

Dockstore Fundamentals:

Introduction to Docker and Descriptors for Reproducible Analysis

Louise Cabansay, Software Engineer, UC Santa Cruz Genomics Institute
Andrew Duncan, Software Engineer, Ontario Institute for Cancer Research
Denis Yuen, Senior Software Engineer, Ontario Institute for Cancer Research

Learning Objectives

- What is Dockstore?
- Introduction to Docker
- Introduction to Descriptors
 - Overview of descriptor languages (CWL, WDL, Nextflow)
 - Practice using WDL
- Dockstore Usage
 - Key features
 - Best practices
- Overall goal today is to give you a good foundation in the basics of Dockstore.
We will be providing a variety of take home materials to guide you on where to go next

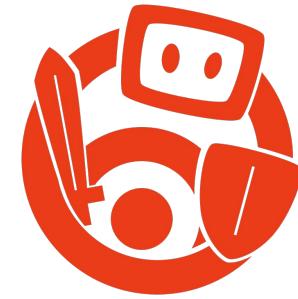
Format and Setup

- Lecture + Examples (slides)
- Q/A:
 - #dockstore-fundamentals on Discord can also be a great place for questions
 - Click “yes” in Zoom when finished with an exercise
 - Introduce TAs



Format and Setup

- Hands-on practice (Instruqt)
 - Message us on Discord if you did not get the link
 - Browser-based tutorial environment for your exercises
 - Close the sidebar in Instruqt for additional screen space



Chrome Browser Required (currently a bug on Instruqt w/ Firefox)

Download : <https://www.google.com/chrome/>

What is Dockstore?

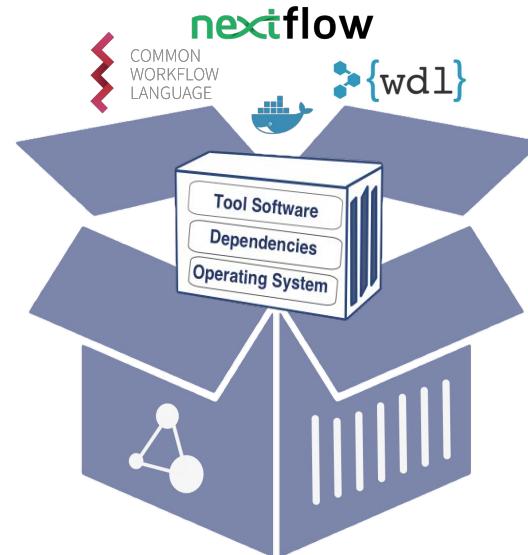
Dockstore is a free and open source platform for sharing scientific tools and workflows.

Portability

Interoperability

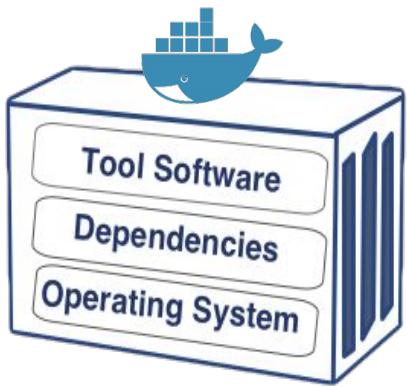
Reproducibility

“An app store for bioinformatics”

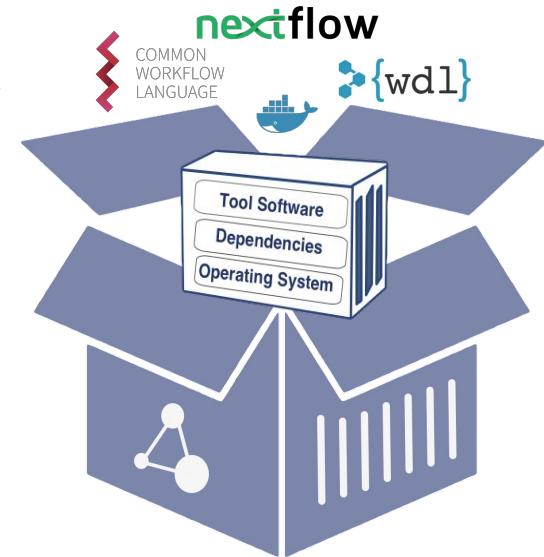


Portability:

Container



Descriptor



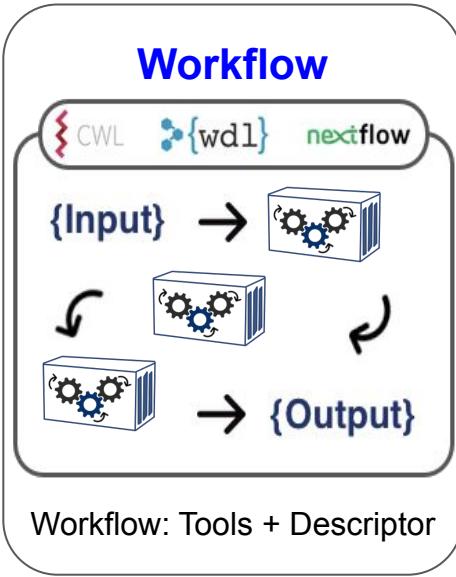
Software is “packaged” using container technology and described using descriptor languages

- Analysis can be moved from environment-to-environment (local machines, clouds, servers) and yet be guaranteed to run on anything that supports Docker

What's on Dockstore? tools and workflows

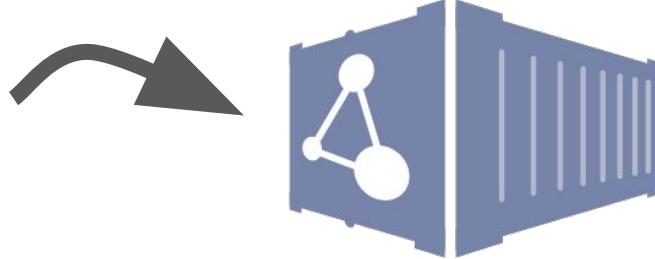


OR



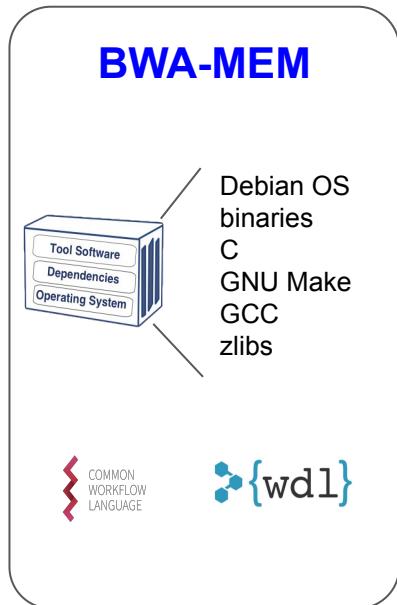
A tool uses a single container and performs a single action or step that is outlined by a descriptor.

