Developing WDL Workflows

January, 2025

Table of Contents

# About this Course

“Developing WDL Workflows” shows a bioinformatics workflow developer how to strategically develop and scale up a WDL workflow that is iterative, reproducible, and efficient in terms of time and resource used. This guide is flexible regardless of where the data is, what computing resources are being used, and what software is being used.

## 0.1 Target Audience

The course is intended for first time developers of the [WDL workflow language](https://github.com/openwdl/wdl), who wants to iteratively develop a WDL bioinformatics workflow. In order to use this guide the audience should be able to comprehend introductory WDL syntax, and should be able to run a WDL workflow on a computing engine of their choice, such as Cromwell, miniwdl, or a cloud computing environment such as Terra, AnVIL, or Dockstore.

### 0.1.1 Relevant Resources

If you are new to WDL, [OpenWDL Docs](https://docs.openwdl.org/en/stable/) is an excellent WDL companion resource to help you get started and is platform agnostic. OpenWDL Docs focuses on the basic grammar of WDL as well as providing excellent cookbook recipes of common WDL workflow structures. In this guide we will reference these basic grammar structures and common workflow cookbook recipes.

## 0.2 Why WDL?

You may have encountered other workflow tools, such as Snakemake or Nextflow, and those are highly capable. Why learn a brand new workflow language? Let’s review some WDL Pros and Cons:

### 0.2.1 WDL Pros

WDL has some really helpful advantages compared to other frameworks:

* **Portability**. WDL can run on nearly any system, whether it be your local computer, or on an HPC cluster, or on the Cloud, with platforms such as DNAnexus. In fact, a lot of developers will prototype a WDL workflow on their own local computer before moving it to the cloud.
* **Reproducibility**. Ever have the headache of having to reproduce the exact package versions to get your workflow to work again? If you use Docker containers to specify your software environment, you do not have to worry about this headache. A workflow will run identically locally, on HPC, or the cloud.
* **Sharing**. A WDL workflow is much easier to share with colleagues and is a good way to get credit for work you do everyday. If you spent time building it, why not share it? WDL is also an open standard and supported by a number of software tools.
* **Running and Making WDL workflows is a transferable skill.** Genomics and Pharma companies rely on WDL workflows to process thousands of FASTA/VCF files for their studies. They need more experts. It makes you more hireable within both Academia and Industry.

### 0.2.2 WDL Cons

Of course, nothing is free. WDL does require you to understand the basic concepts and terminologies including:

* Basics of Docker
* Understanding the WDL framework
* Converting your bash scripts into WDL tasks and workflows

## 0.3 Curriculum

The course covers the following:

* How to write an effective WDL task
* Link multiple WDL tasks together in a workflow
* Organize variables via structs, scale multiple samples via Arrays
* Reuse repeated tasks via task aliasing
* Configure settings for the execution engine

# 1 Introduction to WDL

## 1.1 Introduction

Welcome to building your first WDL workflow! This guide will help you strategically develop and scale up a WDL workflow that is iterative, reproducible, and efficient in terms of time and resource used.

To make sure that we are on the same page, this guide assumes that you are able to run a WDL on a computing engine of your choice, such as Cromwell, miniwdl, or a cloud computing environment such as Terra, AnVIL, or Dockstore. This guide also assumes that you have a beginner’s understanding of the WDL syntax, and we will link out to additional resources to fill in the knowledge gap as needed! If you have never seen the WDL language in action, a great place to start is [OpenWDL docs](https://docs.openwdl.org/en/stable/) – it teaches you the basic syntax and showcases WDL features via concrete examples.

## 1.2 Review of basic WDL syntax

We will do some review of the WDL syntax. A WDL workflow consists of at least one task.

version 1.0  
  
task do\_something {  
 command <<<  
 exit 0  
 >>>  
}  
  
workflow my\_workflow {  
 call do\_something  
}

A workflow, and the tasks it calls, generally has inputs.

version 1.0  
  
task do\_something {  
 input {  
 File fastq  
 }  
 command <<<  
 exit 0  
 >>>  
}  
  
workflow my\_workflow {  
 input {  
 File fq  
 }  
 call do\_something {  
 input:  
 fastq = fq  
 }  
}

The input fq is defined to be a File variable type. WDL supports various variable types, such as String, Integer, Float, and Boolean. For more information on types in WDL, we recommend [OpenWDL’s documentation on variable types](https://docs.openwdl.org/en/stable/WDL/variable_types/).

To access a task-level input variable in a task’s command section, it is usually referenced using ~{*input\_name*} notation, such as (~{fastq}) in the example below. To access a workflow-level variable in a workflow, it is referenced just by its name without any special notation, such as fq in the example below. To access a workflow-level variable in a task, it must be passed into the task as an input.

version 1.0  
  
task do\_something {  
 input {  
 File fastq  
 String basename\_of\_fq  
 }  
 command <<<  
 echo "First ten lines of ~{basename\_of\_fq}: "  
 head ~{fastq} ## example of referring to task-level input  
 >>>  
}  
  
workflow my\_workflow {  
 input {  
 File fq  
 }  
   
 String basename\_of\_fq = basename(fq)  
   
 call do\_something {  
 input:  
 fastq = fq, ## example of referring to workflow-level input  
 basename\_of\_fq = basename\_of\_fq  
 }  
}

Tasks and workflows also typically have outputs. The task-level outputs can be accessed by the workflow or any subsequent tasks. The workflow-level outputs represent the final output of the overall workflow.

version 1.0  
  
task do\_something {  
 input {  
 File fastq  
 String basename\_of\_fq  
 }  
 command <<<  
 echo "First ten lines of ~{basename\_of\_fq}: " >> output.txt  
 head ~{fastq} >> output.txt  
 >>>  
 output {  
 File first\_ten\_lines = "output.txt" ## output variable for task  
 }  
}  
  
workflow my\_workflow {  
 input {  
 File fq  
 }  
   
 String basename\_of\_fq = basename(fq)  
   
 call do\_something {  
 input:  
 fastq = fq,  
 basename\_of\_fq = basename\_of\_fq  
 }  
   
 output {  
 File ten\_lines = do\_something.first\_ten\_lines ## referring to task output here  
 }  
}

## 1.3 Using JSONs to control workflow inputs

Running a WDL workflow generally requires two files: A .wdl file, which contains the actual workflow, and a .json file, which provides the inputs for the workflow.

In the example we showed earlier, the workflow takes in a file referred to by the input variable fq. This needs to be provided by the user. Typically, this is done with a JSON file. Here’s what a JSON file for this workflow might look like:

{  
 "my\_workflow.fq": "./data/example.fq"  
}

JSON files consist of key-value pairs. In this case, the key is "my\_workflow.fq" and the value is the path "./data/example.fq". The first part of the key is the name of the workflow as written in the WDL file, in this case my\_workflow. The variable being represented is referred to its name, in this case, fq. So, in the above example, the file located at the path ./data/example.fq is being input as a variable called fq into the workflow named my\_workflow.

Files aren’t the only type of variable you can refer to when using JSONs. Here’s an example JSON for every common WDL variable type.

{  
 "some\_workflow.file": "./data/example.fq",  
 "some\_workflow.string": "Hello world!",  
 "some\_workflow.integer": 1965,  
 "some\_workflow.float": 3.1415,  
 "some\_workflow.boolean": true,  
 "some\_workflow.array\_of\_files": ["./data/example01.fq", "./data/example02.fq"]  
}

Resources:

For more basic examples of WDL workflow with a single task, we recommend [OpenWDL’s documentation on WDL’s base structure](https://docs.openwdl.org/en/stable/WDL/Base_structure/).

For more information on types in WDL, we recommend [OpenWDL’s documentation on variable types](https://docs.openwdl.org/en/stable/WDL/variable_types/).

If you are having difficulty writing valid JSON files, considering using <https://jsonlint.com/> to check your JSON for any errors.

## 1.4 Running WDL via a computing engine

In order to run a WDL workflow, we need a *computing engine* to execute it. The two most popular WDL executors are [miniwdl](https://miniwdl.readthedocs.io/en/latest/index.html) and [Cromwell](https://cromwell.readthedocs.io/en/stable/). Both computing engines can run WDLs on multiple configurations:

* A local machine
* A High Performance Computing (HPC) Cluster
* A Cloud Computing backend (such as AWS/Terra/DNAnexus).

If you are trying to run WDL at Fred Hutch Cancer Center’s HPC system, you should use the PROOF executor.

If you are computing on a HPC or the Cloud, you should find the best practice of running a WDL computing engine based on your institution’s information technology system.

If you are computing locally on your computer, below is a short guide on how to set that up.

How to run simple workflows locally

Not every WDL workflow will run well on a laptop, but it can be helpful to have a basic setup for testing and catching simple syntax errors. Let’s quickly set up a WDL executor to run our WDLs.

In this course, we will be using miniwdl, but everything in this course will also be compatible with Cromwell unless explicitly stated otherwise. Additionally, almost all WDLs use Docker images, so you will also need to install Docker or a Docker-like alternative.

**Installing Docker and miniwdl is not required to use this course.** We don’t want anybody to get stuck here! If you already have a method for submitting workflows, such as Terra, feel free to use that for this course instead of running workflows directly on your local machine. If you don’t have any way of running workflows at the moment, that’s also okay – we have provided plenty of examples for following along.

### 1.4.1 Installing Docker

**Note: Although Docker’s own docs recommend installing Docker Desktop for Linux,** [**it has been reported**](https://github.com/dockstore/dockstore/issues/5135) **that some WDL executors work better on Linux when installing only Docker Engine (aka Docker CE).** To install Docker on your machine, follow the instructions specific to your operating system [on Docker’s website](https://docs.docker.com/get-docker/). To specifically install only Docker Engine, [use these instructions instead](https://docs.docker.com/engine/install/).

If you are unable to install Docker on your machine, Dockstore (not affiliated with Docker) [provides some experimental alternatives](https://docs.dockstore.org/en/stable/advanced-topics/docker-alternatives.html). Dockstore also provides [a comprehensive introduction to Docker itself](https://docs.dockstore.org/en/stable/getting-started/getting-started-with-docker.html?highlight=engine#where-can-i-run-docker), including how to write a Dockerfile. Much of that information is outside the scope of this WDL-focused course, but it may be helpful for those looking to eventually create their own Docker images.

### 1.4.2 Installing miniwdl

miniwdl is based on Python. If you do not already have Python 3.6 or higher installed, [you can install Python from here](https://www.python.org/downloads/).

Once Python is installed on your system, you can run pip3 install miniwdl from the command line to install miniwdl. For those who prefer to use conda, use conda install -c conda-forge miniwdl instead. Once miniwdl is installed, you can verify it works properly by running miniwdl run\_self\_test. This will run a built-in hello world workflow.

For more information, see [miniwdl’s GitHub repository](https://github.com/chanzuckerberg/miniwdl).

### 1.4.3 Launching a workflow locally with miniwdl

The generic method for running a WDL with miniwdl is the following:

miniwdl run [path\_to\_wdl\_file] -i [path\_to\_inputs\_json]

If you have successfully installed miniwdl, create the following WDL file and name it greetings.wdl:

version 1.0  
  
task greet {  
 input {  
 String user  
 }  
 command <<<  
 echo "Hello ~{user}!" > greets.txt  
 >>>  
 output {  
 String greeting = read\_string("greets.txt")  
 }  
}  
  
workflow my\_workflow {  
 input {  
 String username  
 }  
 call greet {  
 input:  
 user = username  
 }  
}

Next, use this JSON file (or create one of your own) to provide the string that the workflow expects, and call the JSON file greetings.json:

{  
 "my\_workflow.username": "Ash"  
}

On the command line, run the following:

miniwdl run greetings.wdl -i greetings.json

Once the task completes, you should see something like this in your command line:

[timestamp] wdl.w:my\_workflow finish :: job: "call-greet"  
[timestamp] wdl.w:my\_workflow done  
{  
 "dir": "[working directory]/[timestamp]\_my\_workflow",  
 "outputs": {  
 "my\_workflow.greet.greeting": "Hello Ash!"  
 }  
}

Where [timestamp] is the date and time that you are running the workflow, and [working directory] is the working directory that you are running the workflow from. For example:

2023-12-27 13:54:12.209 wdl.w:my\_workflow finish :: job: "call-greet"  
2023-12-27 13:54:12.210 wdl.w:my\_workflow done  
{  
 "dir": "/Users/ash/github/WDL\_Workflows\_Guide/resources/20231227\_135400\_my\_workflow",  
 "outputs": {  
 "my\_workflow.greet.greeting": "Hello Ash!"  
 }  
}

### 1.4.4 Troubleshooting

#### 1.4.4.1 DockerException

If you are seeing a verbose error message that begins with text like this:

2023-12-27 13:43:37.525 wdl.w:my\_workflow.t:call-greet task greet (greetings.wdl Ln 3 Col 1) failed :: dir: "/Users/sammy/github/WDL\_Workflows\_Guide/resources/20231227\_134337\_my\_workflow/call-greet", error: "DockerException", message: "Error while fetching server API version: ('Connection aborted.', FileNotFoundError(2, 'No such file or directory'))", traceback: ["Traceback (most recent call last):", " File \"/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/urllib3/connectionpool.py\", line 790, in urlopen", " response = self.\_make\_request(", " ^^^^^^^^^^^^^^^^^^^", " File \"/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/urllib3/connectionpool.py\",

This is likely caused by miniwdl being unable to connect to Docker Daemon, the underlying technology that runs Docker images. This is necessary with miniwdl even though our example WDL does not specify a Docker image. Make sure you have Docker installed correctly, and make sure Docker is actively running on your machine. If you installed Docker Desktop, simply opening the Docker Desktop app should start Docker Engine. If you installed Docker without Docker Desktop, running dockerd in your command-line should start it. Be aware that starting the Docker Daemon may take a few minutes.

#### 1.4.4.2 Missing required inputs

If you forget to add -i greetings.json to your call, you will see something like this:

my\_workflow (greetings.wdl)  
---------------------------  
  
required inputs:  
 String username  
  
outputs:  
 String greet.greeting  
  
missing required inputs for my\_workflow: username

You may also see this error if you remember to include a JSON file, but it is missing a required input.

#### 1.4.4.3 Check JSON input

If you see an error message like this:

check JSON input; unknown input/output: greetings.username

Double-check your input JSON. The first part of your JSON’s keys refer to the name of the workflow in the WDL file, not the filename of the WDL itself. Even though our WDL is saved as greetings.wdl, within that file, the workflow is named my\_workflow. This means that the input JSON must say "my\_workflow.username", not "greetings.username".

Other common issues with JSON files are mistyping input variables (such as "my\_workflow.ussername") or forgetting to enclose strings in quotation marks. When in doubt, try using <https://jsonlint.com/> to check your input JSON, and double-check the name of your input variables.

Loading…

# 2 Defining a workflow plan

Our WDL guide will center around building a workflow from scratch. As we build out this workflow step-by-step, you will see what strategies and resources are used to develop a workflow that is iterative, reproducible, and efficient in terms of time and resource used. The goal is to use this workflow to illustrate common lessons in writing WDL workflows.

## 2.1 Somatic mutation calling workflow

The workflow used as the example here is tailored to detect somatic mutations in two tumor samples. Initially, the workflow takes as input FASTQ-formatted sequencing data from two tumor specimens and one normal sample (a single normal sample is used here, but typically each tumor might have its own associated normal). Subsequently, it aligns the FASTQ files of each sample with the human reference genome (hg19), proceeds to identify and mark PCR duplicates, and conducts base quality recalibration. Following these steps, the workflow engages in somatic mutation calling, operating in a paired mode, to pinpoint mutations unique to the tumor samples in comparison to the normal one. Concluding the process, the workflow undertakes the annotation of the identified mutations, enriching the dataset with additional insights into the nature of the mutations detected.

The workflow diagram:

[](assets/WDL_101_Course.png)

The tasks involved:

| Task | Function | Inputs | Outputs |
| --- | --- | --- | --- |
| BwaMem | aligns the samples to the reference genome (hg19) | FASTA (.fasta) file | BAM (.bam) file |
| MarkDuplicates | marks PCR duplicates | BAM (.bam) file | BAM (.bam) file |
| ApplyBaseRecalibrator | performs base quality recalibration | BAM (.bam) file | BAM (.bam) file |
| Mutect2 | performs paired somatic mutation calling | BAM (.bam) file | VCF (.vcf) file |
| annovar | annotates the called somatic mutations | VCF (.vcf) file | VCF (.vcf) file |

## 2.2 Workflow testing strategy

As we build out our workflow, how can we verify that it is running correctly besides getting a message such as “Workflow finished with status ‘Succeeded’” or an exit code 0? In [software development](https://www.atlassian.com/continuous-delivery/software-testing), it is essential to test your code to see whether it generates the expected output given a specified input. This principle applies into bioinformatics workflow development also:

1. *Unit Testing*: We need to incorporate tests to ensure that each task we develop is correct.
2. *End-to-end testing*: When we connect all the tasks together to form a workflow, we test that the workflow running end-to-end is correct.

Here are some guidelines for any form of testing:

* The data you use for testing needs to be representative of “real” data.
* You should have an *expectation* of what the resulting output is *before* you run your workflow on it. It can be as specific as a MD5 checksum, or vague such as a certain file format.
* The process is quick to run, ideally in the range of just a few minutes. This often means using a small subset of actual data.
* The data you use for testing is ideally open access so others can verify your workflow also.

## 2.3 Test samples

To serve as an example we use here whole exome sequencing data from three cell lines from the [Cancer Cell Line Encyclopedia](https://pubmed.ncbi.nlm.nih.gov/31068700/).

### 2.3.1 Tumor 1 : HCC4006

HCC4006 is a lung cancer cell line that has a mutation in the gene *EGFR* (Epithelial Growth Factor Receptor), a proto-oncogene. Mutations in *EGFR* result in the abnormal constitutive activation of the EGFR signaling pathway and drive cancer. In this cell-line specifically, the *EGFR* mutation is an in-frame deletion in Exon 19. This mutation results in the constitutive activation of the EGFR protein and is therefore oncogenic.

### 2.3.2 Tumor 2 : CALU1

CALU1 is a lung cancer cell line that has a mutation in the gene *KRAS* (Kirsten rat sarcoma viral oncogene homolog) . *KRAS* is also a proto-oncogene and the most common cancer-causing mutations lock the protein in an active conformation. Constitutive activation of *KRAS* results in carcinogenesis. In this cell-line *KRAS* has a point/missense mutation resulting in the substitution of the amino acid glycine (G) with cysteine (C) at position 12 of the KRAS protein (commonly known as the KRAS G12C mutation). This mutation results in the constitutive activation of KRAS and drives carcinogenesis.

