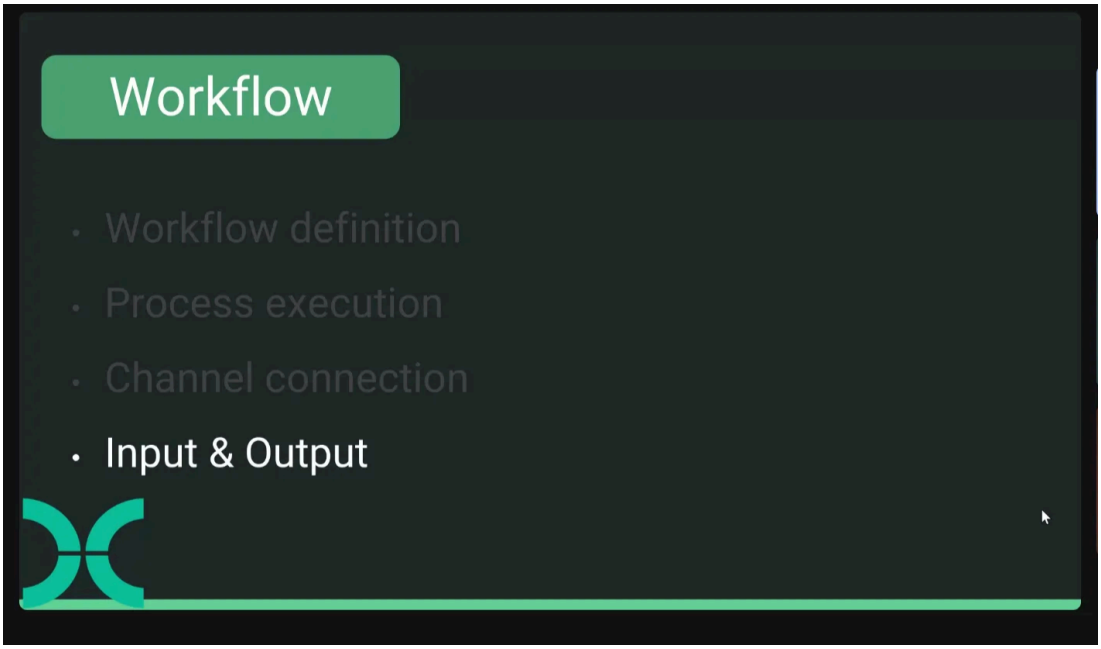
  → Combining multiple channels into one (e.g., `mix`, `combine`).

- **Splitting channels** :

  → Separating data from a channel into multiple outputs using operators.

- **Operators**:

  → Special methods provided by Nextflow to manipulate data in channels, such as `map`, `filter`, `flatMap`, `collect`, etc.



- **Workflow definition** :

  → How to define a workflow in Nextflow using the `workflow { ... }` block.

- **Process execution** :

  → How Nextflow runs the defined processes, managing parallelism and dependencies.

- **Channel connection** :

  → How data flows between processes via channels.

- **Input & Output** :

  → How workflows handle passing input data into processes and collecting outputs, connecting channels and processes to form the overall pipeline.

```
workflow {
Channel.from('Alice', 'Bob', 'Charlie')
.set { names }

    sayHello(name: names)

}
```

EXAMPLE

```
#!/usr/bin/env nextflow

process sayHello {
publishDir 'results', mode: 'copy'

    input:
    val greeting

    output:
    path 'output.txt'

    script:
    """
    echo '$greeting' > output.txt
    """

}

params.greeting = 'Hola mundo!'

workflow {
sayHello(params.greeting)
}
```

🧰 Purpose:

- This script demonstrates a simple **Nextflow pipeline** that prints a greeting message into a file using **channels, process, and workflow structure**.

1️⃣ Shebang Line:

```
#!/usr/bin/env nextflow
```

- Specifies that this script should be run using Nextflow.