A workflow can use multiple containers and executes multiple actions or steps, still outlined by a descriptor

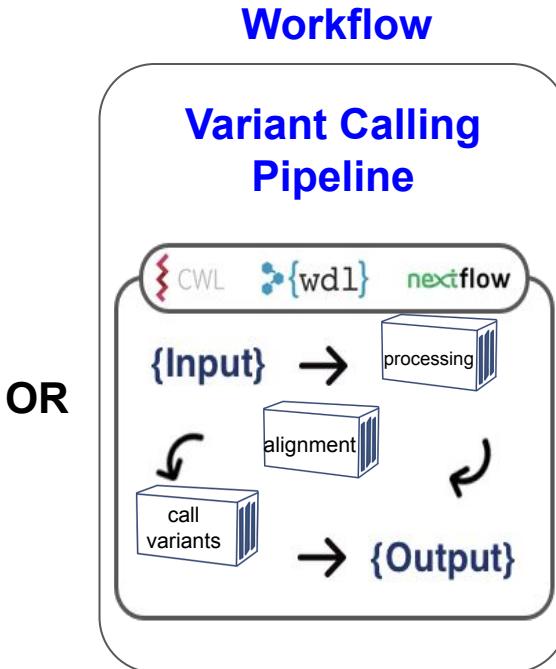


Example:

Tool



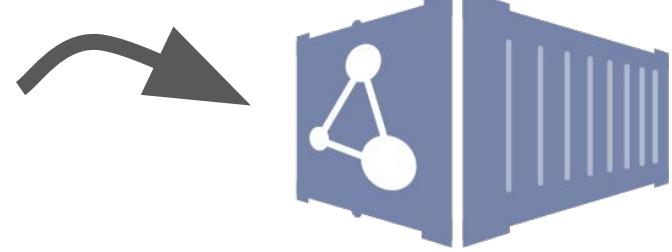
Workflow



OR

A tool uses a single container and performs a single action or step that is outlined by a descriptor.
(ex: alignment)

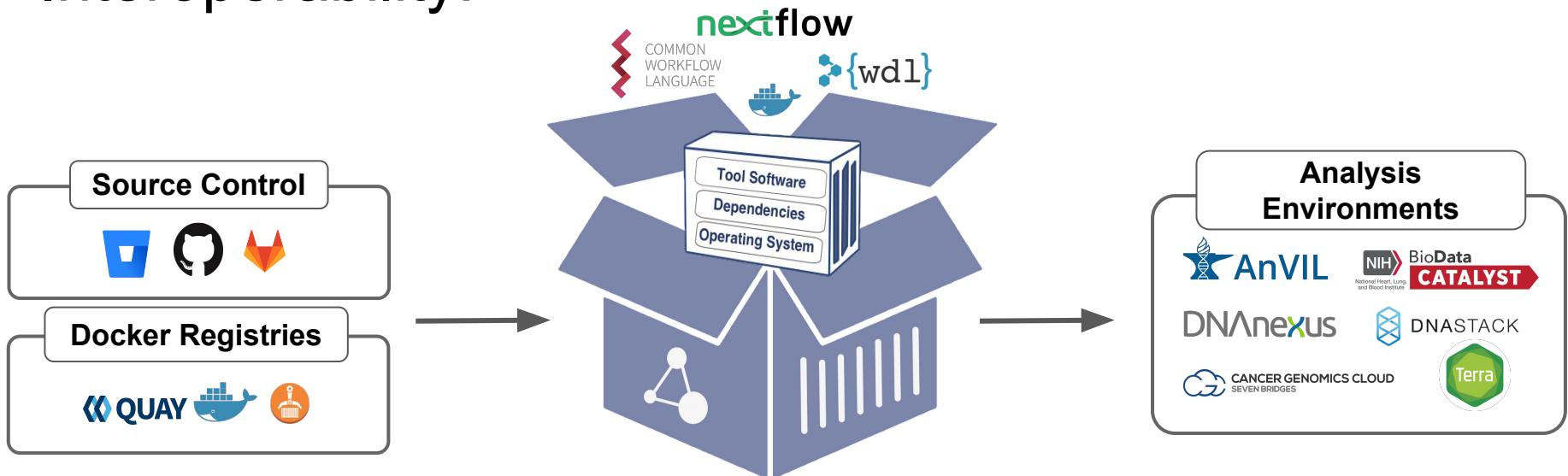
A workflow can use multiple containers and executes multiple actions or steps, still outlined by a descriptor



*note: you *can* register a single tool as a workflow on Dockstore, but a multi-step workflow cannot be registered as a tool.

ex: BWA-MEM can technically be registered as both

Interoperability:



Integration with various sites allows Dockstore to function as centralized catalog of bioinformatics tools and workflows

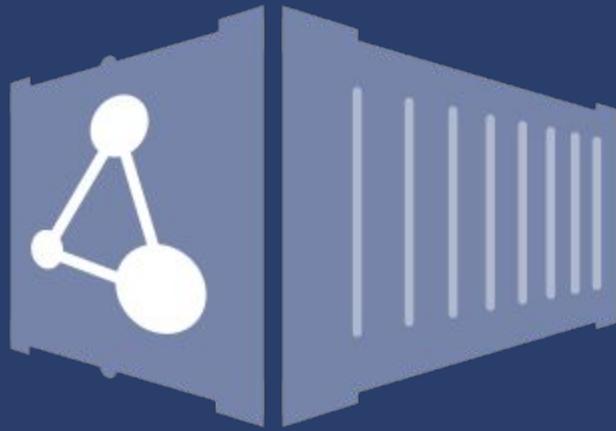
By following GA4GH API standards, Dockstore enables users to run tools and workflows in their preferred compute and analysis environments

700+ tools and workflows published to Dockstore



Reproducibility: Create, Share, Use

- Dockstore is a place for researchers and developers to share their work so that others can also use it
- The combination of containers and workflow languages minimizes redundant and error prone installation
 - Increases the transparency of analysis methods
 - Allows others to verify results and apply existing methods into their own research
- Other Dockstore features to increase reproducibility:
 - Versioning, snapshots, and metadata handling
 - Generating Digital Object Identifiers (DOIs) via Zenodo
 - Organizations and Collections for sharing and findability
 - and more!



Docker Basics

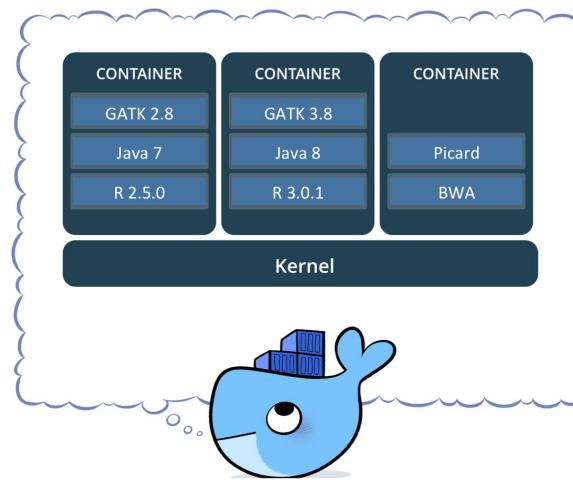
*as used on Dockstore

What is a container? What is Docker?

Container:

A container encapsulates all the software dependencies associated with running a program.

- Allows for portable software that runs quickly and reliably from one computing environment to another.



Docker:

A particularly popular brand of container

- Some users may wish to [explore alternatives](#) like Singularity

What kinds of problems are solved by containers?

- Installation problems
 - Software was built on a different OS (executable files don't run)
 - Install documentation is unclear or out of date
- Dependency problems
 - Software requires different version than what is available on machine (ex. Java, Python)
 - Multiple programs have shared dependencies, but different versions of those dependencies
- Portability problems
 - Software can be run on any host OS that has Docker installed

Docker Concepts: Container, Image, Registry

Container:

A *running* image

- Packaged up, isolated software environment
- All dependencies included

**the terms container and image are often used interchangeably, but there is a slight distinction.

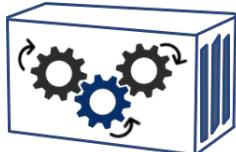


Image:

Packaged up code with its all dependencies *at rest*.

- Allows for portable software that runs quickly and reliably from one computing environment to another.