### 2.3.3 Normal : MOLM13

MOLM 13 is a human leukemia cell line commonly used in research. While it is also a cancer cell line for the purposes of this workflow example we are going to consider it as a “normal”. This cell line does not have mutations in *EGFR* nor in *KRAS* and therefore is a practical surrogate in lieu of a conventional normal sample

### 2.3.4 Test data details

Fastq files for all these three samples were derived from their respective whole exome sequencing. However, for the purpose of this guide we have limited the sequencing reads to span +/- 200 bp around the mutation sites for both genes. In doing so we are able to shrink the data files for quick testing.

### 2.3.5 Access to files

All files needed to run the workflow can be [downloaded here](https://figshare.com/articles/dataset/WDL_101_Dataset/25447528).

Loading…

# 3 The first task

Before we write any sort of WDL – whether it is for somatic mutation calling like we will be going over, or any other bioinformatics task – we need to understand the building blocks of WDL: Tasks!

As mentioned in the first part of this course, every WDL workflow is made up of at least one task. A task typically has inputs, outputs, runtime attributes, and a command section. You can think of a task as a discrete step in a workflow. It can involve a single call to a single bioinformatics tool, a sequence of bash commands, an inline Python script… almost anything you can do non-interactively in a terminal, you can do in a WDL task. In this section, we will go over the parts of a WDL task in more detail to help us write a task for somatic mutation calling.

## 3.1 Inputs

The inputs of a task are the files and/or variables you will passing into your task’s command section. Typically, you will want to include at least one File input in a task, but that isn’t a requirement. You can pass most WDL variable types into a task. In our example workflow, we are starting with a single fastq file per sample, and we know we will need to convert it into a sam file. A sam file is an alignment, so we will need a reference genome to align our fastqs to. We also want to be able to control the threading for this task. Our first task’s inputs will therefore start out looking like this:

task some\_aligner {  
 input {  
 File input\_fastq  
 File ref\_fasta  
 }  
[...]  
}

For some aligners, this would be a sufficient set of inputs, but we have decided to use bwa mem in particular to take us from .fastq to .sam. bwa mem requires a lot of index files, which we will also need as inputs. These index files can be specified via an array, but for now we’ll list everything separately to make sure nothing is being left out.

task BwaMem {  
 input {  
 # main input  
 File input\_fastq  
  
 # reference files  
 File ref\_fasta  
 File ref\_fasta\_index  
 File ref\_dict  
 File ref\_amb  
 File ref\_ann  
 File ref\_bwt  
 File ref\_pac  
 File ref\_sa  
 }  
 [...]  
}

### 3.1.1 Referencing inputs in the command section

The command section of a WDL task is a bash script that will be run [non-interactively](https://tldp.org/LDP/abs/html/intandnonint.html) by the WDL executor. Although it is helpful to think of tasks as discrete steps in a workflow, that does not mean each task needs to be a single line of code or use only one piece of software. You could, for example, call a bioinformatics tool and then reprocess the outputs in the same WDL task.

A WDL task’s input variables are generally referred to in the command section using a tilde (~) and curly braces, using heredoc syntax.

Why use heredox syntax?

You may see WDLs that use this notation for the command section in a task:

task do\_something\_curly\_braces {  
 input {  
 String some\_string  
 }  
 command { ## note the brackets  
 some\_other\_string="FOO"  
 echo ${some\_string}  
 echo $some\_other\_string  
 }  
}

We recommend using heredoc-style syntax instead:

task do\_something\_carrots {  
 input {  
 String some\_string  
 }  
 command <<< ## note the '<<<'  
 some\_other\_string="FOO"  
 echo ~{some\_string}  
 echo $some\_other\_string  
 >>> ## closing '>>>'  
}

Heredoc-style syntax for command sections can be clearer than the alternative, as it makes a clearer distinction between bash variables and WDL variables. This is especially helpful for complicated bash scripts. Heredoc-style syntax is also what the WDL 1.1 spec recommends using in most cases. However, the older non-heredoc style is still perfectly functional for a lot of use cases.

To prevent issues with spaces in String and File types, it is often a good idea to put quotation marks around a String or File variables, like so:

task cowsay {  
 input {  
 String some\_string  
 }  
 command <<<  
 cowsay -t "~{some\_string}"  
 >>>  
}

What can happen if we don’t use quotation marks around String or File variables?

If some\_string is “hello world” then the command section of this task is interpreted as the following:

cowsay -t "hello world"

What happens if we had not wrapped ~{some\_string} in quotation marks? If some\_string was just “hello”, it wouldn’t matter. But because some\_string is two words with a space in between, then the script would be interpreted as cowsay -t hello world and cause an error, because the cowsay program thinks world is another argument. By including quotation marks, cowsay -t "~{some\_string}" can be interpreted as cowsay -t "hello world" and you will correctly get a cow’s greeting instead of an error.

Let’s see how we can reference our inputs in the command section of our task.

task BwaMem {  
 input {  
 File input\_fastq  
 File ref\_fasta  
  
 # these variables may look as though they are unused... but bwa mem needs them!  
 File ref\_fasta\_index  
 File ref\_dict  
 File ref\_amb  
 File ref\_ann  
 File ref\_bwt  
 File ref\_pac  
 File ref\_sa  
 }  
 command <<<  
 # warning: this will not run on all backends! see below for an explanation!  
 bwa mem \  
 -p -v 3 -t 16 -M -R '@RG\tID:foo\tSM:foo2' \  
 "~{ref\_fasta}" "~{input\_fastq}" > my\_nifty\_output.sam   
 >>>  
}

If we were to run this task in a workflow as-is, we might expect it to run on any backend that can handle the hardware requirements. Those hardware requirements are a bit steep – the -t 16 part specifically requests 16 threads, for example – but besides that, it may look like a perfectly functional task.

We have written this task to use 16 threads, as noted by -t 16 in the bwa mem call. If you are running on a backend that cannot provide 16 threads, you may need to adjust set the number of threads to a lower number. Later on in this course, we will discuss how to account for lots of different backends using common workarounds and optional variables.

Unfortunately, even on backends that can provide the necessary computing power, it is quite likely this task will not run as expected. This is because of how inputs work in WDL – or, more specifically, how input files get localized when working with WDL.

### 3.1.2 File localization

When running a WDL, a WDL executor will typically place duplicates of the input files in a brand-new subfolder of the task’s [working directory](https://www.ibm.com/docs/en/zos/3.1.0?topic=directories-working-directory). Typically, you don’t know the name of the directory before runtime – they vary depending on the backend you are running and the WDL executor itself. Thankfully, at runtime, File-type variables such as ~{input\_fastq} and ~{ref\_fasta} will be replaced with paths to their respective files.

For example, if you were to run this workflow on a laptop using miniwdl, ~{ref\_fasta} would likely end up turning into ./\_miniwdl\_inputs/0/ref.fa at runtime. On the other hand, if you were running the exact same workflow with Cromwell, ~{ref\_fasta} would turn into something like /cromwell-executions/BwaMem/97c9341e-9322-9a2f-4f54-4114747b8fff/call-test\_localization/inputs/-2022115965/ref.fa. Keep in mind that these are the paths of *copies* of the input files, and that sometimes input files can be in different subfolders. For example, it’s possible ~{input\_fastq} would be ./\_miniwdl\_inputs/0/sample.fastq while ~{ref\_fasta} may be ./\_miniwdl\_inputs/1/ref.fa.

For many programs, an input file being at ./ref.fa versus /\_miniwdl\_inputs/0/ref.fa is inconsequential. However, this aspect of WDL can occasionally cause issues. bwa mem is a great example of the type of command where this sort of thing can go haywire without proper planning, due to the program making an assumption about some of your input files. Specifically, bwa mem assumes that the reference fasta that you pass in shares the same folder as the other reference files (ref\_amb, ref\_ann, ref\_bwt, etc), and it does not allow you to specify otherwise.

Another example of file localization issue

bwa is not the only program that makes assumptions about where files are located, and assumptions being made do not only affect reference genome files. Bioinformatics programs that take in some sort of index file frequently assume that index file is located in the same directory as the non-index input. For example, if you were to pass in SAMN1234.bam into [covstats](https://github.com/brentp/goleft/tree/master/covstats), it would expect an index file named SAMN1234.bam.bai or SAMN1234.bai in the same directory as the bam file, [as seen in the source code here](https://github.com/brentp/goleft/blob/fa6b00d20d1f73a068ffbab49a5769d173cae56d/covstats/covstats.go#L239). As there is no way to specify that the index file manually, you need to take that into consideration when writing WDLs involving covstats, bwa, and other similar tools.

Thankfully, the solution here is simple: Move all of the input files directly into the working directory.

task BwaMem {  
 input {  
 File input\_fastq  
 File ref\_fasta  
 File ref\_fasta\_index  
 File ref\_dict  
 File ref\_amb  
 File ref\_ann  
 File ref\_bwt  
 File ref\_pac  
 File ref\_sa  
 }  
  
 command <<<  
 set -eo pipefail  
  
 # This can also be done by creating an array and then looping that array,  
 # but we'll do it one line at a time or clarity's sake.  
 mv "~{ref\_fasta}" .  
 mv "~{ref\_fasta\_index}" .  
 mv "~{ref\_dict}" .  
 mv "~{ref\_amb}" .  
 mv "~{ref\_ann}" .  
 mv "~{ref\_bwt}" .  
 mv "~{ref\_pac}" .  
 mv "~{ref\_sa}" .  
  
 bwa mem \  
 [...]  
 >>>  
}

Some backends/executors do not support mv acting on input files. If you are running into problems with this and are working with miniwdl, the --copy-input-files flag will usually allow mv to work. You could also simply use cp to copy the files instead of move them, although this may not be an efficient use of disk space, so consider using mv if your target backends and executors can handle it.

With our files now all in the working directory, we can turn our attention to the bwa task itself. We can no longer directly pass in ~{ref\_fasta} or any of the other files we moved into the working directory, because those variables will point to a non-existent file in a now-empty input directory. There are several ways to solve this problem:

* Assuming the filename of an input is constant, which might be a safe assumption for reference files
* Using the bash built-in basename function
* Using the WDL built-in basename() function along with private variables

We recommend using the last option, as it works for essentially any input and may be more intuitive than the bash basename function. [OpenWDL explains](https://docs.openwdl.org/en/stable/WDL/basename/) how basename() works. The next section will provide an example of using it alongside private variables.

### 3.1.3 Private variables

Is there a variable you wish to use in your task section that is based on another input variable, or do not want people using your workflow to be able to directly overwrite? You can define variables outside the input {} section to create variables that function like private variables. In our case, we create String ref\_fasta\_local as ref\_fasta’s file base name to refer to the files we have moved to the working directory. We also create String base\_file\_name as input\_fastq’s file base name and use it to name our output files, such as "~{base\_file\_name}.sorted\_query\_aligned.bam". The variables read\_group\_id, sample\_name, and platform\_info are created similarly.

task BwaMem {  
 input {  
 File input\_fastq  
 File ref\_fasta  
 File ref\_fasta\_index  
 File ref\_dict  
 File ref\_amb  
 File ref\_ann  
 File ref\_bwt  
 File ref\_pac  
 File ref\_sa  
 }  
   
 # basename() is a built-in WDL function that acts like bash's basename  
 String base\_file\_name = basename(input\_fastq, ".fastq")  
 String ref\_fasta\_local = basename(ref\_fasta)  
   
 String read\_group\_id = "ID:" + base\_file\_name  
 String sample\_name = "SM:" + base\_file\_name  
 String platform\_info = "PL:illumina"  
  
 command <<<  
 set -eo pipefail  
  
 mv "~{ref\_fasta}" .  
 mv "~{ref\_fasta\_index}" .  
 mv "~{ref\_dict}" .  
 mv "~{ref\_amb}" .  
 mv "~{ref\_ann}" .  
 mv "~{ref\_bwt}" .  
 mv "~{ref\_pac}" .  
 mv "~{ref\_sa}" .  
  
  
 bwa mem \  
 -p -v 3 -t 16 -M -R '@RG\t~{read\_group\_id}\t~{sample\_name}\t~{platform\_info}' \  
 ~{ref\_fasta\_local} ~{input\_fastq} > ~{base\_file\_name}.sam   
 samtools view -1bS -@ 15 -o ~{base\_file\_name}.aligned.bam ~{base\_file\_name}.sam  
 samtools sort -@ 15 -o ~{base\_file\_name}.sorted\_query\_aligned.bam ~{base\_file\_name}.aligned.bam  
  
 >>>  
}

## 3.2 Runtime attributes

The runtime attributes of a task tell the WDL executor important information about how to run the task. For a bwa mem task, we want to make sure we have plenty of hardware resources available. We also need to include a reference to the docker image we want the task to actually run in.

runtime {  
 memory: "48 GB"  
 cpu: 16  
 docker: "ghcr.io/getwilds/bwa:0.7.17"  
 disks: "local-disk 100 SSD"  
 }

In WDL 1.0, the interpretation of runtime attributes by different executors and backends is extremely varied. The [WDL 1.0 spec](https://github.com/openwdl/wdl/blob/main/versions/1.0/SPEC.md#runtime-section) allows for arbitrary values here:

Individual backends will define which keys they will inspect so a key/value pair may or may not actually be honored depending on how the task is run. Values can be any expression and it is up to the engine to reject keys and/or values that do not make sense in that context.

This can lead to some pitfalls:

* Some of the attributes in your task’s runtime section may be silently ignored, such as the memory attribute when running Cromwell on the Fred Hutch HPC (as of Feb 2024)
* Some runtime attributes that are unique to particular backends, such as the Fred Hutch HPC’s walltime attribute
* The same runtime attribute working differently on different backends, such as disks acting differently on Cromwell depending on whether it is running on AWS or GCP

When writing WDL 1.0 workflows with specific hardware requirements, keep in mind what your backend and executor is able to interpret. It is also helpful to consider that other people running your workflow may be doing so on different backends and executors. More information can be found in the appendix, where we talk about designing WDLs for specific backends. For now, we will stick with memory, cpu, docker, and disks as this group of four runtime attributes will help us run this workflow on the majority of backends and executors. Even though the Fred Hutch HPC will ignore the memory and disks attributes, for instance, their inclusion will not cause the workflow to fail, but they will allow the workflow to run on Terra.

Some differences between WDL 1.0 and 1.1 on Runtime attributes

Although the focus of this course is on WDL 1.0, it is worth noting that in the [WDL 1.1 spec](https://github.com/openwdl/wdl/blob/main/versions/1.1/SPEC.md#runtime-section), a very different approach to runtime attributes is taken:

There are a set of reserved attributes (described below) that must be supported by the execution engine, and which have well-defined meanings and default values. Default values for all optional standard attributes are directly defined by the WDL specification in order to encourage portability of workflows and tasks; execution engines should NOT provide additional mechanisms to set default values for when no runtime attributes are defined.

If you are writing WDLs under the WDL 1.1 standard, you may have more flexibility with runtime attributes. Be aware that as of February 2024, Cromwell does not support WDL 1.1.

### 3.2.1 Docker images and containers

WDL is built to make use of Docker as it makes handling software dependencies much simpler. Docker images can help address all of these situations:

* Some software is difficult to install or compile on certain systems
* Some programs have conflicting dependencies
* You may not want to directly install software on your system to prevent it from breaking existing software
* You may not have permission to install software if you are using an institute HPC or other shared resource

When you run a WDL task that has a docker runtime attribute, your task will be executed in a Docker container sandbox environment. This container sandbox is derived from a template called a *Docker image*, which packages installed software in a special file system. This is one of the main features of a Docker image – because a Docker image packages the software you need, you can skip much of the installation and dependency issues associated with using new software. Because you take actions within a Docker container sandbox, it’s unlikely for you to “mess up” your main system’s files. Although a Docker container is, strictly speaking, not the same as a virtual machine, it is helpful to think of it as one if you are new to Docker. Docker containers are managed by Docker Engine/Apptainer, and the official Docker GUI is called Docker Desktop.

More information on finding and developing Docker images

Although you will generally need to be able to run Docker in order to run WDLs, you do not need to know how to create Dockerfiles – plaintext files which compile Docker images when run via docker build – to write your own WDLs. Most popular bioinformatics software packages already have ready-to-use Docker images available, which you can typically find on [Docker Hub](https://hub.docker.com/search?q=). Other registries include quay.io and the Google Container Registry. With that being said, if you would like to create your own Docker images, there are many tutorials and [guidelines](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/) available online. You can also learn more about the details of Docker (and why they technically aren’t virtual machines) in [Docker’s official curriculum](https://docker-curriculum.com/#introduction).

## 3.3 Outputs

The outputs of a task are defined in the output section of your task. Typically, this will take the form of directly outputting a file that was created in the command section. When these file outputs are referenced in the output section, you can refer to their path in the Docker container directly. You can also make outputs a function of input variables, including private input variables. This can be helpful if you intend on running this WDL on many different files – each one will get a unique filename based on the input fastq, instead of every sample ending up being named something generic like “converted.sam”. For our bwa mem task, one way to write the output section would be as follows:

output {  
 File analysisReadySorted = "~{base\_file\_name}.sorted\_query\_aligned.bam"  
 }

Another way of writing this is with string concatenation. This is equivalent to what we wrote above – choose whichever version you prefer.

output {  
 File analysisReadySorted = base\_file\_name + ".sorted\_query\_aligned.bam"  
 }

If the output was not in the working directory, we would need to change the output to point to the file’s path relative to the working directory, such as File analysisReadySorted = "some\_folder/~{base\_file\_name}.sorted\_query\_aligned.bam".

Below are some some additional ways you can handle task outputs.

Ouputs as functions of other outputs in the same task

Outputs can (generally, see warning below) also be functions of other outputs in the same task, as long as those outputs are declared first.

task add\_one {  
 input {  
 Int some\_integer  
 }  
 command <<<  
 echo ~{some\_integer} > a.txt  
 echo "1" > b.txt  
 >>>  
 output {  
 Int a = read\_int("a.txt")  
 Int b = read\_int("b.txt")  
 Int c = a + b  
 }  
}

Cromwell does not fully support outputs being a function of the same task’s other outputs. On the Terra backend, the above code example would cause an error.