2️⃣ Process Definition:

```
process sayHello {
    publishDir 'results', mode: 'copy'

    input:
    val greeting

    output:
    path 'output.txt'

    script:
    """
    echo '$greeting' > output.txt
    """
}
```

- `process sayHello`: Defines a process named `sayHello`.
- **`publishDir 'results', mode: 'copy'`**: Output will be copied to the `results/` directory.
- `input:`: Receives a value called `greeting`.
- `output:`: Produces a file `output.txt`.
- `script:`: Executes the command that echoes the greeting into `output.txt`.

3️⃣ Pipeline Parameters:

```
params.greeting = 'Hola mundo!'
```

- Sets a parameter `greeting` with the value `'Hola mundo!'`.

4️⃣ Workflow Block:

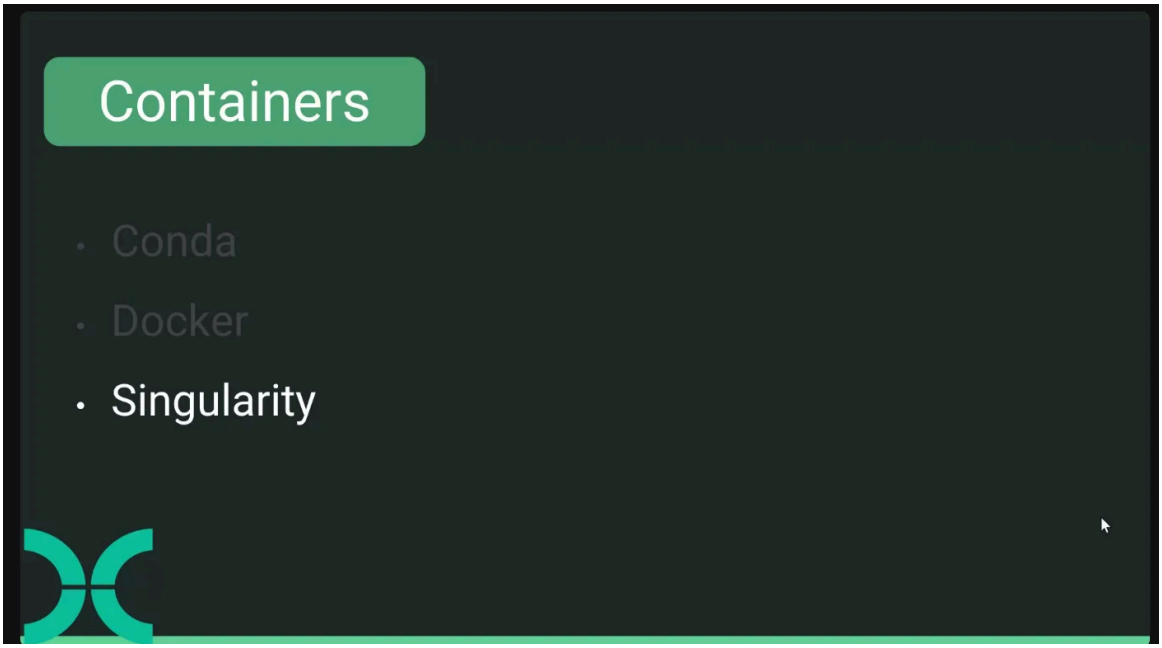
```
workflow {
    sayHello(params.greeting)
```