Official Image

- Official images on Docker Hub are regularly updated and scanned for vulnerabilities 

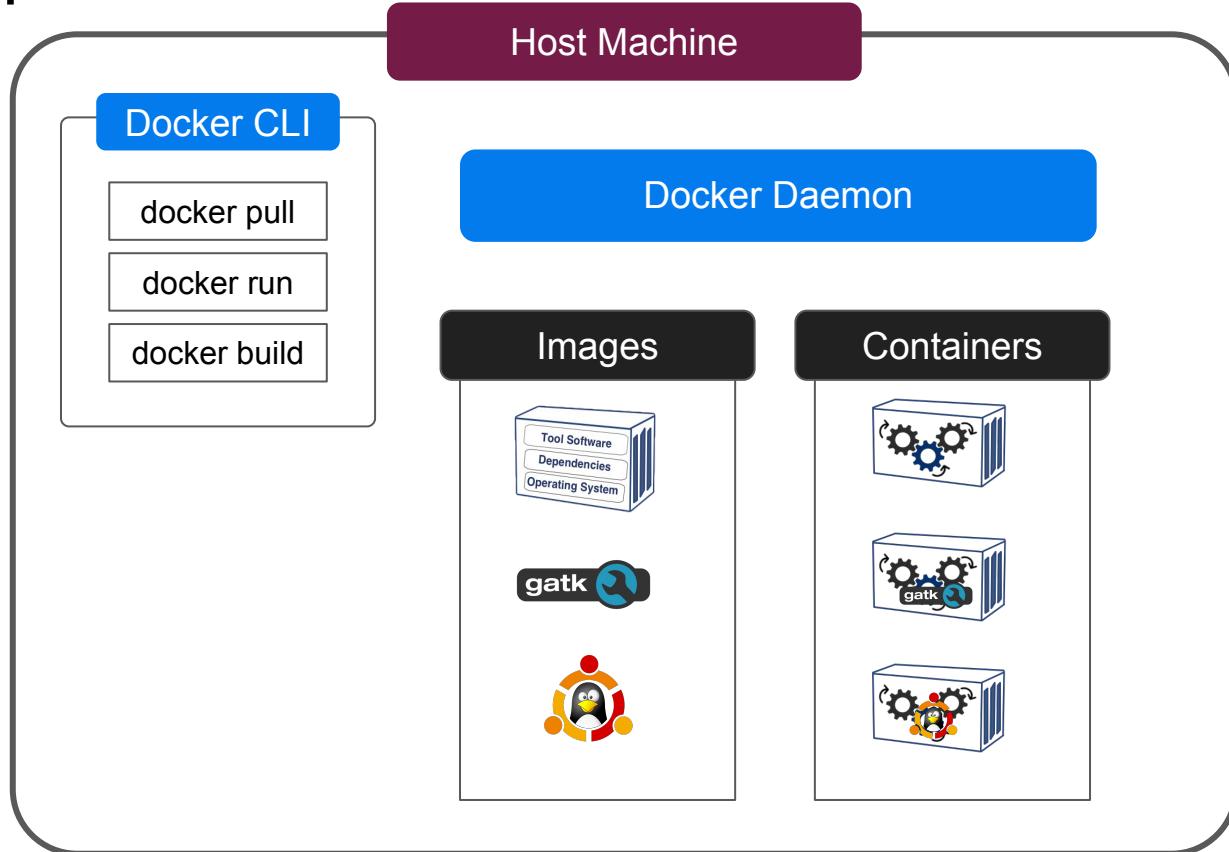
Registry:

Repositories where users can store images privately or publicly in the cloud.

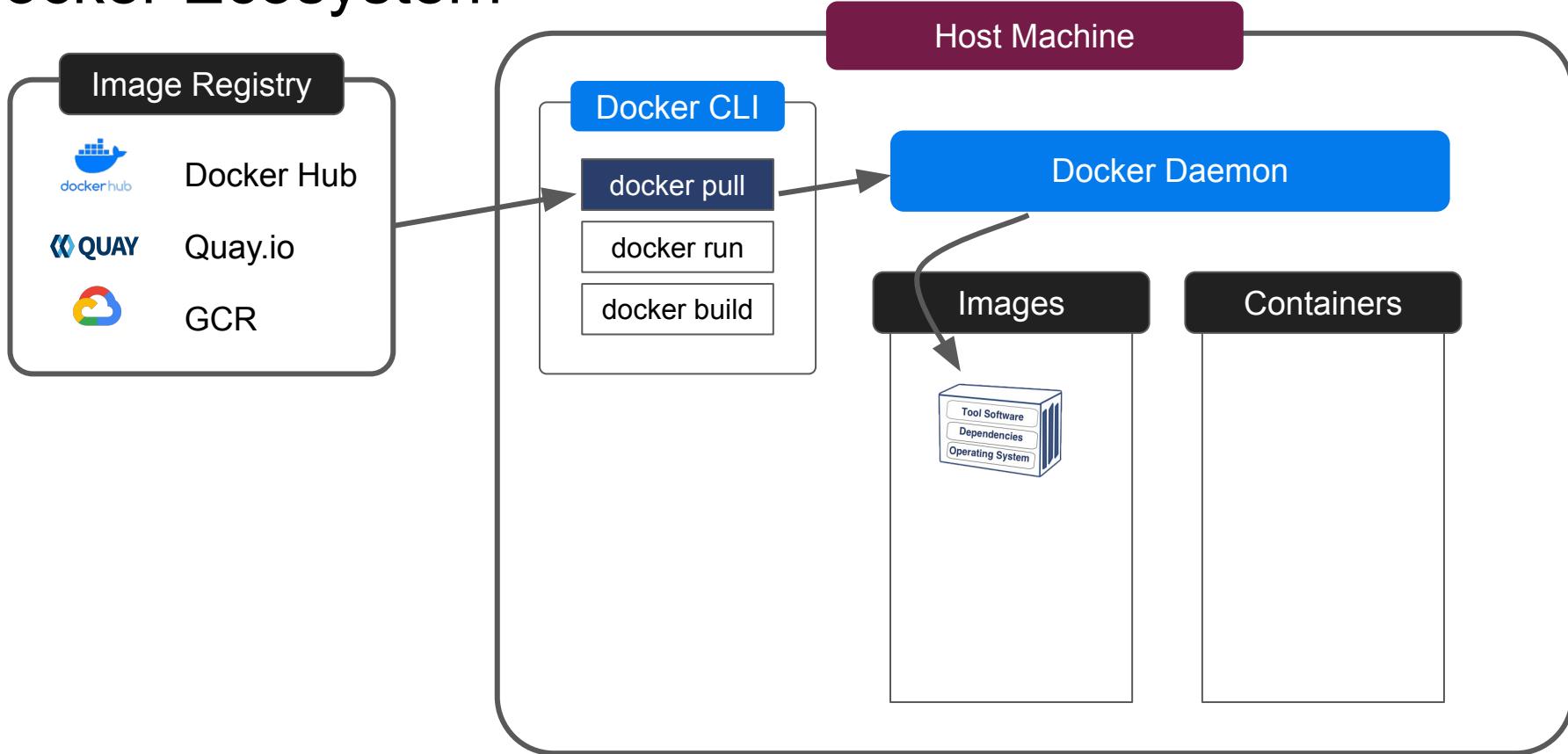
Dockstore itself does not host images, but rather gets them from Image Registries:

- Docker Hub
- Quay.io
- Google Container Registry (GCR)