Grabbing multiple outputs at the same time

To grab multiple outputs at the same time, use glob() to create an array of files. We’ll also take this opportunity to demonstrate iterating through a bash array created from an Array[String] input – for more information on this data type, see chapter six of this course.

task one\_word\_per\_file {  
 input {  
 Array[String] a\_big\_sentence  
 }  
 command <<<  
 ARRAY\_OF\_WORDS=(~{sep=" " a\_big\_sentence})  
 i=0  
 for word in "${!ARRAY\_OF\_WORDS[@]}"  
 do  
 i=$((i+1))  
 echo $word >> $i.txt  
 done  
 >>>  
 output {  
 Array[File] several\_words = glob("\*.txt")  
 }  
}

glob() can also be used to grab just one file via glob("\*.txt")[0] to grab the first thing that matches the glob. This is usually only necessary if you know the extension of an output, but do not have a way of predicting the rest of its filename. Be aware that if anything else in the working directory has the extension you are searching for, you might accidentally grab that one instead of the one you are looking for!

## 3.4 The whole task

We’ve now designed a bwa mem task that can run on almost any backend that supports WDL and can handle the hardware requirements. Issues involving bwa mem expecting reference files to be in the same folder and/or putting output files into input folders have been sidestepped thanks to careful design and consideration. The runtime section clearly defines the expected hardware requirements, and the outputs section defines what we expect the task to give us when all is said and done. We’re now ready to continue with the rest of our workflow.

task BwaMem {  
 input {  
 File input\_fastq  
 referenceGenome refGenome  
 }  
   
 String base\_file\_name = basename(input\_fastq, ".fastq")  
 String ref\_fasta\_local = basename(refGenome.ref\_fasta)  
  
 String read\_group\_id = "ID:" + base\_file\_name  
 String sample\_name = "SM:" + base\_file\_name  
 String platform\_info = "PL:illumina"  
  
 command <<<  
 set -eo pipefail  
  
 mv ~{refGenome.ref\_fasta} .  
 mv ~{refGenome.ref\_fasta\_index} .  
 mv ~{refGenome.ref\_dict} .  
 mv ~{refGenome.ref\_amb} .  
 mv ~{refGenome.ref\_ann} .  
 mv ~{refGenome.ref\_bwt} .  
 mv ~{refGenome.ref\_pac} .  
 mv ~{refGenome.ref\_sa} .  
  
 bwa mem \  
 -p -v 3 -t 16 -M -R '@RG\t~{read\_group\_id}\t~{sample\_name}\t~{platform\_info}' \  
 ~{ref\_fasta\_local} ~{input\_fastq} > ~{base\_file\_name}.sam   
 samtools view -1bS -@ 15 -o ~{base\_file\_name}.aligned.bam ~{base\_file\_name}.sam  
 samtools sort -@ 15 -o ~{base\_file\_name}.sorted\_query\_aligned.bam ~{base\_file\_name}.aligned.bam  
 >>>  
  
 output {  
 File analysisReadySorted = "~{base\_file\_name}.sorted\_query\_aligned.bam"  
 }  
   
 runtime {  
 memory: "48 GB"  
 cpu: 16  
 docker: "ghcr.io/getwilds/bwa:0.7.17"  
 }  
}

## 3.5 Putting the workflow together

A workflow is needed to run the BwaMem task we just built. The workflow’s input variables are defined by the workflow JSON metadata, and are then passed on as inputs in our BwaMem call. When the BwaMem call is complete, the workflow’s output File variable is defined based on the task’s output. Lastly, we have a parameter\_meta component in our workflow that describes each workflow input variable as documentation.

For the workflow to actually “see” the task, the task will either need to be imported at the top of the workflow (just under the version 1.0 string), or included in the same file as the workflow. For simplicity, we will put the workflow and the task in the same file.

## 3.6 Testing your first task

To test your first task and your workflow, you should have expectation of output is. For this first BwaMem task, we just care that the BAM file is created with aligned reads. You can use samtools view output.sorted\_query\_aligned.bam to examine the reads and pipe it to wordcount wc to get the number of total reads. This number should be almost identical as the number of reads from your input FASTQ file if you run wc input.fastq. In other tasks, we might have a more precise expectation of what the output file should be, such as containing the specific somatic mutation call that we have curated.

Here is an example JSON with the [test data](https://figshare.com/articles/dataset/WDL_101_Dataset/25447528) needed to run this single-task workflow:

{  
 "mutation\_calling.sampleFastq": "/path/to/Tumor\_2\_EGFR\_HCC4006\_combined.fastq",  
 "mutation\_calling.ref\_fasta": "/path/to/Homo\_sapiens\_assembly19.fasta",  
 "mutation\_calling.ref\_fasta\_index": "/path/to/Homo\_sapiens\_assembly19.fasta.fai",  
 "mutation\_calling.ref\_dict": "/path/to/Homo\_sapiens\_assembly19.dict",  
 "mutation\_calling.ref\_pac": "/path/to/Homo\_sapiens\_assembly19.fasta.pac",  
 "mutation\_calling.ref\_sa": "/path/to/Homo\_sapiens\_assembly19.fasta.sa",  
 "mutation\_calling.ref\_amb": "/path/to/Homo\_sapiens\_assembly19.fasta.amb",  
 "mutation\_calling.ref\_ann": "/path/to/Homo\_sapiens\_assembly19.fasta.ann",  
 "mutation\_calling.ref\_bwt": "/path/to/Homo\_sapiens\_assembly19.fasta.bwt"  
}

The example JSON using the Fred Hutch HPC

{ “mutation\_calling.sampleFastq”: “/fh/fast/paguirigan\_a/pub/ReferenceDataSets/workflow\_testing\_data/WDL/wdl\_101/HCC4006\_final.fastq”, “mutation\_calling.ref\_fasta”: “/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta”, “mutation\_calling.ref\_fasta\_index”: “/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.fai”, “mutation\_calling.ref\_dict”: “/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.dict”, “mutation\_calling.ref\_pac”: “/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.pac”, “mutation\_calling.ref\_sa”: “/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.sa”, “mutation\_calling.ref\_amb”: “/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.amb”, “mutation\_calling.ref\_ann”: “/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.ann”, “mutation\_calling.ref\_bwt”: “/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.bwt” }

Loading…

# 4 Connecting multiple tasks together in a linear chain

Now that you have a first task in a workflow up and running, the next step is to continue building out the workflow as described in our Workflow Plan:

1. BwaMem aligns the samples to the reference genome.
2. MarkDuplicates marks PCR duplicates.
3. ApplyBaseRecalibrator applies base recalibration.
4. Mutect2 performs somatic mutation calling. For this current iteration, we start with Mutect2TumorOnly which only uses the tumor sample. Later on, we will switch to a version that will allow for comparing tumor samples against a non-tumor normal sample.
5. annovar annotates the called somatic mutations.

We do this via a **linear chain**, in which we feed the output of one task to the input of the next task. Let’s see how to build a linear chain in a workflow.

## 4.1 How to connect tasks together in a workflow

We can easily connect tasks together because of the following: *The output variables of a task can be accessed at the workflow level as inputs for the subsequent task.*

For instance, let’s see the output of our BwaMem task.

output {  
 File analysisReadyBam = "~{base\_file\_name}.aligned.bam"  
 File analysisReadySorted = "~{base\_file\_name}.sorted\_query\_aligned.bam"  
 }

The File variables analysisReadyBam and analysisReadySorted can now be accessed anywhere in the workflow block after the BwaMem task as BwaMem.analysisReadyBam and BwaMem.analysisReadySorted, respectively. Therefore, when we call the MarkDuplicates task, we can pass it the input BwaMem.analysisReadySorted from the BwaMem task:

workflow mutation\_calling {  
 input {  
 File sampleFastq  
  
 # Reference genome  
 File ref\_fasta  
 File ref\_fasta\_index  
 File ref\_dict  
 File ref\_amb  
 File ref\_ann  
 File ref\_bwt  
 File ref\_pac  
 File ref\_sa  
 }  
   
  
 # Map reads to reference  
 call BwaMem {  
 input:  
 input\_fastq = sampleFastq,  
 ref\_fasta = ref\_fasta,  
 ref\_fasta\_index = ref\_fasta\_index,  
 ref\_dict = ref\_dict,  
 ref\_amb = ref\_amb,  
 ref\_ann = ref\_ann,  
 ref\_bwt = ref\_bwt,  
 ref\_pac = ref\_pac,  
 ref\_sa = ref\_sa  
 }  
   
 call MarkDuplicates {  
 input:  
 input\_bam = BwaMem.analysisReadySorted  
 }  
}

Resources:

For a basic introduction to linear chain, see [OpenWDL Docs](https://docs.openwdl.org/en/latest/WDL/Linear_chaining/)’ introduction.

To see other examples of linear chain and variations, see [OpenWDL Docs](https://docs.openwdl.org/en/latest/WDL/add_plumbing/)’s section on workflow plumbing.

## 4.2 Writing MarkDuplicates task

Of course, the task MarkDuplicates hasn’t been written yet! Let’s go through it together:

### 4.2.1 Input

The task takes an input bam file that has been aligned to the reference genome. It needs to be a File input\_bam based on how we introduced it in the workflow above. That is easy to write up:

task MarkDuplicates {  
 input {  
 File input\_bam  
 }  
}

### 4.2.2 Private variables in the task

Similar to the BwaMem task, we will name our output files based on the base name of the original input file in the workflow. Therefore, it makes sense to create a private String variable base\_file\_name that contains this base name of input\_bam. We will use base\_file\_name in the Command section to specify the output bam and metrics files.

task MarkDuplicates {  
 input {  
 File input\_bam  
 }  
 String base\_file\_name = basename(input\_bam, ".sorted\_query\_aligned.bam")  
}

### 4.2.3 Command

As we have been doing all along, we refer to any defined variables from the input or private variables using ~{this} syntax.

task MarkDuplicates {  
 input {  
 File input\_bam  
 }  
  
 String base\_file\_name = basename(input\_bam, ".sorted\_query\_aligned.bam")  
  
 command <<<  
 gatk MarkDuplicates \  
 --INPUT "~{input\_bam}" \  
 --OUTPUT "~{base\_file\_name}.duplicates\_marked.bam" \  
 --METRICS\_FILE "~{base\_file\_name}.duplicate\_metrics" \  
 --CREATE\_INDEX true \  
 --OPTICAL\_DUPLICATE\_PIXEL\_DISTANCE 100 \  
 --VALIDATION\_STRINGENCY SILENT  
 >>>  
}

### 4.2.4 Runtime and Output

We specify a different Docker image that contains the GATK software, and the relevant computing needs. We also specify three different output files, two of which are specified in the command section, and the third is a bam index file that is automatically created by the command section.

Below is the task all together. It has a form very similar to our first task BwaMem.

task MarkDuplicates {  
 input {  
 File input\_bam  
 }  
  
 String base\_file\_name = basename(input\_bam, ".sorted\_query\_aligned.bam")  
  
 command <<<  
 gatk MarkDuplicates \  
 --INPUT "~{input\_bam}" \  
 --OUTPUT "~{base\_file\_name}.duplicates\_marked.bam" \  
 --METRICS\_FILE "~{base\_file\_name}.duplicate\_metrics" \  
 --CREATE\_INDEX true \  
 --OPTICAL\_DUPLICATE\_PIXEL\_DISTANCE 100 \  
 --VALIDATION\_STRINGENCY SILENT  
 >>>  
   
 runtime {  
 docker: "broadinstitute/gatk:4.1.4.0"  
 memory: "48 GB"  
 cpu: 4  
 }  
   
 output {  
 File markDuplicates\_bam = "~{base\_file\_name}.duplicates\_marked.bam"  
 File markDuplicates\_bai = "~{base\_file\_name}.duplicates\_marked.bai"  
 File duplicate\_metrics = "~{base\_file\_name}.duplicate\_metrics"  
 }  
}

### 4.2.5 Testing the workflow

As before, when you add a new task to the workflow, you should always test that it works on your test sample. To check that MarkDuplicates is indeed marking PCR duplicates, you could check for the presence of the [PCR duplicate flag in reads](https://gatk.broadinstitute.org/hc/en-us/articles/360051306171-MarkDuplicates-Picard), which has a decimal value of 1024 in the SAM Flags Field.

## 4.3 The rest of the linear chain workflow

We build out the rest of the tasks in a very similar fashion. Tasks ApplyBaseRecalibrator and Mutect2TumorOnly both have files that need to be localized, but otherwise all the tasks have a very similar form as BwaMem and MarkDuplicates. For this current iteration, we use only the tumor sample for mutation calling. In the following chapters, we will use additional WDL features to make use of tumor and normal samples for mutation calling.

We also expand our input JSON metadata to have files needed for each task:

{  
 "mutation\_calling.sampleFastq": "/path/to/Tumor\_2\_EGFR\_HCC4006\_combined.fastq",  
 "mutation\_calling.ref\_fasta": "/path/to/Homo\_sapiens\_assembly19.fasta",  
 "mutation\_calling.ref\_fasta\_index": "/path/to/Homo\_sapiens\_assembly19.fasta.fai",  
 "mutation\_calling.ref\_dict": "/path/to/Homo\_sapiens\_assembly19.dict",  
 "mutation\_calling.ref\_pac": "/path/to/Homo\_sapiens\_assembly19.fasta.pac",  
 "mutation\_calling.ref\_sa": "/path/to/Homo\_sapiens\_assembly19.fasta.sa",  
 "mutation\_calling.ref\_amb": "/path/to/Homo\_sapiens\_assembly19.fasta.amb",  
 "mutation\_calling.ref\_ann": "/path/to/Homo\_sapiens\_assembly19.fasta.ann",  
 "mutation\_calling.ref\_bwt": "/path/to/Homo\_sapiens\_assembly19.fasta.bwt",  
 "mutation\_calling.ref\_name": "hg19",  
 "mutation\_calling.dbSNP\_vcf\_index": "/path/to/dbsnp\_138.b37.vcf.gz.tbi",  
 "mutation\_calling.dbSNP\_vcf": "/path/to/dbsnp\_138.b37.vcf.gz",  
 "mutation\_calling.known\_indels\_sites\_indices": "/path/to/Mills\_and\_1000G\_gold\_standard.indels.b37.sites.vcf.idx",  
 "mutation\_calling.known\_indels\_sites\_VCFs": "/path/to/Mills\_and\_1000G\_gold\_standard.indels.b37.sites.vcf",  
 "mutation\_calling.af\_only\_gnomad": "/path/to/af-only-gnomad.raw.sites.b37.vcf.gz",  
 "mutation\_calling.af\_only\_gnomad\_index": "/path/to/af-only-gnomad.raw.sites.b37.vcf.gz.tbi",  
 "mutation\_calling.annovar\_protocols": "refGene,knownGene,cosmic70,esp6500siv2\_all,clinvar\_20180603,gnomad211\_exome",  
 "mutation\_calling.annovar\_operation": "g,f,f,f,f,f"  
}

The JSON using the Fred Hutch HPC

{  
 "mutation\_calling.sampleFastq": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/workflow\_testing\_data/WDL/wdl\_101/HCC4006\_final.fastq",  
 "mutation\_calling.ref\_fasta": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta",  
 "mutation\_calling.ref\_fasta\_index": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.fai",  
 "mutation\_calling.ref\_dict": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.dict",  
 "mutation\_calling.ref\_pac": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.pac",  
 "mutation\_calling.ref\_sa": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.sa",  
 "mutation\_calling.ref\_amb": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.amb",  
 "mutation\_calling.ref\_ann": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.ann",  
 "mutation\_calling.ref\_bwt": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.bwt",  
 "mutation\_calling.ref\_name": "hg19",  
 "mutation\_calling.dbSNP\_vcf\_index": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/dbsnp\_138.b37.vcf.gz.tbi",  
 "mutation\_calling.dbSNP\_vcf": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/dbsnp\_138.b37.vcf.gz",  
 "mutation\_calling.known\_indels\_sites\_indices": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Mills\_and\_1000G\_gold\_standard.indels.b37.sites.vcf.idx",  
 "mutation\_calling.known\_indels\_sites\_VCFs": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Mills\_and\_1000G\_gold\_standard.indels.b37.sites.vcf",  
 "mutation\_calling.af\_only\_gnomad": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/af-only-gnomad.raw.sites.b37.vcf.gz",  
 "mutation\_calling.af\_only\_gnomad\_index": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/af-only-gnomad.raw.sites.b37.vcf.gz.tbi",  
 "mutation\_calling.annovar\_protocols": "refGene,knownGene,cosmic70,esp6500siv2\_all,clinvar\_20180603,gnomad211\_exome",  
 "mutation\_calling.annovar\_operation": "g,f,f,f,f,f"  
}

Loading…

# 5 Organizing variables via Structs

In our workflow so far, we see that certain variables are always used together, even for different tasks. For example, variables related to the reference genome are always used for the same purpose and passed on to tasks in almost the same way. This leads to quite a bit of coding redundancy, as when we write down the large set of variables related to the reference genome as task inputs, we are just thinking about one entity. We don’t make distinctions of the reference genome files until the task body itself.

To improve code organization and readability, we can package all variables related to the reference genome into a compound data structure called a **struct**. With a struct variable, we can refer all the packaged variables as one single variable, and also refer to specific variables within the struct without losing any information. [OpenWDL Docs](https://docs.openwdl.org/en/stable/WDL/using_structs/) also has an excellent introduction and examples on structs.

To define a struct, we must declare it outside of a workflow and task:

struct referenceGenome {  
 File ref\_fasta  
 File ref\_fasta\_index  
 File ref\_dict  
 File ref\_amb  
 File ref\_ann  
 File ref\_bwt  
 File ref\_pac  
 File ref\_sa  
 String ref\_name  
}  
  
workflow mutation\_calling {  
 input {  
 File sampleFastq  
 referenceGenome refGenome ## our struct  
 ...  
 }  
   
 # Map reads to reference  
 call BwaMem {  
 input:  
 input\_fastq = sampleFastq,  
 refGenome = refGenome ## our struct   
 }  
}

The referenceGenome struct contains all the variables related to the reference genome, but values cannot be defined here. The struct definition merely lays the skeleton components of the data structure, but contains no actual values.

In our workflow inputs, we remove all of the File variables associated with reference genome definitions, but keep anything that isn’t related to the reference genome, such as sampleFastq. We instead declare a referenceGenome struct variable called refGenome via referenceGenome refGenome. We can access the variables within a struct by the following syntax: structVar.varName, such as refGenome.ref\_name. The [WDL spec](https://github.com/openwdl/wdl/blob/main/versions/1.0/SPEC.md#struct-definition) has more information on how to define and use structs.