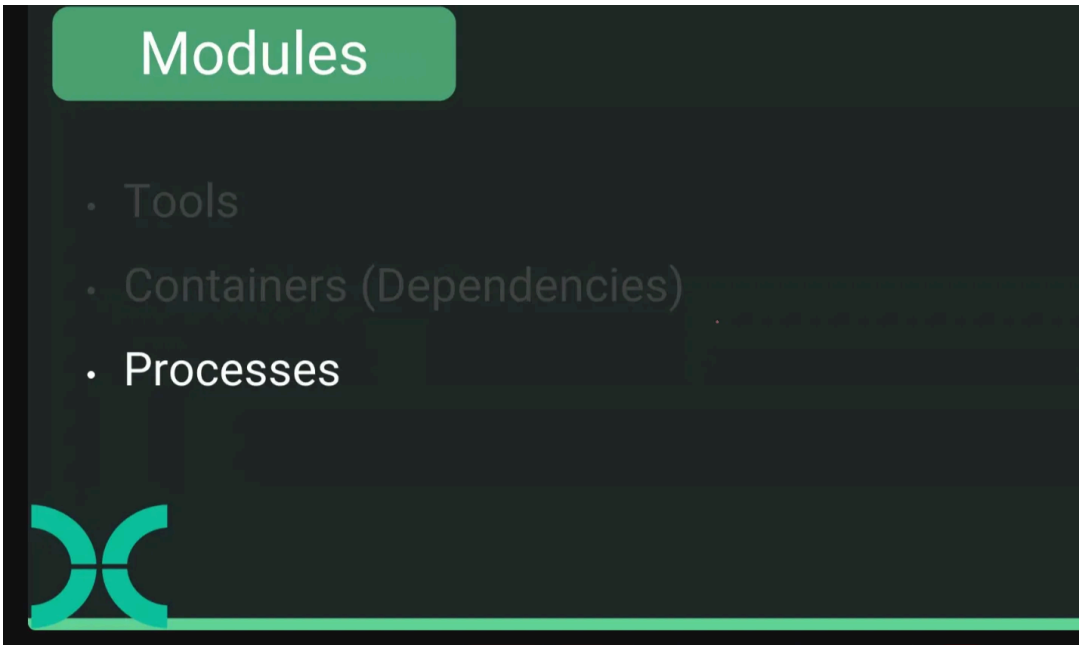
```
}
```

- Calls the process `sayHello`, passing the greeting parameter.

Overall Behavior of the Script:

1. Nextflow initializes the pipeline.
2. The `params.greeting` is passed as input to the `sayHello` process.
3. The process creates a file `output.txt` containing the text `'Hola mundo!'`.
4. The output file is saved in the `results/` directory.





Nextflow process definition written in the `main.nf` file, part of a pipeline module called `idxstats`. The goal of this process is to run `samtools idxstats` on input BAM files and produce index statistics files.

Here is a detailed explanation of the code:

✅ 1. Process Declaration

```
process SAMTOOLS_IDXSTATS {
```

Defines a Nextflow process named `SAMTOOLS_IDXSTATS`.

✅ 2. Metadata and Label

```
tag "$meta.id"
label 'process_single'
```

- `tag "$meta.id"`: Adds a tag for easy logging and monitoring.
- `label 'process_single'`: Used for task scheduling (e.g., resource selection).

✅ 3. Container Setup

```
conda "bioconda::samtools=1.16.1"
container "${ workflow.containerEngine == 'singularity' && !task.ext.singularity_pull_docker_container ? 'https://depot.galaxyproject.org/singularity/samtools:1.16.1--h6899075_1' : 'quay.io/biocontainers/samtools:1.16.1--h6899075_1' }"
```

- Uses **Conda** and/or **Singularity/Docker container** to ensure reproducible environment.
- Automatically selects appropriate container source based on configuration.

✅ 4. Input Specification

```
input:
  tuple val(meta), path(bam), path(bai)
```

- Inputs are:
  - `meta`: Metadata in tuple format.
  - `bam`: Path to the BAM file.
  - `bai`: Path to the index file of the BAM.

✅ 5. Output Specification

```
output:
  tuple val(meta), path("*.idxstats"), emit: idxstats
  path "versions.yml", emit: versions
```

- Outputs:
  - Index stats file (`.idxstats`).
  - Versions file (`versions.yml`) for reproducibility.

✅ 6. Execution Condition

```
when:
  task.ext.when == null || task.ext.when
```

- Executes conditionally based on `task.ext.when` value (used for skipping or forcing execution).