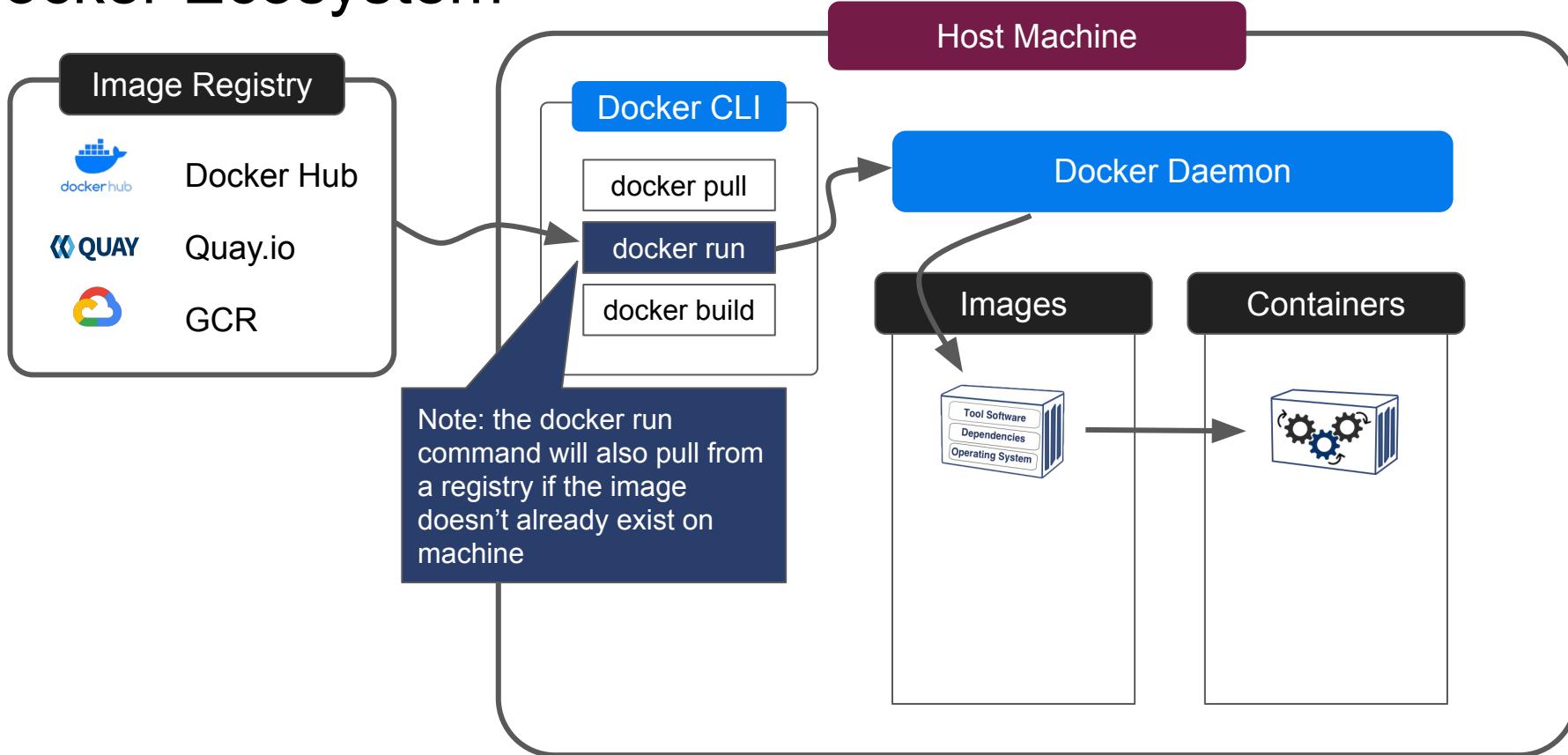
Docker Ecosystem



Docker Ecosystem



Docker Ecosystem



Start Up Instruqt

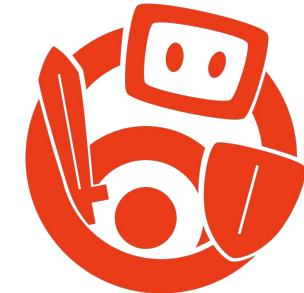
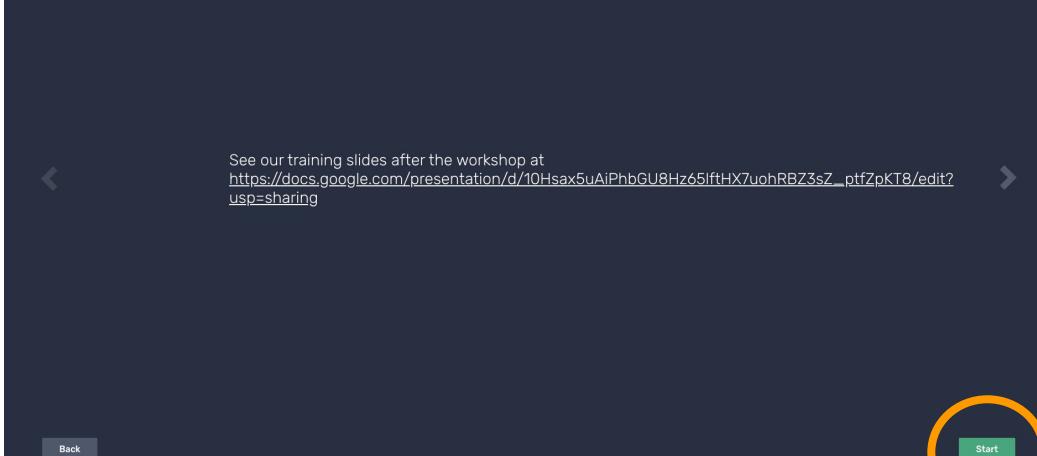
Launch Dockstore CLI environment
Start a Dockstore CLI environment for exercise 1 and 2

Description Notes Tabs Scripts

START

See our training slides after the workshop at
https://docs.google.com/presentation/d/10Hsax5uAiPhbGU8Hz65lftHX7uohRBZ3sZ_ptfZpKT8/edit?usp=sharing

Back Start



Wait for start to show up

Docker Client (CLI)

A command-line utility for:

- downloading and building Docker images
- running Docker containers
- managing both images and containers (think disk space, cleanup)

```
docker [sub-command] [-flag options] [arguments]
```

Basic Docker Sub Commands:

Docker has a whole library of commands, here are some basic examples:

Display system-wide information about your installation of docker:

```
docker info [OPTIONS]
```

Managing docker images:

```
docker image [COMMAND]
```

Managing docker containers:

```
docker container [COMMAND]
```

Running docker containers:

```
docker run [-flags] [registry name]/[path to image repository]:[tag] [arguments]
```

How are containers commonly used?

Run and done

1. Execute docker run command
2. Actions executed in container (stuff happens)
3. Container stops running after completion

How are containers commonly used?

Your main method of containers!!