To give values to refGenome, we need to modify our JSON metadata file. We define the refGenome variable in a nested structure that corresponds to the referenceGenome struct. Let’s take a look:

{  
 "mutation\_calling.refGenome": {  
 "ref\_fasta": "/path/to/Homo\_sapiens\_assembly19.fasta",  
 "ref\_fasta\_index": "/path/to/Homo\_sapiens\_assembly19.fasta.fai",  
 "ref\_dict": "/path/to/Homo\_sapiens\_assembly19.dict",  
 "ref\_pac": "/path/to/Homo\_sapiens\_assembly19.fasta.pac",  
 "ref\_sa": "/path/to/Homo\_sapiens\_assembly19.fasta.sa",  
 "ref\_amb": "/path/to/Homo\_sapiens\_assembly19.fasta.amb",  
 "ref\_ann": "/path/to/Homo\_sapiens\_assembly19.fasta.ann",  
 "ref\_bwt": "/path/to/Homo\_sapiens\_assembly19.fasta.bwt",  
 "ref\_name": "hg19"  
 },  
 "mutation\_calling.dbSNP\_vcf\_index": "/path/to/dbsnp\_138.b37.vcf.gz.tbi",  
 ...  
}

Now refGenome has all the values it needs for our tasks.

In addition, we have replaced all the reference genome inputs in call BwaMem with refGenome in order to pass information to a task via structs.

Within the BwaMem task, we must refer to variables inside the struct, such as refGenome.ref\_name (which has a value of “hg19” using this JSON metadata):

# Align fastq file to the reference genome  
task BwaMem {  
 input {  
 File input\_fastq  
 referenceGenome refGenome  
 }  
   
 String base\_file\_name = basename(input\_fastq, ".fastq")  
 String ref\_fasta\_local = basename(refGenome.ref\_fasta)  
  
 String read\_group\_id = "ID:" + base\_file\_name  
 String sample\_name = "SM:" + base\_file\_name  
 String platform\_info = "PL:illumina"  
  
  
 command <<<  
 set -eo pipefail  
  
 mv "~{refGenome.ref\_fasta}" .  
 mv "~{refGenome.ref\_fasta\_index}" .  
 mv "~{refGenome.ref\_dict}" .  
 mv "~{refGenome.ref\_amb}" .  
 mv "~{refGenome.ref\_ann}" .  
 mv "~{refGenome.ref\_bwt}" .  
 mv "~{refGenome.ref\_pac}" .  
 mv "~{refGenome.ref\_sa}" .  
  
 bwa mem \  
 -p -v 3 -t 16 -M -R '@RG\t~{read\_group\_id}\t~{sample\_name}\t~{platform\_info}' \  
 "~{ref\_fasta\_local}" "~{input\_fastq}" > "~{base\_file\_name}.sam"  
 samtools view -1bS -@ 15 -o "~{base\_file\_name}.aligned.bam" "~{base\_file\_name}.sam"  
 samtools sort -@ 15 -o "~{base\_file\_name}.sorted\_query\_aligned.bam" "~{base\_file\_name}.aligned.bam"  
 >>>  
  
 output {  
 File analysisReadySorted = "~{base\_file\_name}.sorted\_query\_aligned.bam"  
 }  
   
 runtime {  
 memory: "48 GB"  
 cpu: 16  
 docker: "ghcr.io/getwilds/bwa:0.7.17"  
 }  
}

Other tasks in the workflow, such as ApplyBaseRecalibrator and Mutect2TumorOnly also make use of the reference genome, so we pass refGenome to it. The final task annovar only requires the reference genome name, and none of the files in the referenceGenome struct. We make a stylistic choice to pass only refGenome.ref\_name to the input of annovar task call, as the task doesn’t need the full information of the struct. This stylistic choice is based on the principle of passing on the minimally needed information for a modular piece of code to run, which makes the task annovar depend on the minimal amount of inputs. This will also save us time and disk space by not having to localize several gigabytes of reference files into the Docker container that annovar will be running in.

call annovar {  
 input:  
 input\_vcf = Mutect2TumorOnly.output\_vcf,  
 ref\_name = refGenome.ref\_name,  
 annovar\_operation = annovar\_operation,  
 annovar\_protocols = annovar\_protocols  
 }

Putting everything together in the workflow:

The JSON metadata:

{  
 "mutation\_calling.sampleFastq": "/path/to/Tumor\_2\_EGFR\_HCC4006\_combined.fastq",  
 "mutation\_calling.refGenome": {  
 "ref\_fasta": "/path/to/Homo\_sapiens\_assembly19.fasta",  
 "ref\_fasta\_index": "/path/to/Homo\_sapiens\_assembly19.fasta.fai",  
 "ref\_dict": "/path/to/Homo\_sapiens\_assembly19.dict",  
 "ref\_pac": "/path/to/Homo\_sapiens\_assembly19.fasta.pac",  
 "ref\_sa": "/path/to/Homo\_sapiens\_assembly19.fasta.sa",  
 "ref\_amb": "/path/to/Homo\_sapiens\_assembly19.fasta.amb",  
 "ref\_ann": "/path/to/Homo\_sapiens\_assembly19.fasta.ann",  
 "ref\_bwt": "/path/to/Homo\_sapiens\_assembly19.fasta.bwt",  
 "ref\_name": "hg19"  
 },  
 "mutation\_calling.dbSNP\_vcf\_index": "/path/to/dbsnp\_138.b37.vcf.gz.tbi",  
 "mutation\_calling.dbSNP\_vcf": "/path/to/dbsnp\_138.b37.vcf.gz",  
 "mutation\_calling.known\_indels\_sites\_indices": "/path/to/Mills\_and\_1000G\_gold\_standard.indels.b37.sites.vcf.idx",  
 "mutation\_calling.known\_indels\_sites\_VCFs": "/path/to/Mills\_and\_1000G\_gold\_standard.indels.b37.sites.vcf",  
 "mutation\_calling.af\_only\_gnomad": "/path/to/af-only-gnomad.raw.sites.b37.vcf.gz",  
 "mutation\_calling.af\_only\_gnomad\_index": "/path/to/af-only-gnomad.raw.sites.b37.vcf.gz.tbi",  
 "mutation\_calling.annovar\_protocols": "refGene,knownGene,cosmic70,esp6500siv2\_all,clinvar\_20180603,gnomad211\_exome",  
 "mutation\_calling.annovar\_operation": "g,f,f,f,f,f"  
}

The JSON using the Fred Hutch HPC

{  
 "mutation\_calling.sampleFastq": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/workflow\_testing\_data/WDL/wdl\_101/HCC4006\_final.fastq",  
 "mutation\_calling.refGenome": {  
 "ref\_fasta": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta",  
 "ref\_fasta\_index": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.fai",  
 "ref\_dict": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.dict",  
 "ref\_pac": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.pac",  
 "ref\_sa": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.sa",  
 "ref\_amb": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.amb",  
 "ref\_ann": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.ann",  
 "ref\_bwt": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.bwt",  
 "ref\_name": "hg19"  
 },  
 "mutation\_calling.dbSNP\_vcf\_index": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/dbsnp\_138.b37.vcf.gz.tbi",  
 "mutation\_calling.dbSNP\_vcf": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/dbsnp\_138.b37.vcf.gz",  
 "mutation\_calling.known\_indels\_sites\_indices": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Mills\_and\_1000G\_gold\_standard.indels.b37.sites.vcf.idx",  
 "mutation\_calling.known\_indels\_sites\_VCFs": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Mills\_and\_1000G\_gold\_standard.indels.b37.sites.vcf",  
 "mutation\_calling.af\_only\_gnomad": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/af-only-gnomad.raw.sites.b37.vcf.gz",  
 "mutation\_calling.af\_only\_gnomad\_index": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/af-only-gnomad.raw.sites.b37.vcf.gz.tbi",  
 "mutation\_calling.annovar\_protocols": "refGene,knownGene,cosmic70,esp6500siv2\_all,clinvar\_20180603,gnomad211\_exome",  
 "mutation\_calling.annovar\_operation": "g,f,f,f,f,f"  
}

Loading…

# 6 Parallelization via Arrays

We have a workflow that runs on a single sample. What if we want to process multiple samples at once? Let’s look at the various ways we can run our workflow more efficiently, as well as processing many samples in parallel. This is where WDL really shines.

In this chapter, we’ll be going over:

* How to use scattered tasks to run a workflow on multiple samples at once
* How to use arrays effectively
* How to reference arrays in a task’s command section
* How arrays differ from Structs

## 6.1 The array type

Arrays are essentially lists of another [primitive type](https://en.wikipedia.org/wiki/Primitive_data_type). It is most common to see Array[File] in WDLs, but an array can contain integers, floats, strings, and the like. An array can only have one of a given primitive type. For example, an Array[String] could contain the strings “cat” and “dog” but not the integer 1965 (however, it could have “1965” as a string).

In chapter 5, we went over the struct data type and used it to handle a myriad of reference genome files. Arrays differ from structs in that arrays are numerically indexed, which means that a member of the array can be accessed by its position in the array. On the other hand, each variable within a struct has its own name, and you use that name to reference it rather than a numerical index.

In WDL, arrays are 0 indexed, so the “first” value in an array is referenced by [0]. As per the WDL spec, arrays retain their order and are [immutable](https://en.wikipedia.org/wiki/Immutable_object) – if you explicitly define an Array[String] with the members [“foo”, “bar”, “bizz”], you can be confident that “foo” will always be at index 0.

Array[String] foobarbizz = ["foo", "bar", "bizz"]  
String foo = foobarbizz[0] # will always be "foo"

Because arrays are immutable in WDL, if you wish to add values to an array, you will need to define a new array. You can combine an existing array and new values using flatten(), a WDL built-in function that will turned a nested array into a “flat” array, like so:

Array[String] foobarbizz = ["foo", "bar", "bizz"]  
String foo = foobarbizz[0] # will always be "foo"  
Array[Array[String]] foobarbizz\_but\_with\_bonus\_foo = [foobarbizz, foo, foo, foo] # [["foo", "bar", "bizz"], "foo", "foo", "foo"]  
Array[String] foobarbizzfoofoofoo = flatten(foobarbizz\_but\_with\_bonus\_foo) # ["foo", "bar", "bizz", "foo", "foo", "foo"]

## 6.2 Scattered tasks

Scattered tasks allow us to run a WDL task in parallel. This is especially useful on highly scalable backends such as HPCs or the cloud, as it allows us to potentially run hundreds or even thousands of instances of a task at the same time. The most common use case for this is processing many samples at the same time, but it can also be used for processing a single sample’s chromosomes in parallel, or similar situations where breaking up data into discrete “chunks” makes sense.

It should be noted that a scattered task does not work the same way as multithreading, nor does it correlate with the cpu WDL runtime attribute. Every instance of a scattered task takes place in a new Docker image, and is essentially “unaware” of all other instances of that scattered task, with one exception: If an instance of a scattered task errors out, a WDL executor may attempt to shut down other ongoing instances of that scattered task.

### 6.2.1 Troubleshooting

Scattered tasks are relatively simple in theory, but the way they interact with optional types can be unintuitive. As a general rule, you should avoid using optional types as the input of a scattered task whenever possible.

Generally speaking, a WDL executor will try to run as many instances of a scattered task as it thinks your backend’s hardware can handle at once. Sometimes the WDL executor will overestimate what the backend is capable of and run too many instances of a scattered task at once. This almost never happens on scalable cloud-based backends such as Terra, but isn’t uncommon when running scattered tasks on a local machine.

## 6.3 Making our workflow run on multiple samples at once using scattered tasks and arrays

When we originally wrote our workflow, we designed it to only run on a single sample at a time. However, we’ll likely want to run this workflow on multiple samples at the same time. For some workflows, this is a great way to directly compare samples to each other, but for our purposes we simply want to avoid running a workflow 100 times if we can instead run one workflow that handles 100 samples at once.

For starters, we’ll want to change our workflow-level sample variables from File to Array[File]. However, we don’t need to change any of the reference genome files, because every instance of our tasks will be taking in the same reference genome files. In other words, our struct is unchanged.

version 1.0  
  
struct referenceGenome {  
 File ref\_fasta  
 File ref\_fasta\_index  
 File ref\_dict  
 File ref\_amb  
 File ref\_ann  
 File ref\_bwt  
 File ref\_pac  
 File ref\_sa  
 String ref\_name  
}  
  
  
workflow mutation\_calling {  
 input {  
 Array[File] tumorSamples  
  
 referenceGenome refGenome  
   
 File dbSNP\_vcf  
 File dbSNP\_vcf\_index  
 File known\_indels\_sites\_VCFs  
 File known\_indels\_sites\_indices  
 ...  
 }  
 ...  
}

Next, we will want to look at our chain of tasks. Each of these tasks are designed to take in a single sample. In theory, we could rewrite each task to iterate through an array of multiple samples. However, it’s much simpler to keep those tasks as single-sample tasks, and simply run them on one sample at a time. To do this, we encapsulate the task calls in the workflow document with scatter.

scatter (tumorFastq in tumorSamples) {  
 call BwaMem as tumorBwaMem {  
 input:  
 input\_fastq = tumorFastq,  
 refGenome = refGenome  
 }  
 call MarkDuplicates as tumorMarkDuplicates {  
 input:  
 input\_bam = tumorBwaMem.analysisReadySorted  
 }  
  
 call ApplyBaseRecalibrator as tumorApplyBaseRecalibrator{  
 input:  
 input\_bam = tumorMarkDuplicates.markDuplicates\_bam,  
 input\_bam\_index = tumorMarkDuplicates.markDuplicates\_bai,  
 dbSNP\_vcf = dbSNP\_vcf,  
 dbSNP\_vcf\_index = dbSNP\_vcf\_index,  
 known\_indels\_sites\_VCFs = known\_indels\_sites\_VCFs,  
 known\_indels\_sites\_indices = known\_indels\_sites\_indices,  
 refGenome = refGenome  
 }  
 }

A scatter is essentially the WDL version of a [for loop](https://en.wikipedia.org/wiki/For_loop). Every task within that loop will have access to a single File within the Array[File] that it is looping through. Within the scatter, downstream tasks can access outputs of upstream tasks like normal. So, the first tumor fastq file will go through BwaMem, then the resulting bam will go through MarkDuplicates, and the marked bam will undergo base recalibration. Since all three of these tasks are within the same scatter, each task only “sees” one sample at a time.

However, outside the scatter, every task is considered in the context of all samples, so every output of those scattered tasks becomes arrays. As a result, our workflow-level outputs are now Array[File] instead of just File.

output {  
 Array[File] tumoralignedBamSorted = tumorBwaMem.analysisReadySorted  
 Array[File] tumorMarkDuplicates\_bam = tumorMarkDuplicates.markDuplicates\_bam  
 Array[File] tumorMarkDuplicates\_bai = tumorMarkDuplicates.markDuplicates\_bai  
 Array[File] tumoranalysisReadyBam = tumorApplyBaseRecalibrator.recalibrated\_bam   
 Array[File] tumoranalysisReadyIndex = tumorApplyBaseRecalibrator.recalibrated\_bai  
 }

You can reference a full copy of this workflow at the end of this chapter.

## 6.4 Referencing an array in a task

In our example, each task only takes in one sample, so we are not directly inputting arrays into a file. However, it’s important to know how to input arrays in a task’s command section. If a task’s input variable is an array, we must include an array separator. In WDL 1.0, this is done using the sep= expression placeholder. Every value in the WDL Array[String] will be separated by whatever value is declared via sep. In this example, that is a simple space, as that is one way how to construct a bash variable.

task count\_words {  
 input {  
 Array[String] a\_big\_sentence  
 }  
 command <<<  
 ARRAY\_OF\_WORDS=(~{sep=" " a\_big\_sentence})  
 echo ${#ARRAY\_OF\_FILES[@]} >> length.txt  
 # Note how the bash array uses ${} syntax, which could quickly get  
 # confusing if we used that syntax for our WDL variables. This is  
 # why we recommend using tilde + {} for your WDL variables.  
 >>>  
}

It’s usually unnecessary to declare an Array[String], because a single String can have many words in it. That being said, an Array[String] can sometimes come in handy if it is made up of outputs from other tasks. We’ll talk more about chaining tasks together in upcoming chapters.

The WDL 1.1 spec added a new built-in function, sep(), which replaces the sep= expression placeholder for arrays. This same version of the spec also notes that the sep= expression placeholder [are deprecated and will be removed from future versions of WDL](https://github.com/openwdl/wdl/blob/main/versions/1.1/SPEC.md#-expression-placeholder-options). For the time being, we recommend sticking with sep= as it is compatible with both WDL 1.0 and WDL 1.1, even though it is technically deprecated in WDL 1.1.

If you’re not used to working in bash, the syntax for interacting with bash arrays can be unintuitive, but you don’t have to write a WDL’s command section only using bash. In fact, working in another language besides bash within a WDL can be a great way to write code quickly, or perform tasks that are more advanced than what a typical bash script can handle. Just be sure to set sep properly to ensure that your array is interpreted correctly. In this example, we place quotation marks before and after the variable to ensure that the first and last value of the list are given beginning and ending quotation marks respectively.

task count\_words\_python {  
 input {  
 Array[String] a\_big\_sentence  
 }  
 command <<<  
 python << CODE  
 sentence = [ "~{sep='", "' a\_big\_sentence}" ]  
 print(len(sentence))  
 CODE  
 >>>  
 runtime {  
 docker: "python:latest"  
 }  
}

## 6.5 The workflow so far

Let’s take another look at our workflow.