✅ 7. Script Section

```
script:
  def args = task.ext.args ?: ''
  def prefix = task.ext.prefix ?: "${meta.id}"

  """
  samtools idxstats \\
    --threads ${task.cpus - 1} \\
    $bam \\
    > ${prefix}.idxstats

  cat <<-END_VERSIONS > versions.yml
  ${task.process}
  END_VERSIONS
  """
```

- Builds a command to run `samtools idxstats` with CPU threading.
- Saves the result to `${prefix}.idxstats`.
- Saves process metadata to `versions.yml`.

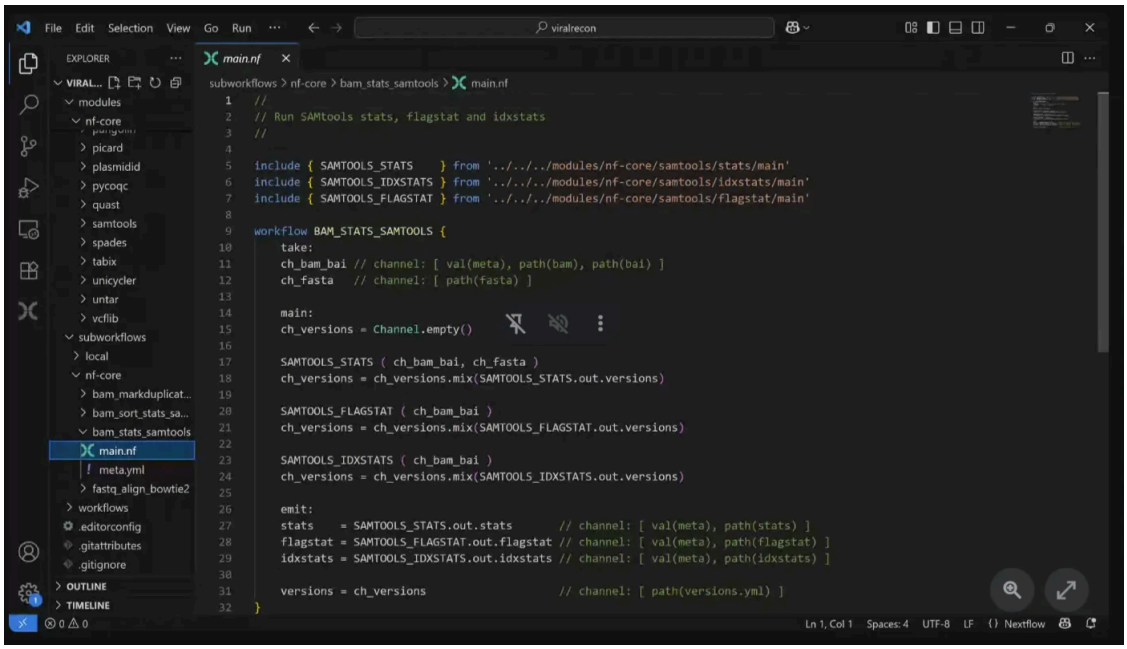
✅ 8. Purpose

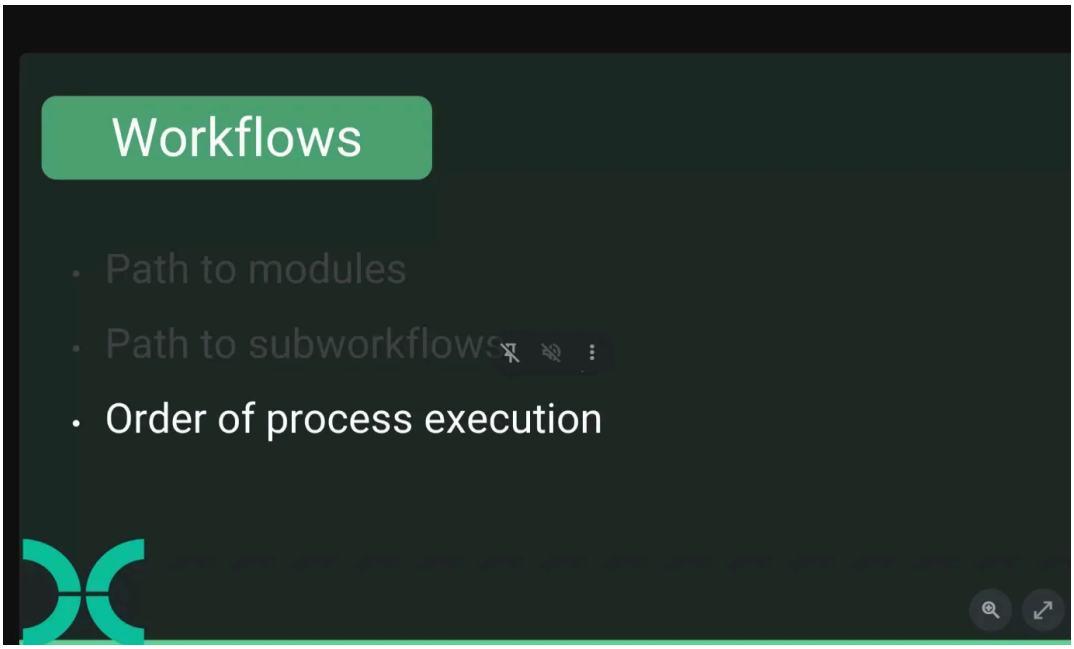
This process is part of a larger Nextflow pipeline (likely viral sequence analysis based on folder structure).

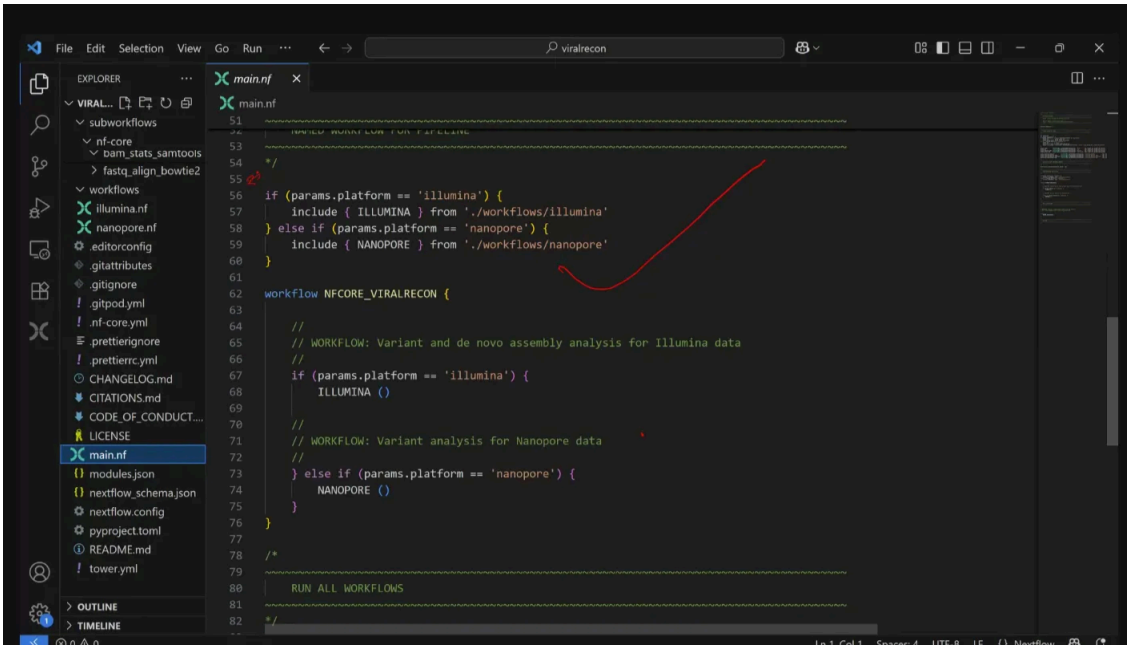
Its role is to calculate alignment statistics for BAM files, which helps assess mapping quality, coverage, etc.