Run and done

1. Execute docker run command
2. Actions executed in container (stuff happens)
3. Container stops running after completion

Run continuously

1. Execute docker run command
2. Container starts and runs in the background *continuously*
3. Other processes can interact with the container (stuff happens)
4. ***Container keeps running unless it is stopped***

How do I ‘run’ a container?

base
run command

dockerhub*
quay.io
gcr.io

Only specify the registry if its *not* dockerhub

The ‘version’ of the image you want to run

```
docker run [-flag options] [registry name]/[path to image repository]:[tag] [arguments]
```

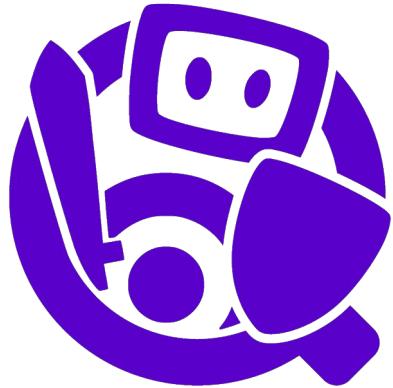
Additional options

Ex. --name to specify a name for the container

Both official containers and user containers are available

generally, the arguments are what gets passed into the container

ex: the command you want to run



instruqt



Exercise #1a: Running containers

```
docker run [-flag options] [registry name]/[path to image repository]:[tag] [arguments]
```

Exercises:

Use the whalesay container from Docker Hub to print a welcome message

```
docker run docker/whalesay cowsay "fill me in"
```

Exploring containers interactively

```
docker run [-flag options] [registry name]/[path to image repository]:[tag] [arguments]
```

-i -t

- The ‘-it’ flags drop you *inside* the container
- (-i) Keeps STDIN open for interactive use and (-t) allocates a terminal
- You can then interact with the container’s terminal like a normal command line

Example:

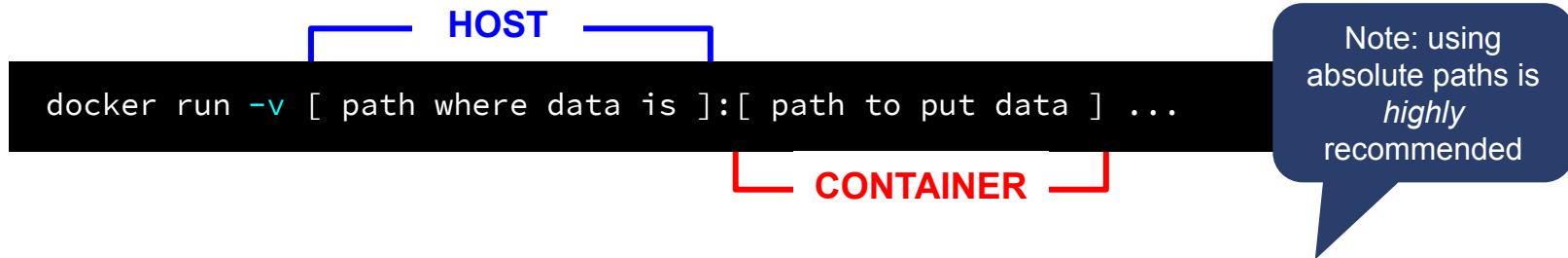
Enter the samtools container and confirm that samtools is installed

```
docker run -it quay.io/ldcabansay/samtools:latest
```

Sharing data between host and container

Bind mounts (-v) (aka two-way data binding)

- Map a host directory to a directory in a container
- Any files added to either directory is available in the other



Output stored in the container directory `/tmp/data` will also be available on the host at `/usr/data`

```
docker run -v /usr/data:/tmp/data ...
```



Exercise #1b: exploring containers

```
docker run [-flag options] [registry name]/[path to image repository]:[tag] [arguments]
```

-i -t -v

HOST

```
docker run -v [ path where data is ]:[ path to put data ] ...
```

CONTAINER

Exercise(s):

Enter the samtools container, but this time bring in some data!

```
docker run -it -v /root/bcc2020-training/data:/data quay.io/ldcabansay/samtools:latest
```

Convert a sam file to a bam file using the samtools container.

```
docker run -v /root/bcc2020-training/data:/data quay.io/ldcabansay/samtools:latest  
samtools view -S -b /data/mini.sam -o /data/mini.bam
```

Data binding: In-depth (Extra Reading)

- Useful tips and tricks to know about databinding
- Advanced features and caveats to keep in mind
- Read more here: <https://docs.docker.com/storage/>

Dockerfiles: Custom Images

- Sometimes an existing image isn't available for the software we want to use or an existing image may be lacking something we require
- **Dockerfiles** are used to create our own custom images
 - Starts from a base image
 - Contains a series of steps that set up our environment
- We can then also share these custom images via an image registry

Primer: How is software installed and used?

Package managers

- Managed collection of software with automated install, upgrades, and removal

Executable Files or Binaries

(ex: *.jar, *.c, grep, tar, diff, md5sum)

- software that has already been built or compiled into an executable file.

Building or running from source files

- Compiling from source build an executable (requires compiler & dependencies)
- For languages that don't need compiling (python), necessary runtime dependencies required in environment



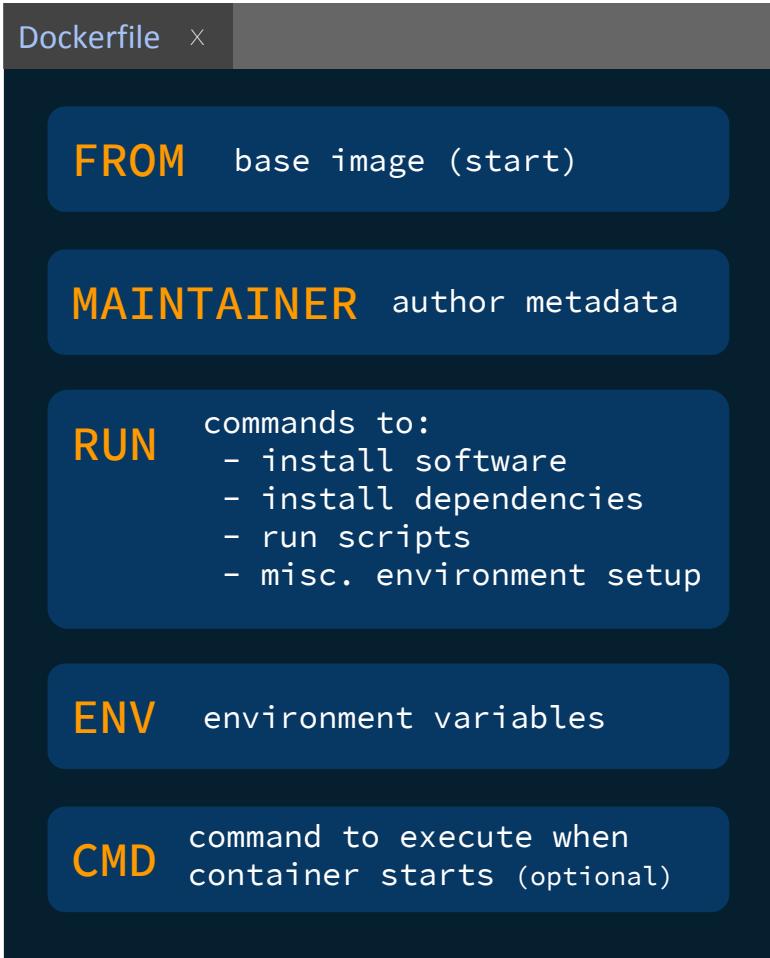
Author of dockerfile programmatically details the software installation and any other steps for environment setup

Image built from the dockerfile can then be used for 'off-the-shelf' software usage by others.