The JSON metadata:

{  
 "mutation\_calling.tumorSamples": ["/path/to/Tumor\_1\_KRAS\_CALU1\_combined\_final.fastq", "/path/to/Tumor\_2\_EGFR\_HCC4006\_combined.fastq"],  
 "mutation\_calling.refGenome": {  
 "ref\_fasta": "/path/to/Homo\_sapiens\_assembly19.fasta",  
 "ref\_fasta\_index": "/path/to/Homo\_sapiens\_assembly19.fasta.fai",  
 "ref\_dict": "/path/to/Homo\_sapiens\_assembly19.dict",  
 "ref\_pac": "/path/to/Homo\_sapiens\_assembly19.fasta.pac",  
 "ref\_sa": "/path/to/Homo\_sapiens\_assembly19.fasta.sa",  
 "ref\_amb": "/path/to/Homo\_sapiens\_assembly19.fasta.amb",  
 "ref\_ann": "/path/to/Homo\_sapiens\_assembly19.fasta.ann",  
 "ref\_bwt": "/path/to/Homo\_sapiens\_assembly19.fasta.bwt",  
 "ref\_name": "hg19"  
 },  
 "mutation\_calling.dbSNP\_vcf\_index": "/path/to/dbsnp\_138.b37.vcf.gz.tbi",  
 "mutation\_calling.dbSNP\_vcf": "/path/to/dbsnp\_138.b37.vcf.gz",  
 "mutation\_calling.known\_indels\_sites\_indices": "/path/to/Mills\_and\_1000G\_gold\_standard.indels.b37.sites.vcf.idx",  
 "mutation\_calling.known\_indels\_sites\_VCFs": "/path/to/Mills\_and\_1000G\_gold\_standard.indels.b37.sites.vcf",  
 "mutation\_calling.af\_only\_gnomad": "/path/to/af-only-gnomad.raw.sites.b37.vcf.gz",  
 "mutation\_calling.af\_only\_gnomad\_index": "/path/to/af-only-gnomad.raw.sites.b37.vcf.gz.tbi",  
 "mutation\_calling.annovar\_protocols": "refGene,knownGene,cosmic70,esp6500siv2\_all,clinvar\_20180603,gnomad211\_exome",  
 "mutation\_calling.annovar\_operation": "g,f,f,f,f,f"  
}

The JSON using the Fred Hutch HPC

{  
 "mutation\_calling.tumorSamples": ["/fh/fast/paguirigan\_a/pub/ReferenceDataSets/workflow\_testing\_data/WDL/wdl\_101/HCC4006\_final.fastq", "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/workflow\_testing\_data/WDL/wdl\_101/CALU1\_combined\_final.fastq"],  
 "mutation\_calling.refGenome": {  
 "ref\_fasta": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta",  
 "ref\_fasta\_index": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.fai",  
 "ref\_dict": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.dict",  
 "ref\_pac": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.pac",  
 "ref\_sa": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.sa",  
 "ref\_amb": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.amb",  
 "ref\_ann": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.ann",  
 "ref\_bwt": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.bwt",  
 "ref\_name": "hg19"  
 },  
 "mutation\_calling.dbSNP\_vcf\_index": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/dbsnp\_138.b37.vcf.gz.tbi",  
 "mutation\_calling.dbSNP\_vcf": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/dbsnp\_138.b37.vcf.gz",  
 "mutation\_calling.known\_indels\_sites\_indices": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Mills\_and\_1000G\_gold\_standard.indels.b37.sites.vcf.idx",  
 "mutation\_calling.known\_indels\_sites\_VCFs": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Mills\_and\_1000G\_gold\_standard.indels.b37.sites.vcf",  
 "mutation\_calling.af\_only\_gnomad": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/af-only-gnomad.raw.sites.b37.vcf.gz",  
 "mutation\_calling.af\_only\_gnomad\_index": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/af-only-gnomad.raw.sites.b37.vcf.gz.tbi",  
 "mutation\_calling.annovar\_protocols": "refGene,knownGene,cosmic70,esp6500siv2\_all,clinvar\_20180603,gnomad211\_exome",  
 "mutation\_calling.annovar\_operation": "g,f,f,f,f,f"  
}

Loading…

# 7 Task Aliasing

Right now, we have developed our workflow for only tumor samples of a patient, but our mutation caller Mutect2 performs best when each tumor is matched with a normal sample. Therefore, in order to make best use of Mutect2, we want to run a similar analysis on the normal samples as well, using tasks such as BwaMem, MarkDuplicates, and ApplyBaseRecalibrator. We could write new tasks for the normal samples, but they are run exactly the same as the tumor samples: we would like to reuse our existing tasks for these tumor samples.

WDL has a feature that allows you to reuse the same task repeatedly through your workflow: **task aliasing**. Task aliasing allows for the re-use of task definitions within the same workflow under different names, or “aliases”. For example, our task for alignment, “BwaMem”, can be aliased for both tumor and normal samples. This way, you don’t need to copy and paste the same task definition multiple times, and when you modify the the task definition, all aliases will be updated.

## 7.1 Aliasing your first task

We first demonstrate task aliasing for tumor-normal matched somatic mutation calling in which we use the same normal sample for both tumor samples. This implies that we have taken multiple tumor samples from the same patient, and we’re comparing all of them against a single normal sample. This is easy to write, but not a very common analysis. After this example, we follow up with a more sophisticated tumor-normal matched somatic mutation calling in which each patient has unique paired tumor-normal samples. This is the popular way of calling somatic mutation.

You can only alias a task that is already defined, so we will start with aliasing the BwaMem task for normal samples rather than writing a new BwaMem task specifically for normal samples. We want to do this so it can run this task on the “normal” samples and store them separately.

First, make sure that in your workflow input, you reference to the normal sample as input.

workflow mutation\_calling {  
 input {  
 ...  
 File normalFastq  
...  
 }

Then, you need to call the task you want to alias and use as to the alias\_of\_your\_choice. Within the task, you need to make sure that all the inputs reflect the new variable, so input\_fastq is directed to normalFastq.

call BwaMem as normalBwaMem {  
 input:  
 input\_fastq = normalFastq,  
 refGenome = refGenome  
 }

In the output of the workflow, you also want to make sure that in you are saving the appropriate outputs to reflect the task alias.

output {  
 File normalalignedBamSorted = normalBwaMem.analysisReadySorted  
}

## 7.2 Aliasing other tasks

We can do this for the other two tasks in our workflow as well for the normal sample:

call MarkDuplicates as normalMarkDuplicates {  
 input:  
 input\_bam = normalBwaMem.analysisReadySorted  
 }  
  
 call ApplyBaseRecalibrator as normalApplyBaseRecalibrator {  
 input:  
 input\_bam = normalMarkDuplicates.markDuplicates\_bam,  
 input\_bam\_index = normalMarkDuplicates.markDuplicates\_bai,  
 dbSNP\_vcf = dbSNP\_vcf,  
 dbSNP\_vcf\_index = dbSNP\_vcf\_index,  
 known\_indels\_sites\_VCFs = known\_indels\_sites\_VCFs,  
 known\_indels\_sites\_indices = known\_indels\_sites\_indices,  
 refGenome = refGenome  
 }

After adding these steps to the workflow, we will have our normal sample aligned and recalibrated. Together with the tumor sample, we can use the paired version of Mutect2 via the new Mutect2Paired task.

The workflow so far using the same normal sample for tumor-normal mutation calling.

version 1.0  
## WDL 101 example workflow  
##   
## This WDL workflow is intended to be used along with the WDL 101 docs.   
## This workflow should be used for inspiration purposes only.   
##  
## We use three samples   
## Samples:  
## MOLM13: Normal sample  
## CALU1: KRAS G12C mutant  
## HCC4006: EGFR Ex19 deletion mutant   
##  
## Input requirements:  
## - combined fastq files for chromosome 12 and 7 +/- 200bp around the sites of mutation only  
##  
## Output Files:  
## - An aligned bam for all 3 samples (with duplicates marked and base quality recalibrated)  
##   
## Workflow developed by Sitapriya Moorthi, Chris Lo and Taylor Firman @ Fred Hutch and Ash (Aisling) O'Farrell @ UCSC LMD: 02/28/24 for use @ Fred Hutch.  
  
struct referenceGenome {  
 File ref\_fasta  
 File ref\_fasta\_index  
 File ref\_dict  
 File ref\_amb  
 File ref\_ann  
 File ref\_bwt  
 File ref\_pac  
 File ref\_sa  
 String ref\_name  
}  
  
  
workflow mutation\_calling {  
 input {  
 Array[File] tumorSamples  
 File normalFastq  
  
 referenceGenome refGenome  
   
 # Files for specific tools  
 File dbSNP\_vcf  
 File dbSNP\_vcf\_index  
 File known\_indels\_sites\_VCFs  
 File known\_indels\_sites\_indices  
 File af\_only\_gnomad  
 File af\_only\_gnomad\_index  
  
 # Annovar options  
 String annovar\_protocols  
 String annovar\_operation  
 }  
  
 # First, process the non-tumor normal sample  
 call BwaMem as normalBwaMem {  
 input:  
 input\_fastq = normalFastq,  
 refGenome = refGenome  
 }  
   
 call MarkDuplicates as normalMarkDuplicates {  
 input:  
 input\_bam = normalBwaMem.analysisReadySorted  
 }  
  
 call ApplyBaseRecalibrator as normalApplyBaseRecalibrator {  
 input:  
 input\_bam = normalMarkDuplicates.markDuplicates\_bam,  
 input\_bam\_index = normalMarkDuplicates.markDuplicates\_bai,  
 dbSNP\_vcf = dbSNP\_vcf,  
 dbSNP\_vcf\_index = dbSNP\_vcf\_index,  
 known\_indels\_sites\_VCFs = known\_indels\_sites\_VCFs,  
 known\_indels\_sites\_indices = known\_indels\_sites\_indices,  
 refGenome = refGenome  
 }  
   
 # Scatter for "tumor" samples   
 scatter (tumorSample in tumorSamples) {  
 call BwaMem as tumorBwaMem {  
 input:  
 input\_fastq = tumorSample,  
 refGenome = refGenome  
 }  
   
 call MarkDuplicates as tumorMarkDuplicates {  
 input:  
 input\_bam = tumorBwaMem.analysisReadySorted  
 }  
  
 call ApplyBaseRecalibrator as tumorApplyBaseRecalibrator{  
 input:  
 input\_bam = tumorMarkDuplicates.markDuplicates\_bam,  
 input\_bam\_index = tumorMarkDuplicates.markDuplicates\_bai,  
 dbSNP\_vcf = dbSNP\_vcf,  
 dbSNP\_vcf\_index = dbSNP\_vcf\_index,  
 known\_indels\_sites\_VCFs = known\_indels\_sites\_VCFs,  
 known\_indels\_sites\_indices = known\_indels\_sites\_indices,  
 refGenome = refGenome  
 }  
  
 call Mutect2Paired {  
 input:  
 tumor\_bam = tumorApplyBaseRecalibrator.recalibrated\_bam,  
 tumor\_bam\_index = tumorApplyBaseRecalibrator.recalibrated\_bai,  
 normal\_bam = normalApplyBaseRecalibrator.recalibrated\_bam,  
 normal\_bam\_index = normalApplyBaseRecalibrator.recalibrated\_bai,  
 refGenome = refGenome,  
 genomeReference = af\_only\_gnomad,  
 genomeReferenceIndex = af\_only\_gnomad\_index  
 }  
  
 call annovar {  
 input:  
 input\_vcf = Mutect2Paired.output\_vcf,  
 ref\_name = refGenome.ref\_name,  
 annovar\_operation = annovar\_operation,  
 annovar\_protocols = annovar\_protocols  
 }  
}  
  
 output {  
 Array[File] tumoralignedBamSorted = tumorBwaMem.analysisReadySorted  
 Array[File] tumorMarkDuplicates\_bam = tumorMarkDuplicates.markDuplicates\_bam  
 Array[File] tumorMarkDuplicates\_bai = tumorMarkDuplicates.markDuplicates\_bai  
 Array[File] tumoranalysisReadyBam = tumorApplyBaseRecalibrator.recalibrated\_bam   
 Array[File] tumoranalysisReadyIndex = tumorApplyBaseRecalibrator.recalibrated\_bai  
 File normalalignedBamSorted = normalBwaMem.analysisReadySorted  
 File normalmarkDuplicates\_bam = normalMarkDuplicates.markDuplicates\_bam  
 File normalmarkDuplicates\_bai = normalMarkDuplicates.markDuplicates\_bai  
 File normalanalysisReadyBam = normalApplyBaseRecalibrator.recalibrated\_bam   
 File normalanalysisReadyIndex = normalApplyBaseRecalibrator.recalibrated\_bai  
 Array[File] Mutect2Paired\_Vcf = Mutect2Paired.output\_vcf  
 Array[File] Mutect2Paired\_VcfIndex = Mutect2Paired.output\_vcf\_index  
 Array[File] Mutect2Paired\_AnnotatedVcf = annovar.output\_annotated\_vcf  
 Array[File] Mutect2Paired\_AnnotatedTable = annovar.output\_annotated\_table  
 }  
  
 parameter\_meta {  
 tumorSamples: "Tumor .fastq, one sample per .fastq file (expects Illumina)"  
 normalFastq: "Non-tumor .fastq (expects Illumina)"  
  
 dbSNP\_vcf: "dbSNP VCF for mutation calling"  
 dbSNP\_vcf\_index: "dbSNP VCF index"  
 known\_indels\_sites\_VCFs: "Known indel site VCF for mutation calling"  
 known\_indels\_sites\_indices: "Known indel site VCF indicies"  
 af\_only\_gnomad: "gnomAD population allele fraction for mutation calling"  
 af\_only\_gnomad\_index: "gnomAD population allele fraction index"  
  
 annovar\_protocols: "annovar protocols: see https://annovar.openbioinformatics.org/en/latest/user-guide/startup"  
 annovar\_operation: "annovar operation: see https://annovar.openbioinformatics.org/en/latest/user-guide/startup"  
 }  
}  
  
####################  
# Task definitions #  
####################  
  
# Align fastq file to the reference genome  
task BwaMem {  
 input {  
 File input\_fastq  
 referenceGenome refGenome  
 }  
   
 String base\_file\_name = basename(input\_fastq, ".fastq")  
 String ref\_fasta\_local = basename(refGenome.ref\_fasta)  
  
 String read\_group\_id = "ID:" + base\_file\_name  
 String sample\_name = "SM:" + base\_file\_name  
 String platform\_info = "PL:illumina"  
  
  
 command <<<  
 set -eo pipefail  
  
 mv "~{refGenome.ref\_fasta}" .  
 mv "~{refGenome.ref\_fasta\_index}" .  
 mv "~{refGenome.ref\_dict}" .  
 mv "~{refGenome.ref\_amb}" .  
 mv "~{refGenome.ref\_ann}" .  
 mv "~{refGenome.ref\_bwt}" .  
 mv "~{refGenome.ref\_pac}" .  
 mv "~{refGenome.ref\_sa}" .  
  
 bwa mem \  
 -p -v 3 -t 16 -M -R '@RG\t~{read\_group\_id}\t~{sample\_name}\t~{platform\_info}' \  
 "~{ref\_fasta\_local}" "~{input\_fastq}" > "~{base\_file\_name}.sam"   
 samtools view -1bS -@ 15 -o "~{base\_file\_name}.aligned.bam" "~{base\_file\_name}.sam"  
 samtools sort -@ 15 -o "~{base\_file\_name}.sorted\_query\_aligned.bam" "~{base\_file\_name}.aligned.bam"  
 >>>  
  
 output {  
 File analysisReadySorted = "~{base\_file\_name}.sorted\_query\_aligned.bam"  
 }  
   
 runtime {  
 memory: "48 GB"  
 cpu: 16  
 docker: "ghcr.io/getwilds/bwa:0.7.17"  
 }  
}  
  
# Mark duplicates on a BAM file  
task MarkDuplicates {  
 input {  
 File input\_bam  
 }  
  
 String base\_file\_name = basename(input\_bam, ".sorted\_query\_aligned.bam")  
  
 command <<<  
 gatk MarkDuplicates \  
 --INPUT "~{input\_bam}" \  
 --OUTPUT "~{base\_file\_name}.duplicates\_marked.bam" \  
 --METRICS\_FILE "~{base\_file\_name}.duplicate\_metrics" \  
 --CREATE\_INDEX true \  
 --OPTICAL\_DUPLICATE\_PIXEL\_DISTANCE 100 \  
 --VALIDATION\_STRINGENCY SILENT  
 >>>  
  
 runtime {  
 docker: "ghcr.io/getwilds/gatk:4.3.0.0"  
 memory: "48 GB"  
 cpu: 4  
 }  
  
 output {  
 File markDuplicates\_bam = "~{base\_file\_name}.duplicates\_marked.bam"  
 File markDuplicates\_bai = "~{base\_file\_name}.duplicates\_marked.bai"  
 File duplicate\_metrics = "~{base\_file\_name}.duplicates\_marked.bai"  
 }  
}  
  
# Base quality recalibration  
task ApplyBaseRecalibrator {  
 input {  
 File input\_bam  
 File input\_bam\_index  
 File dbSNP\_vcf  
 File dbSNP\_vcf\_index  
 File known\_indels\_sites\_VCFs  
 File known\_indels\_sites\_indices  
 referenceGenome refGenome  
 }  
   
 String base\_file\_name = basename(input\_bam, ".duplicates\_marked.bam")  
   
 String ref\_fasta\_local = basename(refGenome.ref\_fasta)  
 String dbSNP\_vcf\_local = basename(dbSNP\_vcf)  
 String known\_indels\_sites\_VCFs\_local = basename(known\_indels\_sites\_VCFs)  
  
  
 command <<<  
 set -eo pipefail  
  
 mv "~{refGenome.ref\_fasta}" .  
 mv "~{refGenome.ref\_fasta\_index}" .  
 mv "~{refGenome.ref\_dict}" .  
  
 mv "~{dbSNP\_vcf}" .  
 mv "~{dbSNP\_vcf\_index}" .  
  
 mv "~{known\_indels\_sites\_VCFs}" .  
 mv "~{known\_indels\_sites\_indices}" .  
  
 samtools index "~{input\_bam}"  
  
 gatk --java-options "-Xms8g" \  
 BaseRecalibrator \  
 -R "~{ref\_fasta\_local}" \  
 -I "~{input\_bam}" \  
 -O "~{base\_file\_name}.recal\_data.csv" \  
 --known-sites "~{dbSNP\_vcf\_local}" \  
 --known-sites "~{known\_indels\_sites\_VCFs\_local}" \  
   
  
 gatk --java-options "-Xms8g" \  
 ApplyBQSR \  
 -bqsr "~{base\_file\_name}.recal\_data.csv" \  
 -I "~{input\_bam}" \  
 -O "~{base\_file\_name}.recal.bam" \  
 -R "~{ref\_fasta\_local}" \  
   
  
 # finds the current sort order of this bam file  
 samtools view -H "~{base\_file\_name}.recal.bam" | grep @SQ | sed 's/@SQ\tSN:\|LN://g' > "~{base\_file\_name}.sortOrder.txt"  
 >>>  
  
 output {  
 File recalibrated\_bam = "~{base\_file\_name}.recal.bam"  
 File recalibrated\_bai = "~{base\_file\_name}.recal.bai"  
 File sortOrder = "~{base\_file\_name}.sortOrder.txt"  
 }  
 runtime {  
 memory: "36 GB"  
 cpu: 2  
 docker: "ghcr.io/getwilds/gatk:4.3.0.0"  
 }  
}  
  
