Developing WDL Workflows

February, 2024

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.

## 0.2 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 transferrable 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 - WDLizing your bash scripts into tasks

## 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, and configure settings for the execution engine.

## 0.4 Relevant resources

An excellent WDL companion resource that is not platform dependent is [OpenWDL Docs](https://docs.openwdl.org/en/stable/). 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.

# 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 ~{this} notation. To access a workflow-level variable in a workflow, it is referenced just by its name without any special notation. 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}  
 >>>  
}  
  
workflow my\_workflow {  
 input {  
 File fq  
 }  
   
 String basename\_of\_fq = basename(fq)  
   
 call do\_something {  
 input:  
 fastq = fq,  
 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"  
 }  
}  
  
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  
 }  
}

## 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 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, 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 and Cromwell. Both can run WDLs on a local machine, High Performance Computing (HPC), or cloud computing backend. 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.

# 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:

1. BwaMem aligns the samples to the reference genome (hg19).
2. MarkDuplicates marks PCR duplicates.
3. ApplyBaseRecalibrator perform base quality recalibration.
4. Mutect2 performs paired somatic mutation calling.
5. annovar annotates the called somatic mutations.

## 2.2 Workflow testing strategy

As we build out our workflow, how do we know 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 is representative of “real” data.
* You 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 protien 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.

# 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  
 Int threads  
 }  
[...]  
}

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 to input. This can be done via an array, but for now we’ll list everything separately to make sure nothing is being left out.

We also want to define a default value for threads so that someone who does not know much about threading can still use the workflow. We want to use this workflow on human data, so we’ll go a little high for the default number of threads and set it to sixteen. In WDL, we do this by declaring Int threads = 16. Make sure to put this in the task (or workflow) inputs section – if you put it elsewhere, that variable cannot be changed from its default value, so it will always be 16.

task BwaMem {  
 input {  
 # main input  
 File input\_fastq  
  
 # options  
 Int threads = 16  
  
 # 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. You could, for example, call a bioinformatics tool and then reprocess the outputs in the same WDL task.

Within the command section, we refer to those variables using ~{this} syntax. For instance, if the user sets threads to 8, then the -t ~{threads} part of the command section below will be interpreted as -t 8.

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

Why we 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 {  
 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 <<<  
 some\_other\_string="FOO"  
 echo ~{some\_string}  
 echo $some\_other\_string  
 >>>  
}

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 variabls, like so:

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

Why we put quotation marks around a String or File variables in Commands.

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  
 Int threads = 16  
  
 # 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 ~{threads} -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. 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 requently 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  
 Int threads = 16  
 }  
  
 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 mved 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".

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  
 Int threads = 16  
 }  
   
 # 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 = "illumina"  
 String platform\_info = "PL:" + platform # Create the platform information  
  
 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 ~{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  
  
 >>>  
}

## 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 filesystem. 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, and 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, 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 bioinformatic 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 essentially 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  
 Int threads = 16  
 }  
   
 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 = "illumina"  
 String platform\_info = "PL:" + platform # Create the platform information  
  
  
 command <<<  
 set -eo pipefail  
  
 #can we iterate through a struct??  
 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  
 >>>  
  
 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.

version 1.0  
  
workflow mutation\_calling {  
 input {  
 # Sample info  
 File tumorFastq  
 # Reference Genome information  
 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  
 #Optional BwaMem threading variable  
 Int? bwa\_mem\_threads  
 }  
  
 # Map reads to reference  
 call BwaMem {  
 input:  
 input\_fastq = input\_fastq,  
 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,  
 threads = bwa\_mem\_threads  
  
 }  
   
 # Outputs that will be retained when execution is complete  
 output {  
 File alignedBamSorted = BwaMem.analysisReadySorted  
 }  
  
 parameter\_meta {  
 sampleFastq: "Sample .fastq (expects Illumina)"  
 ref\_fasta: "Reference genome to align reads to"  
 ref\_fasta\_index: "Reference genome index file (created by bwa index)  
 ref\_dict: "Reference genome dictionary file (created by bwa index)"  
 ref\_amb: "Reference genome non-ATCG file (created by bwa index)"  
 ref\_ann: "Reference genome ref seq annotation file (created by bwa index)"  
 ref\_bwt: "Reference genome binary file (created by bwa index)"  
 ref\_pac: "Reference genome binary file (created by bwa index)"  
 ref\_sa: "Reference genome binary file (created by bwa index)"  
 }  
# End workflow  
}  
  
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  
 Int threads = 16  
 }  
   
 String read\_group\_id = "ID:" + base\_file\_name  
 String sample\_name = "SM:" + base\_file\_name  
 String platform = "illumina"  
 String platform\_info = "PL:" + platform # Create the platform information  
  