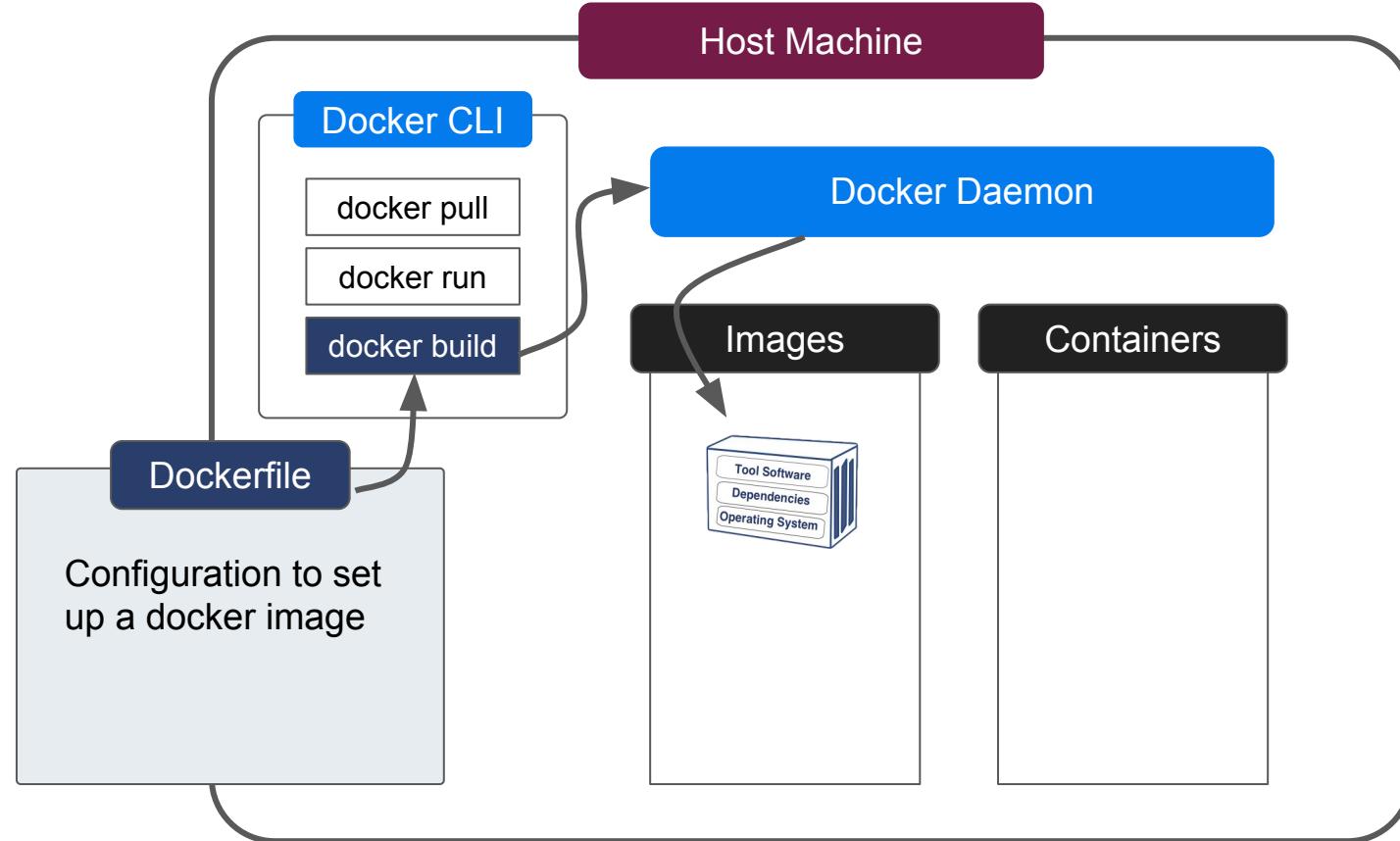
Dockerfiles Overview:

A simple text file with instructions to build an image:

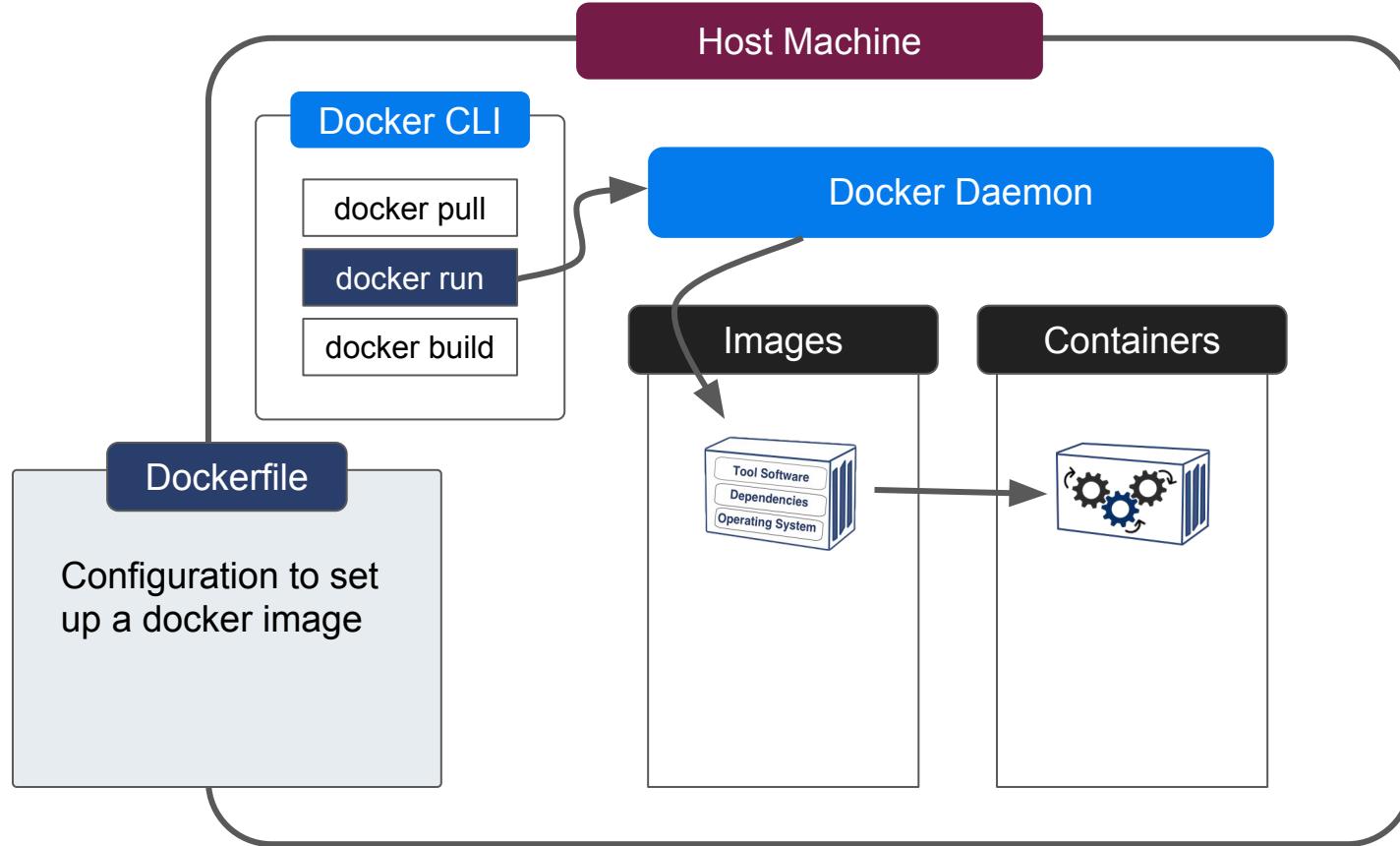
- YAML syntax
- Start from a base image
- Metadata
- Install software and dependencies
- Set up scripts
- Other environment prep
- Define commands to run when container starts



Dockerfiles - local

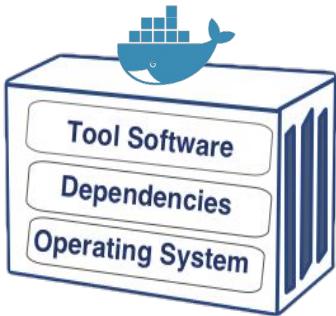


Dockerfiles - local



Example: BWA (via package manager)