# Variant calling via mutect2 (tumor-and-normal mode)  
task Mutect2Paired {  
 input {  
 File tumor\_bam  
 File tumor\_bam\_index  
 File normal\_bam  
 File normal\_bam\_index  
 referenceGenome refGenome  
 File genomeReference  
 File genomeReferenceIndex  
 }  
  
 String base\_file\_name\_tumor = basename(tumor\_bam, ".recal.bam")  
 String ref\_fasta\_local = basename(refGenome.ref\_fasta)  
 String genomeReference\_local = basename(genomeReference)  
  
 command <<<  
 set -eo pipefail  
  
 mv "~{refGenome.ref\_fasta}" .  
 mv "~{refGenome.ref\_fasta\_index}" .  
 mv "~{refGenome.ref\_dict}" .  
  
 mv "~{genomeReference}" .  
 mv "~{genomeReferenceIndex}" .  
  
 gatk --java-options "-Xms16g" Mutect2 \  
 -R "~{ref\_fasta\_local}" \  
 -I "~{tumor\_bam}" \  
 -I "~{normal\_bam}" \  
 -O preliminary.vcf.gz \  
 --germline-resource "~{genomeReference\_local}" \  
  
 gatk --java-options "-Xms16g" FilterMutectCalls \  
 -V preliminary.vcf.gz \  
 -O "~{base\_file\_name\_tumor}.mutect2.vcf.gz" \  
 -R "~{ref\_fasta\_local}" \  
 --stats preliminary.vcf.gz.stats \  
 >>>  
  
 runtime {  
 docker: "ghcr.io/getwilds/gatk:4.3.0.0"  
 memory: "24 GB"  
 cpu: 1  
 }  
  
 output {  
 File output\_vcf = "${base\_file\_name\_tumor}.mutect2.vcf.gz"  
 File output\_vcf\_index = "${base\_file\_name\_tumor}.mutect2.vcf.gz.tbi"  
 }  
}  
  
# Annotate VCF using annovar  
task annovar {  
 input {  
 File input\_vcf  
 String ref\_name  
 String annovar\_protocols  
 String annovar\_operation  
 }  
 String base\_vcf\_name = basename(input\_vcf, ".vcf.gz")  
   
 command <<<  
 set -eo pipefail  
   
 perl /annovar/table\_annovar.pl "~{input\_vcf}" /annovar/humandb/ \  
 -buildver "~{ref\_name}" \  
 -outfile "~{base\_vcf\_name}" \  
 -remove \  
 -protocol "~{annovar\_protocols}" \  
 -operation "~{annovar\_operation}" \  
 -nastring . -vcfinput  
 >>>  
 runtime {  
 docker : "ghcr.io/getwilds/annovar:${ref\_name}"  
 cpu: 1  
 memory: "2GB"  
 }  
 output {  
 File output\_annotated\_vcf = "~{base\_vcf\_name}.${ref\_name}\_multianno.vcf"  
 File output\_annotated\_table = "~{base\_vcf\_name}.${ref\_name}\_multianno.txt"  
 }  
}

## 7.3 Paired tumor normal calling

Now that we are comfortable with task aliasing, we follow up with a more sophisticated tumor-normal matched somatic mutation calling in which each patient has unique paired tumor-normal samples. In order to do so, we need to ensure that each patient has an unique tumor and normal sample. We could modify our workflow inputs to be:

workflow mutation\_calling {  
 input {  
 Array[File] tumorSamples  
 Array[file] normalSamples  
 ...  
 }  
}

but that would be a bit awkward to scatter through, as we would need to create a separate index to keep track of which sample we are using in the array of tumorSamples and normalSamples.

Instead, we write a struct for our paired samples:

struct pairedSample {  
 File tumorSample  
 File normalSample  
}

so that in our workflow inputs, we use an array of pairedSample:

workflow mutation\_calling {  
 input {  
 Array[pairedSample] samples  
 ...  
 }  
}

and we can scatter on samples. Within our scatter, we use task aliasing to call tumor-specific and normal-specific tasks until Mutect2Paired and annovar.

scatter (sample in samples) {  
  
 #Tumors  
 call BwaMem as tumorBwaMem {  
 input:  
 input\_fastq = sample.tumorSample,  
 refGenome = refGenome  
 }  
  
 #Normals  
 call BwaMem as normalBwaMem {  
 input:  
 input\_fastq = sample.normalSample,  
 refGenome = refGenome  
 }  
   
 ...  
   
}

Putting it together:

The JSON metadata:

{  
 "mutation\_calling.samples": [{"tumorSample": "/path/to/Tumor\_1\_KRAS\_CALU1\_combined\_final.fastq",  
 "normalSample": "/path/to/Normal\_1\_MOLM13\_combined\_final.fastq"},  
 {"tumorSample": "/path/to/Tumor\_2\_EGFR\_HCC4006\_combined.fastq",  
 "normalSample": "/path/to/Normal\_2\_MOLM13\_combined\_final.fastq"}],  
 "mutation\_calling.refGenome": {  
 "ref\_fasta": "/path/to/Homo\_sapiens\_assembly19.fasta",  
 "ref\_fasta\_index": "/path/to/Homo\_sapiens\_assembly19.fasta.fai",  
 "ref\_dict": "/path/to/Homo\_sapiens\_assembly19.dict",  
 "ref\_pac": "/path/to/Homo\_sapiens\_assembly19.fasta.pac",  
 "ref\_sa": "/path/to/Homo\_sapiens\_assembly19.fasta.sa",  
 "ref\_amb": "/path/to/Homo\_sapiens\_assembly19.fasta.amb",  
 "ref\_ann": "/path/to/Homo\_sapiens\_assembly19.fasta.ann",  
 "ref\_bwt": "/path/to/Homo\_sapiens\_assembly19.fasta.bwt",  
 "ref\_name": "hg19"  
 },  
 "mutation\_calling.dbSNP\_vcf\_index": "/path/to/dbsnp\_138.b37.vcf.gz.tbi",  
 "mutation\_calling.dbSNP\_vcf": "/path/to/dbsnp\_138.b37.vcf.gz",  
 "mutation\_calling.known\_indels\_sites\_indices": "/path/to/Mills\_and\_1000G\_gold\_standard.indels.b37.sites.vcf.idx",  
 "mutation\_calling.known\_indels\_sites\_VCFs": "/path/to/Mills\_and\_1000G\_gold\_standard.indels.b37.sites.vcf",  
 "mutation\_calling.af\_only\_gnomad": "/path/to/af-only-gnomad.raw.sites.b37.vcf.gz",  
 "mutation\_calling.af\_only\_gnomad\_index": "/path/to/af-only-gnomad.raw.sites.b37.vcf.gz.tbi",  
 "mutation\_calling.annovar\_protocols": "refGene,knownGene,cosmic70,esp6500siv2\_all,clinvar\_20180603,gnomad211\_exome",  
 "mutation\_calling.annovar\_operation": "g,f,f,f,f,f"  
}

The JSON using the Fred Hutch HPC

{  
 "mutation\_calling.sampleFastq": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/workflow\_testing\_data/WDL/wdl\_101/HCC4006\_final.fastq",  
 "mutation\_calling.ref\_fasta": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta",  
 "mutation\_calling.ref\_fasta\_index": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.fai",  
 "mutation\_calling.ref\_dict": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.dict",  
 "mutation\_calling.ref\_pac": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.pac",  
 "mutation\_calling.ref\_sa": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.sa",  
 "mutation\_calling.ref\_amb": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.amb",  
 "mutation\_calling.ref\_ann": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.ann",  
 "mutation\_calling.ref\_bwt": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Homo\_sapiens\_assembly19.fasta.bwt",  
 "mutation\_calling.ref\_name": "hg19",  
 "mutation\_calling.dbSNP\_vcf\_index": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/dbsnp\_138.b37.vcf.gz.tbi",  
 "mutation\_calling.dbSNP\_vcf": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/dbsnp\_138.b37.vcf.gz",  
 "mutation\_calling.known\_indels\_sites\_indices": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Mills\_and\_1000G\_gold\_standard.indels.b37.sites.vcf.idx",  
 "mutation\_calling.known\_indels\_sites\_VCFs": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/Mills\_and\_1000G\_gold\_standard.indels.b37.sites.vcf",  
 "mutation\_calling.af\_only\_gnomad": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/af-only-gnomad.raw.sites.b37.vcf.gz",  
 "mutation\_calling.af\_only\_gnomad\_index": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/af-only-gnomad.raw.sites.b37.vcf.gz.tbi",  
 "mutation\_calling.annovar\_protocols": "refGene,knownGene,cosmic70,esp6500siv2\_all,clinvar\_20180603,gnomad211\_exome",  
 "mutation\_calling.annovar\_operation": "g,f,f,f,f,f"  
}

Now we have a complete workflow that reflects our original plan in [Defining a workflow plan](https://hutchdatascience.org/WDL_Workflows_Guide/defining-a-workflow-plan.html)!

Loading…

# 8 Optional types

WDL supports defining optional variables, which are denoted with a question mark next to the type declaration. These variables may or may not defined. Optional types are powerful, but they can quickly cause problems in complicated workflows. This is especially the case when working with scattered tasks. In this section, we’ll go over how optional variables can be used, and where they can cause problems.

## 8.1 Optional inputs

The most common use case for optional variables are optional inputs for the user to provide some sort of file, for example, a reference genome or a metadata file. Sometimes, you will want optional inputs to fall back on a known file or value.

A good place to use optional types in our workflow would be in the bwa mem task. bwa mem on default settings may attempt to use too many threads, which could cause this task to fail if it ran on weaker hardware such as a laptop. In order to allow the user to adjust how many threads bwa mem uses, and by extension make our WDL run on a wider variety of machines, we can edit the task to add a new variable. This new variable will replace -t 16 in our call to bwa mem.

task BwaMem {  
 input {  
 File input\_fastq  
 referenceGenome refGenome  
  
 # Number of threads to use - must be defined  
 Int threads  
 }  
 command <<<  
 set -eo pipefail  
  
 mv "~{refGenome.ref\_fasta}" .  
 mv "~{refGenome.ref\_fasta\_index}" .  
 mv "~{refGenome.ref\_dict}" .  
 mv "~{refGenome.ref\_amb}" .  
 mv "~{refGenome.ref\_ann}" .  
 mv "~{refGenome.ref\_bwt}" .  
 mv "~{refGenome.ref\_pac}" .  
 mv "~{refGenome.ref\_sa}" .  
  
 bwa mem \  
 -p -v 3 -t ~{threads} -M -R '@RG\t~{read\_group\_id}\t~{sample\_name}\t~{platform\_info}' \  
 "~{ref\_fasta\_local}" "~{input\_fastq}" > "~{base\_file\_name}.sam"   
 samtools view -1bS -@ 15 -o "~{base\_file\_name}.aligned.bam" "~{base\_file\_name}.sam"  
 samtools sort -@ 15 -o "~{base\_file\_name}.sorted\_query\_aligned.bam" "~{base\_file\_name}.aligned.bam"  
[...]  
}

Because ~{threads} is an integer variable, it cannot have a space in it, so we don’t need to surround ~{threads} with quotation marks like we should String or File variables.

Now we can set Int? bwa\_mem\_threads as a workflow-level optional input. Making it optional means that users who do not want to think about threads – perhaps they’re using an HPC or other powerful backend – do not have to worry about filling in that value. In the workflow body, when we call bwa mem, the value of bwa\_mem\_threads is passed to the BwaMem task.

workflow mutation\_calling {  
 input {  
 Array[File] tumorSamples  
 File normalFastq  
  
 referenceGenome refGenome  
  
 # Optional variable for bwa mem  
 Int? bwa\_mem\_threads  
   
 # Files for specific tools  
 File dbSNP\_vcf  
 File dbSNP\_vcf\_index  
 File known\_indels\_sites\_VCFs  
 File known\_indels\_sites\_indices  
 File af\_only\_gnomad  
 File af\_only\_gnomad\_index  
  
 # Annovar options  
 String annovar\_protocols  
 String annovar\_operation  
 }  
 call BwaMem as normalBwaMem {  
 input:  
 input\_fastq = normalFastq,  
 refGenome = refGenome,  
 threads = bwa\_mem\_threads  
 }  
[...]  
}

But what happens if the workflow-level variable bwa\_mem\_threads isn’t defined? The task BwaMem will then get no value for the task-level variable threads. This will cause an error at runtime. We can’t solve this merely by making the task level variable optional too – otherwise, if threads isn’t defined, our bwa mem call will provide -t without a number after it, which is invalid.

One way to address this is with the WDL built-in function select\_first(), which takes in an array of values. In the workflow body, we can use this plus a fallback value (or another variable) to ensure that the BwaMem task always gets a valid integer for the threads argument.

call BwaMem as normalBwaMem {  
 input:  
 input\_fastq = normalFastq,  
 refGenome = refGenome,  
 threads = select\_first([bwa\_mem\_threads, 16])  
 }

This is a perfectly valid way of doing things, but the task itself still relies on that variable being defined. In the context of our workflow, that doesn’t matter. However, keep in mind it is a common practice to reuse code for multiple scripts – whether that is directly importing other WDL tasks using WDL’s importing feature, or simply copy-pasting boilerplate code. With that in mind, it’s a good idea to make tasks as “stand-alone” as possible, and not reliant on the workflow calling them to use select\_first().

As such, it’s more common to make the task itself handle this by providing a fallback value, like so:

task BwaMem {  
 input {  
 File input\_fastq  
 referenceGenome refGenome  
 Int threads = 16 # if a workflow passes an optional variable with no value, fall back to 16  
 }  
 [...]  
}

Here’s another way of providing a fallback value for optional inputs within a task.

In the following task, the phylogenetic tree program UShER expects a reference genome, an input phylogenetic tree, samples in the form of diff files, and an output file name. However, the Docker image we’re using (ashedpotatoes/usher-plus:0.0.2) already contains the H37Rv reference genome, which is the standard reference genome for *Mycobacterium tuberculosis*. If we are working with tuberculosis samples, there is no reason to provide a reference genome – we might as well use the one that’s already in the Docker image.

task usher\_sampled\_diff {  
 input {  
 File diff  
 File input\_tree # in MAT format, extension should be .pb  
 String output\_file\_name  
 File? ref\_genome  
 }  
  
 command <<<  
 if [[ "~{ref\_genome}" = "" ]]  
 then  
 # if not defined, fall back to H37Rv in the Docker image  
 ref="/HOME/usher/ref/Ref.H37Rv/ref.fa"  
 else  
 ref="~{ref\_genome}"  
 fi  
  
 usher-sampled --diff "~{diff}" \  
 -i "${input\_tree}" \  
 --ref "$ref" \  
 -o "~{output\_file\_name}"  
 >>>  
  
 runtime {  
 docker: "ashedpotatoes/usher-plus:0.0.2"  
 }  
  
 output {  
 File usher\_tree = output\_file\_name  
 }  
}

## 8.2 Optional outputs

Optional outputs are usually created by explicitly declaring a task or workflow’s output variable to be optional. They can also be created by enclosing a task within an if-block, which will cause all outputs of that task to be considered optional.

### 8.2.1 Declaring a task’s output to be optional

Most backends will save the task-level outputs of a task, even if those outputs are not workflow-level outputs. This can be very useful if you want to be able to check intermediate files, or perhaps log files for certain bioinformatic tools. However, if you’re not actively debugging, those files may take up a lot of space on your backend if you’re running on many samples over time.

You could change the workflow every time you wish to keep those intermediate inputs, but it would be easier to make the output optional.

task usher\_sampled\_diff {  
 input {  
 File diff  
 File input\_tree # in MAT format, extension should be .pb  
 String output\_file\_name  
 File? ref\_genome  
 Boolean detailed\_clades  
 }  
 # UShER will output clade information if you pass the -D flag to it, but  
 # we don't want users to have to type in flags as a String variable. Instead,  
 # we just make a Boolean variable called detailed\_clades, and outside the inputs  
 # section, we write the string "-D" if detailed\_clades is true, or an empty string  
 # if detailed\_clades is false.  
 String detailed\_clades\_flag = if !(detailed\_clades) then "" else "-D "  
  
 command <<<  
 if [[ "~{ref\_genome}" = "" ]]  
 then  
 ref="/HOME/usher/ref/Ref.H37Rv/ref.fa"  
 else  
 ref="~{ref\_genome}"  
 fi  
  
 usher-sampled ~{detailed\_clades\_flag} \   
 --diff "~{diff}" \  
 -i "${input\_tree}" \  
 --ref "$ref" \  
 -o "~{output\_file\_name}"  
 >>>  
  
 runtime {  
 docker: "ashedpotatoes/usher-plus:0.0.2"  
 }  
  
 output {  
 File usher\_tree = output\_file\_name  
 File? clades = "clades.txt" # only if detailed\_clades = true  
 }  
}

If we want, we can make this task-level output a workflow-level output – perhaps we use a backend that does not save task-level outputs, or it is simply easier to access workflow-level outputs on this particular backend. Provided that the task is not scattered, this is very easy to do. (We’ll talk about scattered tasks later, as they complicate matters.) In the workflow output sections, we simply define the output as optional like we did in the task itself.

### 8.2.2 Making an entire task optional

Let’s say we want to make an entire task optional. Perhaps it provides additional QC information, or isn’t relevant to certain data types.

Although WDL has a concept of if, it does not have any concept of else, so to make mutually exclusive tasks, you essentially need use two mutually exclusive if-statements. In the example below, we use two mutually exclusive if-statements to decide whether to run task\_x or task\_y. This is a valid way of doing things, and will work as expected:

workflow do\_x\_or\_y {  
 input {  
 Boolean do\_x = true # if true do task\_x, if false do task\_y  
 File some\_input  
 }  
 if (do\_x = true) {  
 call task\_x {  
 input:  
 some\_input = some\_input  
 }  
 }  
 if (do\_x = false) {  
 call task\_y {  
 input:  
 some\_input = some\_input  
 }  
 }  
}

However, we need to keep in mind that WDL is not “aware” that these two tasks are mutually exclusive. You and I both know that if task\_x has run, then task\_y must not have run, but your WDL executor does not know this. So, as a consequence, you cannot use mutually exclusive if-blocks to set a variable to a particular value.