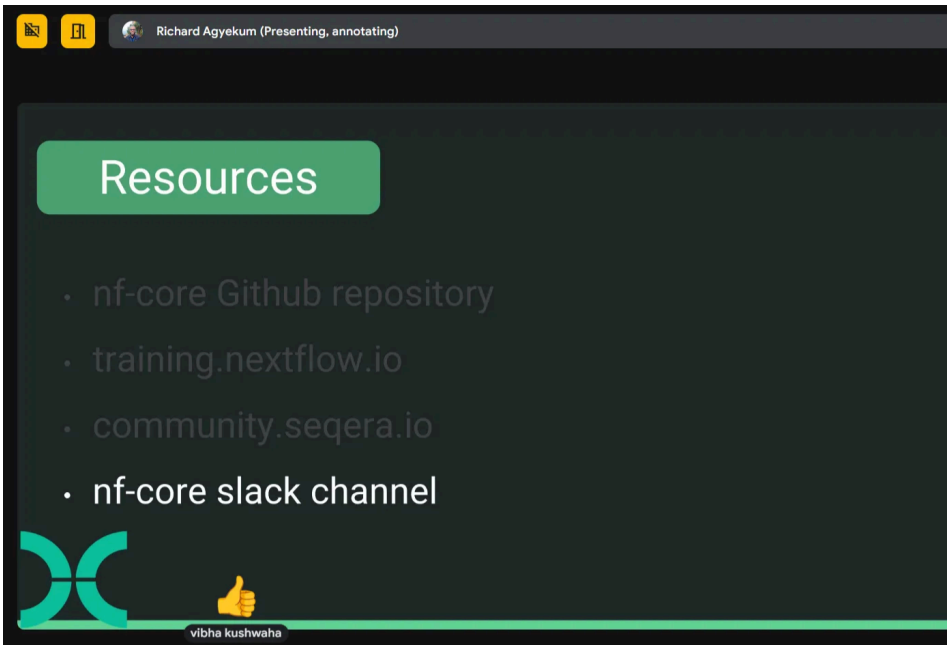
📂 Folder Context (from Explorer Sidebar)

- Located in: `modules/nf-core/samtools/idxstats/`
- Other related files:
  - `meta.yml`
  - `index`, `sort`, `stats`, etc.




Workflows
Path to modules
Path to subworkflows
Order of process execution


File Edit Selection View Go Run
viralrecon
EXPLORER
main.nf
subworkflows
nf-core
bam_stats_samtools
fastq_align_bowtie2
workflows
illumina.nf
nanopore.nf
.editorconfig
.gitattributes
.gitignore
.gitpod.yml
.nf-core.yml
.prettierignore
.prettierrc.yml
CHANGELOG.md
CITATIONS.md
CODE_OF_CONDUCT....
LICENSE
main.nf
modules.json
nextflow_schema.json
nextflow.config
pyproject.toml
README.md
tower.yml
OUTLINE
TIMELINE
*/
if (params.platform == 'illumina') {
include { ILLUMINA } from './workflows/illumina'
} else if (params.platform == 'nanopore') {
include { NANOPORE } from './workflows/nanopore'
}
workflow NFCORE_VIRALRECON {
//
// WORKFLOW: Variant and de novo assembly analysis for Illumina data
//
if (params.platform == 'illumina') {
ILLUMINA ()
//
// WORKFLOW: Variant analysis for Nanopore data
//
} else if (params.platform == 'nanopore') {
NANOPORE ()
}
}
/*
RUN ALL WORKFLOWS
*/


Richard Agyekum (Presenting, annotating)
Resources
nf-core Github repository
training.nextflow.io
community.seqera.io
nf-core slack channel
vibha kushwaha