- Images of containers are built from Dockerfiles
- Dockerfiles describe the packaged up environment:
 - operating system or base image to build upon
 - dependencies needed for the software
 - the actual analysis software



```
Dockerfile  ×  
1 #####  
2 # Dockerfile to build a sample container for bwa  
3 #####  
4  
5 # Start with a base image  
6 FROM ubuntu:18.04  
7  
8 # Add file author/maintainer and contact info (optional)  
9 MAINTAINER Louise Cabansay <lcabansa@ucsc.edu>  
10  
11 #set user you want to install packages as  
12 USER root  
13  
14 #update package manager & install dependencies (if any)  
15 RUN apt update  
16  
17 # install analysis software from package manager  
18 RUN apt install -y bwa  
19  
20
```

Basic Commands: docker build

```
docker build [-flag options] [build context]
```

-t -f

(-t) : Builds and creates a tag **v1.0** for a **bwa** image (if in dockerfile directory)

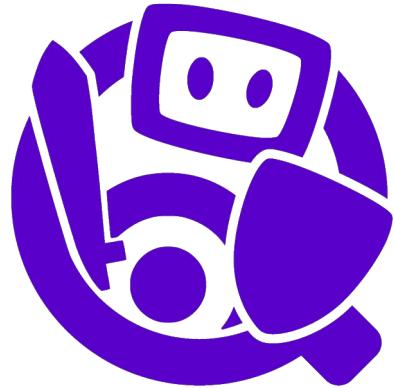
```
docker build -t bwa:v1.0 .
```

(-f) : Build a specific Dockerfile by providing path to file (relative to build context)

```
docker build -t bwa:v1.0 -f dockerfiles/bwa/Dockerfile .
```

View built Docker images

```
docker image ls
```



instruqt

Exercise #2a: Writing your first Dockerfile: tabix

Dockerfiles describe the packaged up environment:

- operating system or base image to build upon:
 - ubuntu:18.04
- Install dependencies needed: N/A
- Install actual analysis software:
 - tabix

```
Dockerfile x  
1  # Start with a base image  
2  FROM { base image name }  
3  
4  
5  # Add file author/maintainer and contact info (optional)  
6  MAINTAINER {your name} <youremail@research.edu>  
7  
8  # set user you want to install packages as  
9  USER root  
10  
11 # update package manager & install dependencies (if any)  
12 RUN apt update  
13  
14 # install analysis software from package manager  
15 RUN apt install -y { software package name }  
16  
17  
18  
19
```

Build an image from Dockerfile:

```
docker image build -t { name } -f { path to dockerfile } .
```



Exercise #2b: Try out your new container!

```
docker container run [-flag options] [registry name]/[path to image repository]:[tag] [args]
```

Exercises:

1. Verify that your image was built (get the image ID to use in part 2)

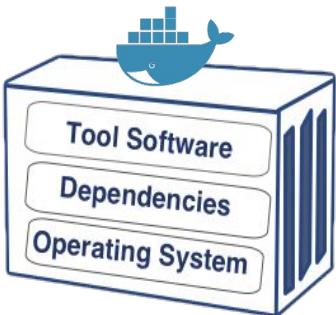
```
docker image ls
```

2. Use your local image to view the tabix command help

```
docker run [image id] tabix
```

Ex: bamstats (executable)

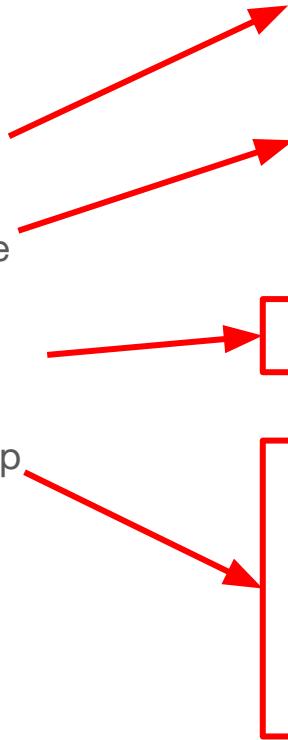
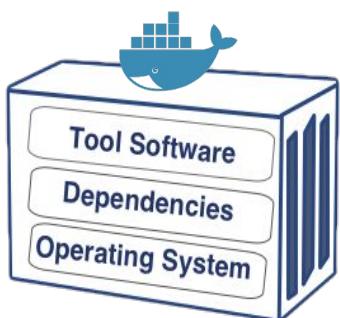
- Dockerfiles describe the packaged up environment:
 - operating system or base image to build upon
 - dependencies needed for the software
 - the actual analysis software
 - Here it's an already compiled executable
 - scripts or commands to complete software and environment set-up



```
Dockerfile x
1 # Start with a base image
2 FROM ubuntu:14.04
3
4 # Add file author/maintainer and contact info (optional)
5 MAINTAINER Brian OConnor <briandoconnor@gmail.com>
6
7 # install software dependencies
8 USER root
9 RUN apt-get -m update && apt-get install -y wget unzip \
10 openjdk-7-jre zip
11
12
13 # manual software installation from source
14 # get the tool and install it in /usr/local/bin
15 RUN wget -q http://downloads.sourceforge.net/project
16 /bamstats/BAMStats-1.25.zip
17
18 # commands/scripts to finish software setup
19 RUN unzip BAMStats-1.25.zip && \
20 rm BAMStats-1.25.zip && \
21 mv BAMStats-1.25 /opt/
22
23 COPY bin/bamstats /usr/local/bin/
24 RUN chmod a+x /usr/local/bin/bamstats
25
26 # switch back to the ubuntu user so this tool (and the
27 files written) are not owned by root
28 RUN groupadd -r -g 1000 ubuntu && useradd -r -g ubuntu -u
29 1000 -m ubuntu
30 USER ubuntu
31
32 # command /bin/bash is executed when container starts
33 CMD ["/bin/bash"]
34
35
36
37
```

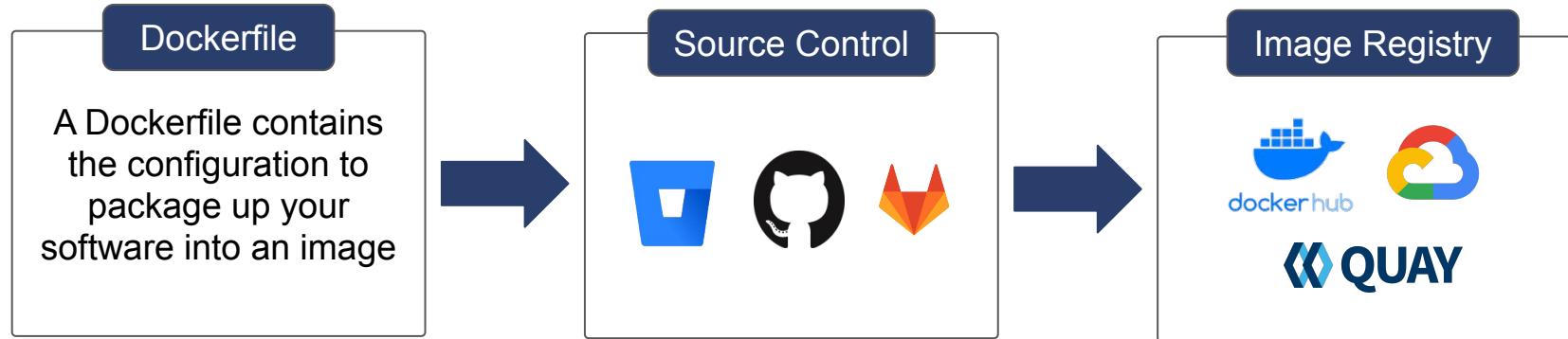
Example: samtools (compile from source files)