# this workflow will fail miniwdl check or womtool  
workflow do\_x\_or\_y {  
 input {  
 Boolean do\_x = true # if true do task\_x, if false do task\_y  
 File some\_input  
 }  
 if (do\_x = true) {  
 Int some\_variable = 10 # this is problematic, even though only one of these if blocks will execute!  
 call task\_x {  
 input:  
 some\_input = some\_input  
 }  
 }  
 if (do\_x = false) {  
 Int some\_variable = 5 # this is problematic, even though only one of these if blocks will execute!  
 call task\_y {  
 input:  
 some\_input = some\_input  
 }  
 }  
}

Another consequence is that both of your mutually exclusive tasks’ outputs will be considered optional types, even though you can be certain that set of outputs absolutely do exist and one set of outputs absolutely do not exist.

workflow do\_x\_or\_y {  
 input {  
 Boolean do\_x = true # if true do task\_x, if false do task\_y  
 File some\_input  
 }  
 if (do\_x = true) {  
 call task\_x {  
 input:  
 some\_input = some\_input  
 }  
 }  
 if (do\_x = false) {  
 call task\_y {  
 input:  
 some\_input = some\_input  
 }  
 }  
  
 outputs {  
 # these both MUST be optional  
 File? task\_x\_output = task\_x.something  
 File? task\_y\_output = task\_y.something  
 }  
}

Checking if a variable is defined

You can check if a variable is defined at runtime using the WDL built-in defined(), which returns a Boolean value. However, defined() can give unexpected output, as we will go over later. For the time being, let’s use a simple example where we set a Boolean variable via defined(). This will work as expected – if task\_x ran and created the output task\_x.something, then did\_x\_run will be true, and so on.

workflow do\_x\_or\_y {  
 input {  
 Boolean do\_x = true # if true do task\_x, if false do task\_y  
 File some\_input  
 }  
 if (do\_x = true) {  
 call task\_x {  
 input:  
 some\_input = some\_input  
 }  
 }  
 if (do\_x = false) {  
 call task\_y {  
 input:  
 some\_input = some\_input  
 }  
 }  
  
 # these work as you would expect  
 Boolean did\_x\_run = defined(task\_x.something)  
 Boolean did\_y\_run = defined(task\_y.something)  
  
 outputs {  
 File? task\_x\_output = task\_x.something  
 File? task\_y\_output = task\_y.something  
 Boolean did\_x\_run = did\_x\_run  
 Boolean did\_y\_run = did\_y\_run  
 }  
}

Note that although you could decide whether or not to execute a task by wrapping that task in a defined() if-block, you cannot use defined() to coerce a variable. It’s generally best to use select\_first() for type coercion, although we will mention some exceptions when it comes to arrays.

# if some\_integer has type Int?, we cannot coerce it like this:  
 if defined(some\_integer) {  
 Int real\_some\_integer = some\_integer  
 }  
 # because outside of the if-block, real\_some\_integer is considered optional

You should not use defined() with optional arrays, as it often acts in unexpected ways. It is very easy to accidentally create a defined() check that always returns true, as WDL executors may define “optional” arrays even if they have no values. For example, let’s consider task\_x and task\_y again. To recap, these are mutually exclusive tasks which each output a single File. If these tasks are scattered from outside their respective if-blocks, the their outputs are no longer File?, instead, they are arrays. But are they arrays of optional File?s, or are the arrays themselves optional? Let’s try to find out:

workflow do\_x\_or\_y {  
 input {  
 Boolean do\_x = true # if true do task\_x, if false do task\_y  
 Array[File] some\_inputs  
 }  
 scatter(some\_input in some\_inputs) {  
 if (do\_x = true) {  
 call task\_x {  
 input:  
 some\_input = some\_input  
 }  
 }  
 if (do\_x = false) {  
 call task\_y {  
 input:  
 some\_input = some\_input  
 }  
 }  
 }  
  
 # these both always return true!  
 Boolean did\_x\_run = defined(task\_x.something)  
 Boolean did\_y\_run = defined(task\_y.something)  
  
 outputs {  
 File? task\_x\_output = task\_x.something  
 File? task\_y\_output = task\_y.something  
 Boolean did\_x\_run = did\_x\_run  
 Boolean did\_y\_run = did\_y\_run  
 }  
}

Instead of using defined() on an array, consider simply checking its length. An empty array has a length of 0, so if the length of an array is greater than 1, you know that there is at least one item existing in that array.

Working with optional arrays and advanced usage of select\_first()

Coercing arrays with optional components

Optional arrays can be difficult to work with. One of the reasons for this is that they sometimes give unexpected values for defined(). As mentioned earlier, this can be somewhat alleviated by checking the length of an array rather than whether or not it is defined, but the problems do not end there: Attempting to scatter on an optional array can cause issues that are difficult to debug.

It’s generally best to coerce optional arrays into not-optional arrays as much as possible. Thankfully, we can use select\_all() to select only the defined elements of an array. For our example above, we can go even further by creating an array that will always have a defined value, whether or not do\_x is true. We combine the power of select\_all() and flatten() to create the following:

Array[File] x\_or\_y\_outfiles = flatten(select\_all(x.outfile), select\_all(y.outfile))

Make sure not to use flatten(select\_all(x.outfile, y.outfile))! That would result in an Array[File] that has a null value, which can cause hard-to-diagnose issues later on in your pipeline.

If you’re absolutely certain your variable, in this particular scope, is defined, you can make the second member of the select\_first() array a random bogus fallback value. You may want to add a comment to prevent you or other programmers accidentally breaking things should they edit the code later.

if defined(some\_integer) { # needed to define real\_some\_integer, beware of changing this!  
 Int real\_some\_integer = select\_first([some\_integer, 1975])  
}

## 8.3 The final workflow

At this point, we’ve completed our workflow. Here’s what it looks like now:

Loading…

# 9 Optimization

In our last chapter, we will explore ways to optimize our completed workflow.

Generally, in computing, we are working with the following finite resources:

* **Computing time**: a task can take milliseconds to days to complete. We find ways to reduce our computing time so that we have results sooner.
* **Central Processing Unit (CPU) usage**: a task requires at least one CPU to be the “brain” of the computer to run computation, move data around, and access the memory. Tasks can make use of multiple CPUs to process in parallel.
* **Memory usage (RAM)**: a task needs a place to store data while computing on the job. Reducing the memory usage can allow the task to be more efficient.
* **Disk usage**: A task usually requires some input data and have an output, and needs disk storage to store the information when the task isn’t running.

Changing usage of any of these resources often affect other resources. For instance, there’s often a tradeoff between computing time and memory usage. In order to be efficient with these finite resources, we will introduce some optimization methods that are common in building an efficient workflow, and provide an example on a simple optimization method, “Embarrassingly Parallel Scatter-Gather”.

## 9.1 Common optimizing/parallelizing methods

### 9.1.1 Memory optimization

To optimize the amount of memory used for a task, you can profile the task to understand how much memory is being used and where the bottlenecks are. Most programming languages have a large selection of memory profilers to analyze the performance of your code, and they can be starting points to consider how to use less memory. When a bottleneck is found, you might consider to use a more efficient algorithm to reduce memory usage. Or, perhaps you might use more optimal data structures, such as databases or sparse data structures, that better fit the data type of your problem.

Another common memory analysis is to understand how memory usage scales relative to the input for the task. For example, as you increase the VCF file size for a task, how much does the memory usage scale? Is there something you can do so that it scales with a smaller magnitude?

A technique you may want to consider if your memory usage scales with your input data size is that you could break down the input into smaller parts, and run your task on these smaller parts, each requesting smaller amount of memory. This wouldn’t reduce the total amount of memory you use, but many computing backends prefer many tasks using small amounts of a memory instead of a single job requiring a large amount of memory. This would also help optimizing computing time. Below, we look at different ways of parallelization.

### 9.1.2 Embarrassingly Parallel Scatter-Gather

A sub-task is called “Embarrassingly Parallel” if the sub-task can be run completely independent of other sub-tasks to finish the task. When we run multiple samples in our workflow, it is a form of “Embarrassingly Parallel”, because processing each tumor-normal pair through the workflow is done independently of other samples on separate computers (also known in high performance computing as “nodes”). This technique reduces the computing time and breaks down CPU and memory usage into smaller, more affordable resource requests. Later in this chapter we will show an detailed example of using this technique to split a BAM into individual chromosomes to run a resource demanding task.

### 9.1.3 Multithreading (Shared-Memory Parallelism)

Sometimes, you want to use multiple CPUs on a single computer to run sub-components of your task in parallel. This is called “multithreading”. Here, a single task accesses multiple CPUs on a computer to split up the task, and these CPUs share the same memory. Due to shared memory, each of these CPUs are not quite independent, and some communication is needed between CPUs to make this efficient. Well-documented and optimized bioinformatics software will often have usage options to use multiple cores to speed up the task. In our previous chapters, we used BWA MEM with an optional default of 16 CPUs. If you are developing your own task, then there are many built in operations that can make use of multiple CPUs, as well as low-level threading methods to help you develop a multi-threaded program.

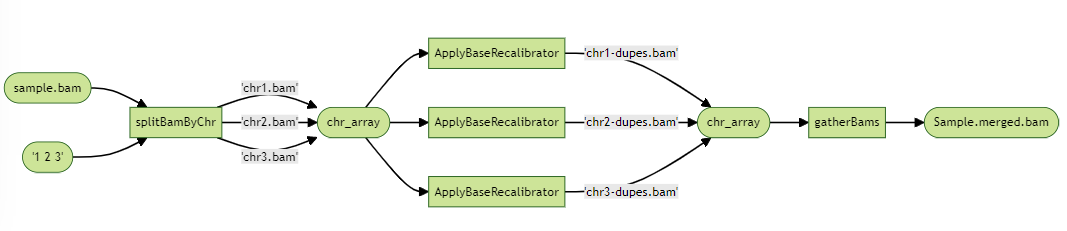
### 9.1.4 Multiprocessing (Distributed-Memory Parallelism)

Sometimes, the scale of the work requires coordination of multiple computers working dependently of each other. This is called “multiprocessing”. This is common in working with large scale data, using tools such as Spark or Hadoop. We will not be talking about multiprocessing extensively in this guide.

### 9.1.5 Graphical Processing Units (GPUs)

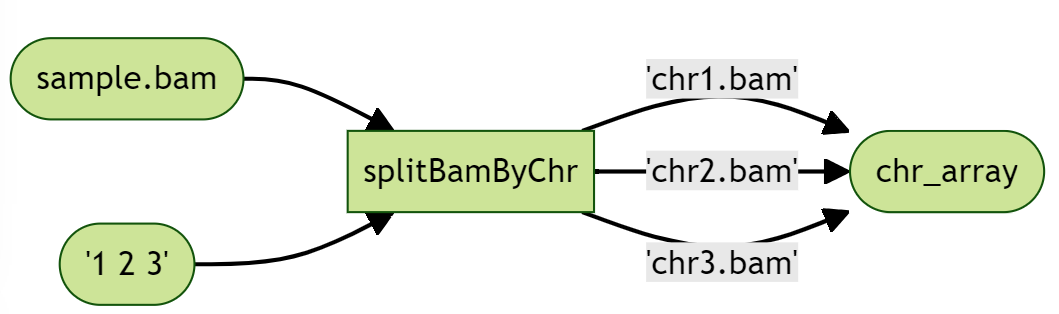
Graphical processing units are an additional processing tool to CPUs, in which a different computer hardware is used. There are usually a few handful of CPUs on a computer, but there can be thousands of GPUs, each capable of doing a small and simple task quickly. GPUs are used for graphical displays, and scientific computing problems such as training neutral networks and solving differential equations. We will not be talking about GPUs extensively in this guide.

## 9.2 Scatter-Gather on chromosomes



Scatter-Gather By Chromosome

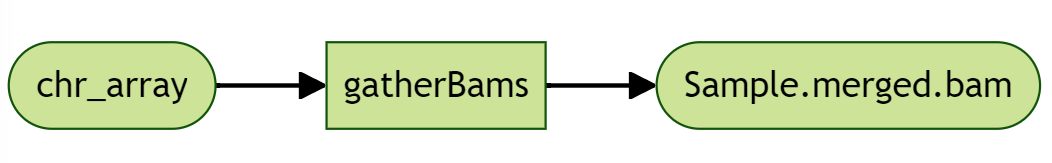
Suppose that one our tasks, ApplyBaseRecalibrator, is taking too long to run and taking on too much memory. We might first look into the [tool’s documentation](https://gatk.broadinstitute.org/hc/en-us/articles/360036898312-BaseRecalibrator) to see if there are ways use it more optimally, such as provide more CPUs, but we don’t see it. We turn to Embarrassingly Parallel Scatter-Gather, in which we split a BAM file by its chromosomes to create 23 BAM files, run ApplyBaseRecalibrator on each of these BAMs, and merge all of these calibrated BAM files together. This is called “Scatter-Gather” because we have to scatter our BAMs into smaller parts to run, and then gather them together in the end.



Split By Chromosome

We first write a new task to split our BAM file, which takes the original BAM, an array of chromosomes to scatter, and outputs an array of Files of the smaller BAM files. The output array of Files is stored as bams and indexFiles, and illustrated in the diagram as “chr\_array”.

# Split a BAM file by chromosomes  
task splitBamByChr {  
 input {  
 File bamToSplit  
 File baiToSplit  
 Array[String] chromosomes  
 }  
  
 String baseFileName = basename(bamToSplit, ".bam")  
  
 command <<<  
 set -eo pipefail  
 #For each chromosome...  
 for x in ~{sep=' ' chromosomes}; do  
 outputFile="~{baseFileName}\_${x}.bam"  
 samtools view -b -@ 3 "~{bamToSplit}" $x > $outputFile  
 samtools index $outputFile  
 done  
 # List all bam and bai files created  
 ls \*.bam > bam\_list.txt  
 ls \*.bam.bai > bai\_list.txt  
 >>>  
   
 output {  
 Array[File] bams = read\_lines("bam\_list.txt")  
 Array[File] indexFiles = read\_lines("bai\_list.txt")  
 }  
  
 runtime {  
 docker: "fredhutch/bwa:0.7.17"  
 cpu: 4  
 }

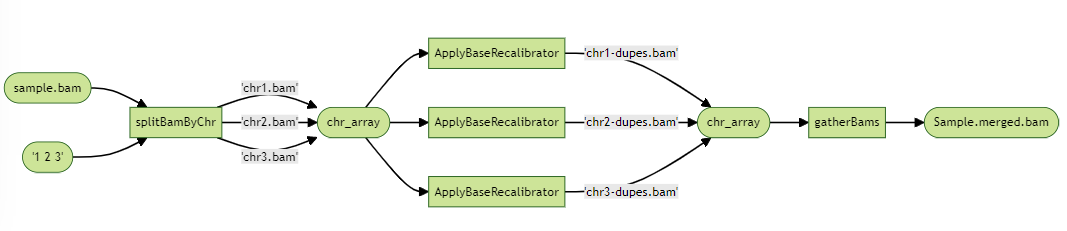


GatherBams

We also need a task to gather all the BAMs together, which takes in an array of Files, and outputs a merged BAM. The input array of Files is referred as bams, and illustrated in the diagram as chr\_array.

#Gather an array of BAMs  
task gatherBams {  
 input {  
 Array[File] bams  
 String clean\_baseName\_regex  
 }  
  
 String temp = basename(bams[0], ".bam")  
 String baseFileName = sub(temp, clean\_baseName\_regex, "")  
  
 command <<<  
 set -eo pipefail  
 samtools merge -c -@3 "~{baseFileName}".merged.bam ~{sep=' ' bams}  
 samtools index "~{baseFileName}".merged.bam  
 >>>  
   
 runtime {  
 cpu: 4  
 docker: "fredhutch/bwa:0.7.17"  
 }  
   
 output {  
 File merged\_bam = "~{baseFileName}.merged.bam"  
 File merged\_bai = "~{baseFileName}.merged.bam.bai"  
 }  
}

In our workflow, after MarkDuplicates task, we first use splitBamByChr to divvy up our BAM file, and then use a scatter over the output array of scattered BAMs to run ApplyBaseRecalibrator on each of the scatted BAMs. Finally, we take the output of ApplyBaseRecalibrator as an array to be merged together via gatherBams:



Scatter Gather Full Workflow

#Split by chromosomes  
 call splitBamByChr as tumorSplitBamByChr {  
 input:  
 bamToSplit = tumorMarkDuplicates.markDuplicates\_bam,  
 baiToSplit = tumorMarkDuplicates.markDuplicates\_bai,  
 chromosomes = chrs\_to\_split  
 }  
  
#Scatter by chromosomes  
scatter(i in range(length(tumorSplitBamByChr.indexFiles))) {  
 File tumorSubBam = tumorSplitBamByChr.bams[i]  
 File tumorSubBamIndex = tumorSplitBamByChr.indexFiles[i]  
  
 call ApplyBaseRecalibrator as tumorApplyBaseRecalibrator {  
 input:  
 input\_bam = tumorSubBam,  
 input\_bam\_index = tumorSubBamIndex,  
 dbSNP\_vcf = dbSNP\_vcf,  
 dbSNP\_vcf\_index = dbSNP\_vcf\_index,  
 known\_indels\_sites\_VCFs = known\_indels\_sites\_VCFs,  
 known\_indels\_sites\_indices = known\_indels\_sites\_indices,  
 refGenome = refGenome  
 }  
}  
  
#Gather all chromosomes together   
call gatherBams as tumorGatherBams {  
 input:  
 bams = tumorApplyBaseRecalibrator.recalibrated\_bam,  
 clean\_baseName\_regex = ".duplicates\_marked\_12.recal"  
}

Our final workflow:

Loading…

# 10 Appendix: Backends and Executors

Generally speaking, WDL workflows are quite portable thanks to their usage of Docker images to maintain software dependencies. However, the executor used to run WDLs and what backend they are being run upon can lead to specific scenarios where minor tweaks to your WDL are necessary to ensure portability.

## 10.1 Commonly used runtime attributes

Runtime attributes do not behave the same on all platforms. Here are how some of the most commonly used runtime attributes work on some of the most common WDL setups.

| Attribute | Fred Hutch HPC | Local Cromwell | Local miniwdl | Terra |
| --- | --- | --- | --- | --- |
| bootDiskSizeGB | n/a | n/a | n/a | Request a disk of this size to boot the Docker image (useful for very large Docker images) |
| cpu | Reserve at most this many cores | n/a | Reserve at most this many cores | Request a minimum of this many cores (scales with memory) |
| disks | n/a | n/a | n/a | Request this much disk size - soft requirement, if not specified, will request 10 GB |
| docker | Run the task in this Docker image | Run the task in this Docker image | Run the task in this Docker image | Run the task in this Docker image |
| memory | n/a | n/a | Maximum amount of memory to use | Minimum amount of memory to use (scales with CPU) |
| preemptible | n/a | n/a | n/a | Attempt running on a preemptible instance this many times, then switch to a non-preemptible |
| walltime, partition, modules, dockerSL, account | See back-end notes for Fred Hutch HPC below. | n/a | n/a | n/a |

## 10.2 General advice

* Consider using more runtime attributes, not fewer. miniwdl and Cromwell will ignore runtime attributes as necessary, so including a runtime attribute that only applies to a particular backend will not harm portability on other backends.
* The Dockstore CLI, as of v1.15, uses Cromwell to run WDLs. Advice about Cromwell will therefore also apply to the Dockstore CLI.
* If running a workflow with a scattered task on a local compute, consider using miniwdl instead of Cromwell.
* Whether you are using miniwdl or Cromwell locally, make sure Docker has enough resources to be able to download and run the Docker images specified in your WDL tasks.
* Workflow systems with a UI like Terra may become unresponsive if you run a task scattered more than 1000x. Outputs may need to be interacted with using an API specific to that backend.