 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 ~{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 -n -@ 15 -o "~{base\_file\_name}.sorted\_query\_aligned.bam" "~{base\_file\_name}.aligned.bam"  
  
 >>>  
 output {  
 File analysisReadyBam = "~{base\_file\_name}.aligned.bam"  
 File analysisReadySorted = "~{base\_file\_name}.sorted\_query\_aligned.bam"  
 }  
 runtime {  
 memory: "48 GB"  
 cpu: 16  
 docker: "ghcr.io/getwilds/bwa:0.7.17"  
 disks: "local-disk 100 SSD"  
 }  
}

## 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 needed to run this single-task workflow:

{  
 "mutation\_calling.tumorFastq": "/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"  
}

If you are not running on the Fred Hutch HPC, you’ll need to modify your JSON file to point to wherever you have the data files stored. You can download the same fastq we’re using from [our sandbox repo](https://github.com/fhdsl/WDL-sandbox/tree/main/data), and the reference files can be generated via samtools index or [downloaded from the Broad Institute’s mirror](https://data.broadinstitute.org/snowman/hg19/).

# 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.
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 minidata\_mutation\_calling\_v1 {  
 input {  
 # Sample info  
 File sampleFastq  
 # Reference Genome information  
 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.

version 1.0  
  
workflow minidata\_mutation\_calling\_v1 {  
 input {  
 # Sample info  
 File sampleFastq  
 # Reference Genome information  
 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  
 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  
 File annovarTAR  
 String annovar\_protocols  
 String annovar\_operation  
 String ref\_name  
 }  
   
 # 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  
 }  
  
 call ApplyBaseRecalibrator {  
 input:  
 input\_bam = MarkDuplicates.markDuplicates\_bam,  
 input\_bam\_index = MarkDuplicates.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,  
 ref\_dict = ref\_dict,  
 ref\_fasta = ref\_fasta,  
 ref\_fasta\_index = ref\_fasta\_index  
 }  
  
 call Mutect2TumorOnly {  
 input:  
 input\_bam = ApplyBaseRecalibrator.recalibrated\_bam,  
 input\_bam\_index = ApplyBaseRecalibrator.recalibrated\_bai,  
 ref\_dict = ref\_dict,  
 ref\_fasta = ref\_fasta,  
 ref\_fasta\_index = ref\_fasta\_index,  
 genomeReference = af\_only\_gnomad,  
 genomeReferenceIndex = af\_only\_gnomad\_index  
 }  
   
 call annovar {  
 input:  
 input\_vcf = Mutect2TumorOnly.output\_vcf,  
 ref\_name = ref\_name,  
 annovarTAR = annovarTAR,  
 annovar\_operation = annovar\_operation,  
 annovar\_protocols = annovar\_protocols  
 }  
   
 # Outputs that will be retained when execution is complete  
 output {  
 File alignedBamSorted = BwaMem.analysisReadySorted  
 File markDuplicates\_bam = MarkDuplicates.markDuplicates\_bam  
 File markDuplicates\_bai = MarkDuplicates.markDuplicates\_bai  
 File analysisReadyBam = ApplyBaseRecalibrator.recalibrated\_bam   
 File analysisReadyIndex = ApplyBaseRecalibrator.recalibrated\_bai  
 File Mutect\_Vcf = Mutect2TumorOnly.output\_vcf  
 File Mutect\_VcfIndex = Mutect2TumorOnly.output\_vcf\_index  
 File Mutect\_AnnotatedVcf = annovar.output\_annotated\_vcf  
 File Mutect\_AnnotatedTable = annovar.output\_annotated\_table  
 }  
}  
  
  
  
# TASK DEFINITIONS  
  
# Align fastq file to the reference genome  
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  
 }  
   
 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 = "illumina"  
 String platform\_info = "PL:" + platform # Create the platform information  
  
  
 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  
 >>>  
  
 output {  
 File analysisReadySorted = "~{base\_file\_name}.sorted\_query\_aligned.bam"  
 }  
   
 runtime {  
 memory: "48 GB"  
 cpu: 16  
 docker: "fredhutch/bwa:0.7.17"  
 }  
}  
  
# Mark duplicates  
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}.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  
 File ref\_dict  
 File ref\_fasta  
 File ref\_fasta\_index  
 }  
   
 String base\_file\_name = basename(input\_bam, ".duplicates\_marked.bam")  
   
 String ref\_fasta\_local = basename(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 ~{ref\_fasta} .  
 mv ~{ref\_fasta\_index} .  
 mv ~{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: "broadinstitute/gatk:4.1.4.0"  
 }  
}  
  
