WDL Workflows Guide

February, 2024

Table of Contents

# About this Course

This WDL workflow development guide 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. The audience should be able to comprehend 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 Curriculum

The course covers…

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

Welcome!

* Review of basic WDL syntax
* How to use input JSONs
* (optional) Installing Docker and miniwdl
* How to run simple workflows locally

## 1.1 Review of basic 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  
 }  
}

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  
 }  
}

## 1.2 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.

In WDL, like most programming languages, variables have a specific type. 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 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.

## 1.3 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.

The two most popular WDL executors are miniwdl and Cromwell. Both can run WDLs on a local machine, HPC, or cloud computing backend. 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.

### 1.3.1 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.

### 1.3.2 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).

### 1.3.3 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!"  
 }  
}

### 1.3.4 Troubleshooting

#### 1.3.4.1 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.

#### 1.3.4.2 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.

#### 1.3.4.3 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)  
  
 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\tID:foo\tSM:foo2' \  
 "~{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"  
  
 >>>  
}

## 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: "fredhutch/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 analysisReadyBam = "~{base\_file\_name}.aligned.bam"  
 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 analysisReadyBam = base\_file\_name + ".aligned.bam"  
 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 analysisReadyBam = "some\_folder/~{base\_file\_name}.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  
 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 base\_file\_name = basename(input\_fastq, ".fastq")  
 String ref\_fasta\_local = basename(ref\_fasta)  
  
 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\tID:foo\tSM:foo2' \  
 "~{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: "fredhutch/bwa:0.7.17"  
 disks: "local-disk 100 SSD"  
 }  
}

## 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 minidata\_test\_alignment {  
 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  
 #Optional BwaMem threading variable  
 Int? bwa\_mem\_threads  
 }  
  
 # 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,  
 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 base\_file\_name = basename(input\_fastq, ".fastq")  
 String ref\_fasta\_local = basename(ref\_fasta)  
  
 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\tID:foo\tSM:foo2' \  
 "~{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: "fredhutch/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.

# 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 |
| --- | --- |
| **Pedagogy** |  |
| Lead Content Instructor(s) | [FirstName LastName](link%20to%20personal%20website) |
| Lecturer(s) (include chapter name/link in parentheses if only for specific chapters) - make new line if more than one chapter involved | Delivered the course in some way - video or audio |
| Content Author(s) (include chapter name/link in parentheses if only for specific chapters) - make new line if more than one chapter involved | If any other authors besides lead instructor |
| Content Contributor(s) (include section name/link in parentheses) - make new line if more than one section involved | Wrote less than a chapter |
| Content Editor(s)/Reviewer(s) | Checked your content |
| Content Director(s) | Helped guide the content direction |
| Content Consultants (include chapter name/link in parentheses or word “General”) - make new line if more than one chapter involved | Gave high level advice on content |
| Acknowledgments | Gave small assistance to content but not to the level of consulting |
| **Production** |  |
| Content Publisher(s) | Helped with publishing platform |
| Content Publishing Reviewer(s) | Reviewed overall content and aesthetics on publishing platform |
| **Technical** |  |
| Course Publishing Engineer(s) | Helped with the code for the technical aspects related to the specific course generation |
| 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/) |  |
| **Art and Design** |  |
| Illustrator(s) | Created graphics for the course |
| Figure Artist(s) | Created figures/plots for course |
| Videographer(s) | Filmed videos |
| Videography Editor(s) | Edited film |
| Audiographer(s) | Recorded audio |
| Audiography Editor(s) | Edited audio recordings |
| **Funding** |  |
| Funder(s) | Institution/individual who funded course including grant number |
| Funding Staff | Staff members who help with funding |

## ─ 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 2024-02-21   
##   
## ─ 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

# 4 References