## 10.3 Executor-specific notes

### 10.3.1 Cromwell

Cromwell running on a local machine has a tendency to heavily use system resources, especially when running scattered tasks. Sometimes, it will use too many resources at once. When this happens, your workflow’s tasks will tend to fail with exit code 137, or you will see hints about running out of memory in the logs of your tasks. You may also observe Docker becoming unresponsive. You can fix Docker by restarting Docker or your machine, but you will likely want to prevent this issue rather than keep having to restart Docker. To prevent this, modify concurrent-job-limit for your backend in the Cromwell configuration file.

The default Cromwell configuration file,, [can be found on Cromwell’s GitHub repo](https://github.com/broadinstitute/cromwell/blob/86/cromwell.example.backends/cromwell.examples.conf). To point Cromwell to a specific configuration file, run Cromwell with the flag -Dconfig.file= followed by the address of the config file to use.

If you are using the Dockstore CLI, [follow these instructions](https://docs.dockstore.org/en/stable/advanced-topics/dockstore-cli/local-cromwell-config.html?highlight=cromwell) to use a custom configuration file for Dockstore’s version of Cromwell. Because the Docker lockup issue mentioned previously isn’t uncommon, it is recommended to use a custom configuration file that limits concurrent-job-limit process on a local backend.

Other notes:

* Cromwell runs as a jar file, so it is very portable and does not need to be “installed” provided you have a modern Java runtime environment.
* If you just want to check workflows are valid, you can run womtool as a jar file (available on Cromwell’s GitHub page). This will check not only the WDL file you pass it, but also any WDLs it imports.
* Cromwell does not use call caching on most backends, but it is the default on Terra. For non-Terra backends, it can be enabled in the Cromwell configuration file.
* Cromwell supports the gpu and disks runtime attributes on certain backends. If using the gpu runtime attribute, make sure your task is set up correctly to properly use this resource.
* Generally speaking, Cromwell will be able to directly modify input files by default without causing permission errors, unlike miniwdl.
* Cromwell uses a system known as [hog factors](https://cromwell.readthedocs.io/en/stable/cromwell_features/HogFactors/) to adjust how task order is prioritized.

### 10.3.2 miniwdl

The authors of this course have informally observed that miniwdl appears to be “safer” in terms of using up too many resources than Cromwell. However, if you are sharing a compute with other users, it is still worth looking at the miniwdl configuration file and limiting how many Docker images you spin up at the same time.

The default miniwdl configuration file, which you can use as a template for setting up your own configuration file for miniwdl, [can be found on miniwdl’s GitHub repo](https://github.com/chanzuckerberg/miniwdl/blob/v1.11.1/WDL/runtime/config_templates/default.cfg). To point miniwdl to a specific configuration file, run miniwdl run with the --cfg flag.

Other notes:

* miniwdl is a Python program, and it must be installed via pip or pip3 to use.
* miniwdl’s equivalent to womtool is miniwdl check, but it also includes shellcheck to check the command section of your tasks.
* miniwdl supports call caching, but it is turned off by default.
* miniwdl does not support the gpu or disks runtime attributes and will ignore them if present in a task’s runtime section.
* By default, miniwdl does not duplicate input files. If your workflow only needs to read input files, this helps save disk space, but if your workflow directly modifies input files, this can result in permission errors. A simple fix is to run miniwdl with the --copy-input-files flag.

## 10.4 Backend-specific notes

### 10.4.1 HPCs

It is difficult to provide specific advice on HPCs, as they can vary greatly. Some general notes:

* Some HPCs do not support the use of Docker due to security concerns
  + HPCs that do not allow the use of Docker may be able to run WDLs using alternative container technologies such as podman or rootless Docker
* Some HPCs will use disks to determine which disk to run on, which can be useful for managing disk space

#### 10.4.1.1 Fred Hutch HPC

* [PROOF](https://sciwiki.fredhutch.org/datascience/proof/) is a Shiny frontend that launches Cromwell Server and lets you manage your jobs from a GUI. User guide can be found [here](https://sciwiki.fredhutch.org/dasldemos/proof-how-to/).
* The Fred Hutch HPC supports the use of multiple JSON files going into the same workflow
* The Fred Hutch HPC supports the use of Docker, so the docker runtime attribute works as expected
* memory and gpu runtime attributes are ignored, but cpu works as expected. Here is [some information](https://sciwiki.fredhutch.org/scicomputing/compute_jobs/#memory) on approximating memory use.
* walltime attribute is a string (“HH:MM:SS”) that specifies how much time you want to request for the task. Can also specify >1 day, e.g. “1-12:00:00” is 1 day+12 hours. The amount of walltime you can request depends on which cluster partition you want to use. More info [here](https://sciwiki.fredhutch.org/scicomputing/compute_jobs/#wall-time).
* partition attribute specifies the [cluster partition](https://sciwiki.fredhutch.org/compdemos/gizmo_partition_index/)to use. The default is “campus-new”.
* modules attribute is a a space-separated list of the [environment modules](https://sciwiki.fredhutch.org/scicomputing/compute_environments/) you’d like to load (in that order) prior to running the task. For example, “modules:”GATK/4.2.6.1-GCCcore-11.2.0 SAMtools/1.16.1-GCC-11.2.0”, the GATK module will be loaded first, followed by the SAMtools module.
* account attribute allows users connected to multiple PI accounts to specify which account to use for each task, to manage cluster allocations. An example of the value for this variable is “paguirigan\_a”, following the PI’s “lastname\_firstNameInitital” pattern.

### 10.4.2 GCP/Terra

GCP powers Terra, which is a Cromwell-based platform that is commonly used for running WDL workflows, but it can also be used on its own. Some special considerations need to be taken into account when designing workflows for compatibility on this backend.

#### 10.4.2.1 Preemptibles

Preemptible machines are an excellent way to save money when running workflows. A preemptible machine is a Google Cloud machine that is significantly cheaper (often less than half the price) than a standard one, at the cost of potentially being stopped suddenly. When running a task on a preemptible machine using Cromwell, if the preemptible is preempted (stopped suddenly), Cromwell will automatically retry the task. This does mean that in a worst case scenario, such a task could take about twice as long to run as normal and end up slightly more expensive, so you will want to weigh the costs and benefits. As a general rule of thumb, if you expect a task to take less than 2 hours, it is usually worth trying to use preemptible machines.

#### 10.4.2.2 Disk space

Running Cromwell on GCP is one of the few times that the disks runtime attribute is a soft requirement. Due to how Cromwell works, it must request a certain amount of disk space from GCP before running the task’s command section. In other words, you must know roughly how much disk space your task will use before it runs. If not specified in the runtime attribute of a task, Cromwell will request 10 GB of SSD space.

A helpful way to handle disk space in WDLs you anticipate will be run on GCP/Terra is to make disk size a function of the size of your task inputs. The task inputs and any private variable declarations made outside of the command section are all calculated before Cromwell requests a disk from GCP, so you can use this section to define a disk size function. For example:

version 1.0  
  
task do\_something {  
 input {  
 Array[File] some\_array\_of\_files  
 File one\_file  
 }  
   
 Int predicted\_disk\_space = ceil(size(some\_array\_of\_files, "GB")) + ceil(size(one\_file, "GB")) + 1  
 # size(x, "GB") returns a float representing the size of x in gigabytes  
 # ceil() is a WDL built-in function that rounds a float up into an integer  
   
 command <<<  
 pass  
 >>>  
   
 runtime {  
 docker: "ubuntu-latest"  
 disks: "local-disk " + predicted\_disk\_space + " HDD"  
 }  
}

Common pitfalls:

* Not wrapping size() in ceil() – the disks: runtime attribute requires an integer, not a float
* Not using "GB" when calling size() – if you don’t specify units for size() it will return bytes!
* Forgetting the space between “local-disk” and the integer (or the integer and “HDD”/“SSD”)

As a more in-depth example of how to use disk space, see the following WDL:

version 1.0  
  
task autoscale\_disk\_size\_for\_basically\_anything {  
 input {  
 File ref\_genome # required file  
 File? ref\_alt # optional file  
 Array[File] fastqs # required array of files  
 File tarball\_bams # uncompressed tarball of bam files  
 File compressed\_vcfs # compressed (tar.gz) tarball of VCFs  
 Int? addl\_disk\_gb # user-defined integer   
 }  
  
 # ref\_genome always exists, so we can size() on it safely. size() returns a float,  
 # but some backends require disk size is an integer, so we use ceil() to turn  
 # that float into an integer. ceil() always rounds up.  
 # You should ALWAYS explictly defining which unit you are using with size(). I   
 # recommend sticking with GB. If you do not define a unit, it is easy to accidentally  
 # request ludricious amounts of disk space, eg:  
 # 1 GB --> 1073741824 bytes --> 1073741824 GB --> 1.07 exabytes  
 Int disk\_ref = ceil(size(ref\_genome, "GB"))  
  
 # ref\_alt does not always exist, and size() will break if you try to use it on an  
 # undefined file. To prevent this, we use select\_first to fall back on the always-  
 # defined ref\_genome file if ref\_alt is undefined.  
 Int disk\_alt = ceil(size(select\_first([ref\_alt, ref\_genome]), "GB"))  
  
 # Of course, this means that if ref\_alt is undefined, then we don't really need   
 # disk\_alt, because the size of ref\_genome is already covered by disk\_ref.  
 Int disk\_both\_refs = if defined(ref\_alt) then disk\_alt + disk\_ref else disk\_ref  
  
 # You might be wondering why we don't just do this:  
 # Int disk\_alt = if defined(ref\_alt) then ceil(size(ref\_alt, "GB")) else 0  
 # This is synatically valid, but generally speaking it is best to avoid running  
 # WDL built-in functions (except select\_all, select\_first, and defined) on optional  
 # variables as much as possible.   
  
 # When dealing with arrays, size() will return the sum of all Files in the array.  
 Int disk\_fastqs = ceil(size(fastqs, "GB"))  
  
 # If you are passing in a tarball, you will likely be expanding it. If so, consider  
 # doubling the disk size to account for it in both states.  
 Int disk\_tarball = 2\*ceil(size(tarball\_bams, "GB"))  
  
 # A compressed tar.gz is harder to predict. As a very rough rule of thumb, 10x should  
 # be enough. If you are passing in a tar.gz made from an upstream task, you may want  
 # to decrease this number based on the compression ratio you used.  
 Int disk\_compressed = 10\*ceil(size(compressed\_vcfs))  
  
 # Because addl\_disk\_gb is an optional integer with no fallback, you cannot add it to   
 # another integer, eg, this will error as "Non-numeric operand to + operator":  
 # Int final\_disk\_size = disk\_both\_refs + disk\_fastqs + disk\_tarball + disk\_compressed + addl\_disk\_gb  
 # We have to use select\_first() to coerce it into a not-optional integer first.  
 Int disk\_addl = select\_first([addl\_disk\_gb, 0])  
  
 # Finally, let's add everything up. (Recall that disk\_both\_refs accounts for ref\_genome and  
 # also ref\_alt.)  
 Int final\_disk\_size = disk\_both\_refs + disk\_fastqs + disk\_tarball + disk\_compressed + disk\_addl  
  
 command <<<  
 echo disk\_ref: ~{disk\_ref}  
 echo disk\_alt: ~{disk\_alt}  
 echo disk\_both\_refs: ~{disk\_both\_refs}  
 echo disk\_fastqs: ~{disk\_fastqs}  
 echo disk\_tarball: ~{disk\_tarball}  
 echo disk\_compressed: ~{disk\_compressed}  
 echo addl\_disk\_gb: ~{addl\_disk\_gb}  
 echo final\_disk\_size: ~{final\_disk\_size}  
 >>>  
  
 runtime {  
 # On GCP!Cromwell, if you do not define the units in the runtime attribute, this  
 # will be interpreted as the size in GB. This is why it's so easy to accidentally  
 # request the wrong amount of disk space if you do not define the units for size();  
 # In one place WDL defaults to bytes but in the other it defaults to GB.  
 disks: "local-disk " + final\_disk\_size + " HDD"  
 }  
}

Loading…

# About the Authors

These credits are based on our [course contributors table guidelines](https://www.ottrproject.org/more_features.html#giving-credits-to-contributors).

| Credits | Names |
| --- | --- |
| Lead Content Instructor(s) | Ash O’Farrell, Sitapriya Moorthi, Chris Lo |
| Content Editor(s)/Reviewer(s) | Amy Paguirigan, Carrie Wright, Ted Laderas |
| Content Director(s) | Amy Paguirigan |
| Content Consultants | Ash O’Farrell |
| **Production** |  |
| Content Publisher(s) | Carrie Wright |
| **Technical** |  |
| Template Publishing Engineers | [Candace Savonen](https://www.cansavvy.com/), [Carrie Wright](https://carriewright11.github.io/), [Ava Hoffman](https://www.avahoffman.com/) |
| Publishing Maintenance Engineer | [Candace Savonen](https://www.cansavvy.com/) |
| Technical Publishing Stylists | [Carrie Wright](https://carriewright11.github.io/), [Ava Hoffman](https://www.avahoffman.com/), [Candace Savonen](https://www.cansavvy.com/) |
| Package Developers ([ottrpal](https://github.com/jhudsl/ottrpal)) [Candace Savonen](https://www.cansavvy.com/), [John Muschelli](https://johnmuschelli.com/), [Carrie Wright](https://carriewright11.github.io/) |  |

## ─ Session info ───────────────────────────────────────────────────────────────  
## setting value  
## version R version 4.3.2 (2023-10-31)  
## os Ubuntu 22.04.4 LTS  
## system x86\_64, linux-gnu  
## ui X11  
## language (EN)  
## collate en\_US.UTF-8  
## ctype en\_US.UTF-8  
## tz Etc/UTC  
## date 2025-01-10  
## pandoc 3.1.1 @ /usr/local/bin/ (via rmarkdown)  
##   
## ─ Packages ───────────────────────────────────────────────────────────────────  
## package \* version date (UTC) lib source  
## bookdown 0.41 2024-10-16 [1] CRAN (R 4.3.2)  
## cachem 1.0.8 2023-05-01 [1] RSPM (R 4.3.0)  
## cli 3.6.2 2023-12-11 [1] RSPM (R 4.3.0)  
## devtools 2.4.5 2022-10-11 [1] RSPM (R 4.3.0)  
## digest 0.6.34 2024-01-11 [1] RSPM (R 4.3.0)  
## ellipsis 0.3.2 2021-04-29 [1] RSPM (R 4.3.0)  
## evaluate 0.23 2023-11-01 [1] RSPM (R 4.3.0)  
## fastmap 1.1.1 2023-02-24 [1] RSPM (R 4.3.0)  
## fs 1.6.3 2023-07-20 [1] RSPM (R 4.3.0)  
## glue 1.7.0 2024-01-09 [1] RSPM (R 4.3.0)  
## htmltools 0.5.7 2023-11-03 [1] RSPM (R 4.3.0)  
## htmlwidgets 1.6.4 2023-12-06 [1] RSPM (R 4.3.0)  
## httpuv 1.6.14 2024-01-26 [1] RSPM (R 4.3.0)  
## knitr 1.48 2024-07-07 [1] CRAN (R 4.3.2)  
## later 1.3.2 2023-12-06 [1] RSPM (R 4.3.0)  
## lifecycle 1.0.4 2023-11-07 [1] RSPM (R 4.3.0)  
## magrittr 2.0.3 2022-03-30 [1] RSPM (R 4.3.0)  
## memoise 2.0.1 2021-11-26 [1] RSPM (R 4.3.0)  
## mime 0.12 2021-09-28 [1] RSPM (R 4.3.0)  
## miniUI 0.1.1.1 2018-05-18 [1] RSPM (R 4.3.0)  
## pkgbuild 1.4.3 2023-12-10 [1] RSPM (R 4.3.0)  
## pkgload 1.3.4 2024-01-16 [1] RSPM (R 4.3.0)  
## profvis 0.3.8 2023-05-02 [1] RSPM (R 4.3.0)  
## promises 1.2.1 2023-08-10 [1] RSPM (R 4.3.0)  
## purrr 1.0.2 2023-08-10 [1] RSPM (R 4.3.0)  
## R6 2.5.1 2021-08-19 [1] RSPM (R 4.3.0)  
## Rcpp 1.0.12 2024-01-09 [1] RSPM (R 4.3.0)  
## remotes 2.4.2.1 2023-07-18 [1] RSPM (R 4.3.0)  
## rlang 1.1.4 2024-06-04 [1] CRAN (R 4.3.2)  
## rmarkdown 2.25 2023-09-18 [1] RSPM (R 4.3.0)  
## sessioninfo 1.2.2 2021-12-06 [1] RSPM (R 4.3.0)  
## shiny 1.8.0 2023-11-17 [1] RSPM (R 4.3.0)  
## stringi 1.8.3 2023-12-11 [1] RSPM (R 4.3.0)  
## stringr 1.5.1 2023-11-14 [1] RSPM (R 4.3.0)  
## urlchecker 1.0.1 2021-11-30 [1] RSPM (R 4.3.0)  
## usethis 2.2.3 2024-02-19 [1] RSPM (R 4.3.0)  
## vctrs 0.6.5 2023-12-01 [1] RSPM (R 4.3.0)  
## xfun 0.48 2024-10-03 [1] CRAN (R 4.3.2)  
## xtable 1.8-4 2019-04-21 [1] RSPM (R 4.3.0)  
## yaml 2.3.8 2023-12-11 [1] RSPM (R 4.3.0)  
##   
## [1] /usr/local/lib/R/site-library  
## [2] /usr/local/lib/R/library  
##   
## ──────────────────────────────────────────────────────────────────────────────

# 11 References