  
  
# Mutect 2 calling  
  
task Mutect2TumorOnly {  
 input {  
 File input\_bam  
 File input\_bam\_index  
 File ref\_dict  
 File ref\_fasta  
 File ref\_fasta\_index  
 File genomeReference  
 File genomeReferenceIndex  
 }  
  
 String base\_file\_name = basename(input\_bam, ".recal.bam")  
 String ref\_fasta\_local = basename(ref\_fasta)  
 String genomeReference\_local = basename(genomeReference)  
  
command <<<  
 set -eo pipefail  
  
 mv ~{ref\_fasta} .  
 mv ~{ref\_fasta\_index} .  
 mv ~{ref\_dict} .  
 mv ~{genomeReference} .  
 mv ~{genomeReferenceIndex} .  
  
 gatk --java-options "-Xms16g" Mutect2 \  
 -R ~{ref\_fasta\_local} \  
 -I ~{input\_bam} \  
 -O preliminary.vcf.gz \  
 --germline-resource ~{genomeReference\_local} \  
   
 gatk --java-options "-Xms16g" FilterMutectCalls \  
 -V preliminary.vcf.gz \  
 -O ~{base\_file\_name}.mutect2.vcf.gz \  
 -R ~{ref\_fasta\_local} \  
 --stats preliminary.vcf.gz.stats \  
   
>>>  
  
runtime {  
 docker: "broadinstitute/gatk:4.1.4.0"  
 memory: "24 GB"  
 cpu: 1  
 }  
  
output {  
 File output\_vcf = "${base\_file\_name}.mutect2.vcf.gz"  
 File output\_vcf\_index = "${base\_file\_name}.mutect2.vcf.gz.tbi"  
 }  
  
}  
  
  
  
  
# annotate with annovar  
task annovar {  
 input {  
 File input\_vcf  
 String ref\_name  
 File annovarTAR  
 String annovar\_protocols  
 String annovar\_operation  
}  
 String base\_vcf\_name = basename(input\_vcf, ".vcf.gz")  
   
 command <<<  
 set -eo pipefail  
   
 tar -xzvf ~{annovarTAR}  
   
 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 : "perl:5.28.0"  
 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"  
 }  
}

# 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 minidata\_mutation\_calling\_v1 {  
 input {  
 File sampleFastq  
 referenceGenome refGenome  
 ...  
 }  
 # Map reads to reference  
 call BwaMem {  
 input:  
 input\_fastq = sampleFastq,  
 refGenome = refGenome  
 }  
}

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:

{  
 "minidata\_mutation\_calling\_v1.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"  
 },  
 "minidata\_mutation\_calling\_v1.dbSNP\_vcf\_index": "/fh/fast/paguirigan\_a/pub/ReferenceDataSets/genome\_data/human/hg19/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):

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 = "illumina"  
 String platform\_info = "PL:" + platform # Create the platform information  
  
  
 command <<<  
 set -eo pipefail  
  
 #can we iterate through a struct??  
 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: "fredhutch/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,  
 annovarTAR = annovarTAR,  
 annovar\_operation = annovar\_operation,  
 annovar\_protocols = annovar\_protocols  
 }

Putting everything together in the workflow:

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 minidata\_mutation\_calling\_v1 {  
 input {  
 File sampleFastq  
  
 referenceGenome refGenome  
   
 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  
   
 File annovarTAR  
 String annovar\_protocols  
 String annovar\_operation  
 }  
   
 # Map reads to reference  
 call BwaMem {  
 input:  
 input\_fastq = sampleFastq,  
 refGenome = refGenome  
 }  
   
 call MarkDuplicates {  
 input:  
 input\_bam = BwaMem.analysisReadySorted  
 }  
  
 call ApplyBaseRecalibrator {  
 input:  
 input\_bam = MarkDuplicates.markDuplicates\_bam,  
 input\_bam\_index = MarkDuplicates.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 Mutect2TumorOnly {  
 input:  
 input\_bam = ApplyBaseRecalibrator.recalibrated\_bam,  
 input\_bam\_index = ApplyBaseRecalibrator.recalibrated\_bai,  
 refGenome = refGenome,  
 genomeReference = af\_only\_gnomad,  
 genomeReferenceIndex = af\_only\_gnomad\_index  
 }  
   
 call annovar {  
 input:  
 input\_vcf = Mutect2TumorOnly.output\_vcf,  
 ref\_name = refGenome.ref\_name,  
 annovarTAR = annovarTAR,  
 annovar\_operation = annovar\_operation,  
 annovar\_protocols = annovar\_protocols  
 }  
   
 # Outputs that will be retained when execution is complete  
 output {  
 File alignedBamSorted = BwaMem.analysisReadySorted  
 File markDuplicates\_bam = MarkDuplicates.markDuplicates\_bam  
 File markDuplicates\_bai = MarkDuplicates.markDuplicates\_bai  
 File analysisReadyBam = ApplyBaseRecalibrator.recalibrated\_bam   
 File analysisReadyIndex = ApplyBaseRecalibrator.recalibrated\_bai  
 File Mutect\_Vcf = Mutect2TumorOnly.output\_vcf  
 File Mutect\_VcfIndex = Mutect2TumorOnly.output\_vcf\_index  
 File Mutect\_AnnotatedVcf = annovar.output\_annotated\_vcf  
 File Mutect\_AnnotatedTable = annovar.output\_annotated\_table  
 }  
}  
  
  
  
  
  
  
# 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 = "illumina"  
 String platform\_info = "PL:" + platform # Create the platform information  
  
  
 command <<<  
 set -eo pipefail  
  
 #can we iterate through a struct??  
 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: "fredhutch/bwa:0.7.17"  
 }  
}  
  
# Mark duplicates (not SPARK, for some reason that does something weird)  
task MarkDuplicates {  
 input {  
 File input\_bam  
 }  
  
 String base\_file\_name = basename(input\_bam, ".sorted\_query\_aligned.bam")  
 String output\_bam = "~{base\_file\_name}.duplicates\_marked.bam"  
 String output\_bai = "~{base\_file\_name}.duplicates\_marked.bai"  
 String metrics\_file = "~{base\_file\_name}.duplicate\_metrics"  
  
 command <<<  
 gatk MarkDuplicates \  
 --INPUT ~{input\_bam} \  
 --OUTPUT ~{output\_bam} \  
 --METRICS\_FILE ~{metrics\_file} \  
 --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 = "~{output\_bam}"  
 File markDuplicates\_bai = "~{output\_bai}"  
 File duplicate\_metrics = "~{metrics\_file}"  
 }  
}  
  
# 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} #redundant? markduplicates already does this?  
  
 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: "broadinstitute/gatk:4.1.4.0"  
 }  
}  
  
  
  
# Mutect 2 calling  
  
task Mutect2TumorOnly {  
 input {  
 File input\_bam  
 File input\_bam\_index  
 referenceGenome refGenome  
 File genomeReference  
 File genomeReferenceIndex  
 }  
  
 String base\_file\_name = basename(input\_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 ~{input\_bam} \  
 -O preliminary.vcf.gz \  
 --germline-resource ~{genomeReference\_local} \  
   
 gatk --java-options "-Xms16g" FilterMutectCalls \  
 -V preliminary.vcf.gz \  
 -O ~{base\_file\_name}.mutect2.vcf.gz \  
 -R ~{ref\_fasta\_local} \  
 --stats preliminary.vcf.gz.stats \  
   
>>>  
  
runtime {  
 docker: "broadinstitute/gatk:4.1.4.0"  
 memory: "24 GB"  
 cpu: 1  
 }  
  
output {  
 File output\_vcf = "${base\_file\_name}.mutect2.vcf.gz"  
 File output\_vcf\_index = "${base\_file\_name}.mutect2.vcf.gz.tbi"  
 }  
  
}  
  
  
  
  
# annotate with annovar  
task annovar {  
 input {  
 File input\_vcf  
 String ref\_name  
 File annovarTAR  
 String annovar\_protocols  
 String annovar\_operation  
}  
 String base\_vcf\_name = basename(input\_vcf, ".vcf.gz")  
   
 command <<<  
 set -eo pipefail  
   
 tar -xzvf ~{annovarTAR}  
   
 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 : "perl:5.28.0"  
 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"  
 }  
}

# 6 Using Arrays For Parallelization and Other Use Cases

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 primative 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 4, 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](https://en.wikipedia.org/wiki/Immutable_object), if you wish to add values to an array, you will need to define a new array.

## 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 minidata\_mutation\_calling\_chapter6 {  
 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. They can only “see” one file at a time. However, outside the scatter, every task is considered in the context of every sample, so every output of those scattered tasks become arrays. As a result, our 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 [here](../resources/chapter-6.wdl).

## 6.4 Referencing an array in a task

Because each task only takes in one sample, we are not directly inputting arrays into a file. However, it’s important to know how to do this. 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"  
 }  
}

# 7 Appendix: Backends and Executors

Generally speaking, WDL workflows are quite portable thanks to their usage of Docker images to maintain software depenendencies. 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.

## 7.1 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 | Maximum amount of memory to use | 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 | How much [walltime](https://en.wikipedia.org/wiki/Elapsed_real_time) to request for a task | n/a | n/a | n/a |

## 7.2 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.

## 7.3 Executor-specific notes

### 7.3.1 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.

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 cacheing 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.

### 7.3.2 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.

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 cacheing, 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.

## 7.4 Backend-specific notes

### 7.4.1 GCP

#### 7.4.1.1 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”)

#### 7.4.1.2 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.

### 7.4.2 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

#### 7.4.2.1 Fred Hutch HPC

* The Fred Hutch HPC supports the use of Docker, so the docker runtime attribute works as expected
* The memory and gpu runtime attributes are ignored, but cpu works as expected
* The Fred Hutch HPC supports the use of multiple JSON files going into the same workflow

# 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 4.0.2 (2020-06-22)  
## os Ubuntu 20.04.5 LTS   
## system x86\_64, linux-gnu   
## ui X11   
## language (EN)   
## collate en\_US.UTF-8   
## ctype en\_US.UTF-8   
## tz Etc/UTC   
## date 2024-02-29   
##   
## ─ Packages ───────────────────────────────────────────────────────────────────  
## package \* version date lib source   
## assertthat 0.2.1 2019-03-21 [1] RSPM (R 4.0.5)   
## bookdown 0.24 2023-03-28 [1] Github (rstudio/bookdown@88bc4ea)   
## cachem 1.0.7 2023-02-24 [1] CRAN (R 4.0.2)   
## callr 3.5.0 2020-10-08 [1] RSPM (R 4.0.2)   
## cli 3.6.1 2023-03-23 [1] CRAN (R 4.0.2)   
## crayon 1.3.4 2017-09-16 [1] RSPM (R 4.0.0)   
## desc 1.2.0 2018-05-01 [1] RSPM (R 4.0.3)   
## devtools 2.3.2 2020-09-18 [1] RSPM (R 4.0.3)   
## digest 0.6.25 2020-02-23 [1] RSPM (R 4.0.0)   
## ellipsis 0.3.1 2020-05-15 [1] RSPM (R 4.0.3)   
## evaluate 0.20 2023-01-17 [1] CRAN (R 4.0.2)   
## fastmap 1.1.1 2023-02-24 [1] CRAN (R 4.0.2)   
## fs 1.5.0 2020-07-31 [1] RSPM (R 4.0.3)   
## glue 1.4.2 2020-08-27 [1] RSPM (R 4.0.5)   
## htmltools 0.5.5 2023-03-23 [1] CRAN (R 4.0.2)   
## knitr 1.33 2023-03-28 [1] Github (yihui/knitr@a1052d1)   
## magrittr 2.0.3 2022-03-30 [1] CRAN (R 4.0.2)   
## memoise 2.0.1 2021-11-26 [1] CRAN (R 4.0.2)   
## pkgbuild 1.1.0 2020-07-13 [1] RSPM (R 4.0.2)   
## pkgload 1.1.0 2020-05-29 [1] RSPM (R 4.0.3)   
## prettyunits 1.1.1 2020-01-24 [1] RSPM (R 4.0.3)   
## processx 3.4.4 2020-09-03 [1] RSPM (R 4.0.2)   
## ps 1.4.0 2020-10-07 [1] RSPM (R 4.0.2)   
## R6 2.4.1 2019-11-12 [1] RSPM (R 4.0.0)   
## remotes 2.2.0 2020-07-21 [1] RSPM (R 4.0.3)   
## rlang 1.1.0 2023-03-14 [1] CRAN (R 4.0.2)   
## rmarkdown 2.10 2023-03-28 [1] Github (rstudio/rmarkdown@02d3c25)  
## rprojroot 2.0.3 2022-04-02 [1] CRAN (R 4.0.2)   
## sessioninfo 1.1.1 2018-11-05 [1] RSPM (R 4.0.3)   
## stringi 1.5.3 2020-09-09 [1] RSPM (R 4.0.3)   
## stringr 1.4.0 2019-02-10 [1] RSPM (R 4.0.3)   
## testthat 3.0.1 2023-03-28 [1] Github (R-lib/testthat@e99155a)   
## usethis 1.6.3 2020-09-17 [1] RSPM (R 4.0.2)   
## withr 2.3.0 2020-09-22 [1] RSPM (R 4.0.2)   
## xfun 0.26 2023-03-28 [1] Github (yihui/xfun@74c2a66)   
## yaml 2.2.1 2020-02-01 [1] RSPM (R 4.0.3)   
##   
## [1] /usr/local/lib/R/site-library  
## [2] /usr/local/lib/R/library

# 8 References