- Images of containers are built from Dockerfiles
- Dockerfiles describe the packaged up environment:
 - operating system or base image to build upon
 - dependencies needed for the software
 - the actual analysis software
 - Scripts or commands to complete environment set- up



```
Dockerfile x
1 # Start with a base image
2 FROM ubuntu:18.04
3
4 # Add file author/maintainer and contact info (optional)
5 MAINTAINER Louise Cabansay <lcabansa@ucsc.edu>
6
7 # install software dependencies
8 RUN apt update && apt -y upgrade && apt install -y \
9     wget build-essential libncurses5-dev zlib1g-dev \
10    libbz2-dev liblzma-dev libcurl3-dev \
11
12 WORKDIR /usr/src
13
14 # get the software source files
15 RUN wget https://github.com/samtools/samtools/releases/
16 download/1.10/samtools-1.10.tar.bz2
17
18
19 # installation commands to compile source files
20 tar xjf samtools-1.10.tar.bz2 && \
21 rm samtools-1.10.tar.bz2 && \
22 cd samtools-1.10 && \
23 ./configure --prefix $(pwd) && \
24 make
25
26 # add newly built executables to path
27 ENV PATH="/usr/src/samtools-1.10:${PATH}"
28
29
30
31
```

Sharing your Dockerfiles and Images



Dockstore recommends storing your Dockerfile in an external repository (Bitbucket, GitHub, GitLab) and then registering your source controlled Dockerfile to an image registry (Docker Hub, Quay.io, Google Container Registry, etc)

Best Practices (Take home reading)

- Start from official images (https://docs.docker.com/docker-hub/official_images/)
- Only add what you need (keep containers light),
 - One way to do this is to use multiple containers in your workflows
 - Recommend not including large reference data within containers
 - Instead provide that data at runtime through data binding or via platform
- Version your containers for re-use later (use --tag when building)

What Next?

Docker is great, it tells us how to install software.

However, it doesn't tell us how to use software.

Descriptor languages are the solution!

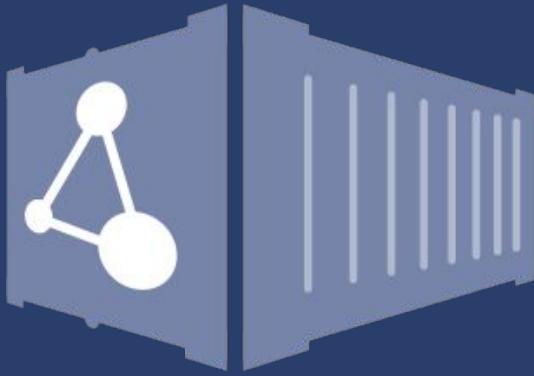
Break

What Next?

Docker is great, it tells us how to install software.

However, it doesn't tell us how to use software.

Descriptor languages are the solution!



Intro to Descriptors (WDL)

Components and Concepts shared by Descriptors

Container:

Packaged up code with all of its dependencies. This allows for portable software that runs quickly and reliably from one computing environment to another.



Descriptor:

A workflow language used to describe how to run your pipeline.

- Which containers
- What steps and when
- Define parameters
 - I/O data
 - compute requirements
- Metadata



nextflow

Parameter File (wdl, cwl):

- Specifies the actual input/output files (local, ftp, http, or cloud)
- Set compute resources
- JSON, YAML

```
{  
    "bam_input": {  
        "class": "File",  
        "format": "http://edamontology.org/format_2572",  
        "path": "/tmp/NA12878.chrom20.ILLUMINA.bwa.CEU.low_coverage.20121211.bam"  
    },  
    "bamstats_report": {  
        "class": "File",  
        "path": "/tmp/bamstats_report.zip"  
    }  
}
```

CWL: Common Workflow Language

CWL: Common Workflow Language

- Open and portable standard for describing analysis workflows and tools
- Alternative to WDL/nextflow that might be present in your lab/cloud
- [Upcoming tutorial example but in CWL](#)

Implementations/Engines:

- CWL doesn't have a single official engine
- CWLtool - reference implementation
- Other implementations include Arvados, Toil, and Cromwell

Analysis Platforms (Launch-with)

- Seven Bridges Cancer Genomics Cloud (CGC)



Nextflow

Nextflow

- Fluent domain-specific language also for scientific workflows using software containers
- View as alternative to CWL/WDL that might be present in your lab/cloud
- [Upcoming tutorial example but in Nextflow](#)

The logo for Nextflow, featuring the word "nextflow" in a bold, lowercase sans-serif font. The letter "e" is stylized with a green swoosh that extends from the top of the "n" over to the "f".

Running nextflow workflows

- Works on local machines, HPC, AWS, Google Cloud
- Cloud support via [Sequera Labs](#)

WDL: Workflow Description Language

WDL: Workflow Description Language

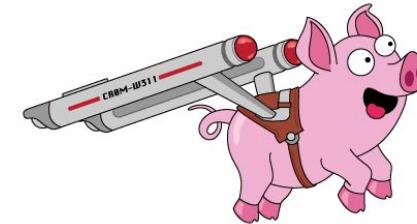
- human-readable and writable descriptor language

Engines:

- Cromwell: “A Workflow Management System geared towards scientific workflows”
 - The first execution engine that understands WDL
- Other engines: TOIL, miniWDL

Analysis Platforms (Launch-with)

- Terra (AnVIL, BioDataCatalyst)
- DNAstack
- DNAnexus
- Also works on local machines, HPC, and Cloud



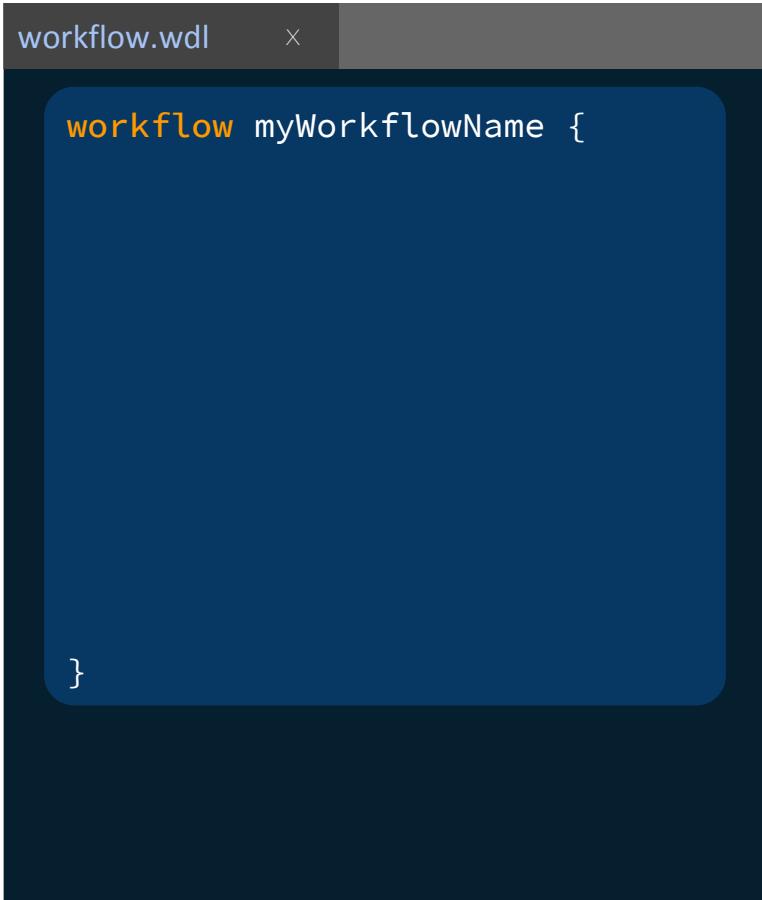
What's in a WDL? Top-level Components

workflow.wdl X

3 top-level components that are part of the core structure of a WDL script

- Workflow
- Call
- Task

Top-level Components - Workflow



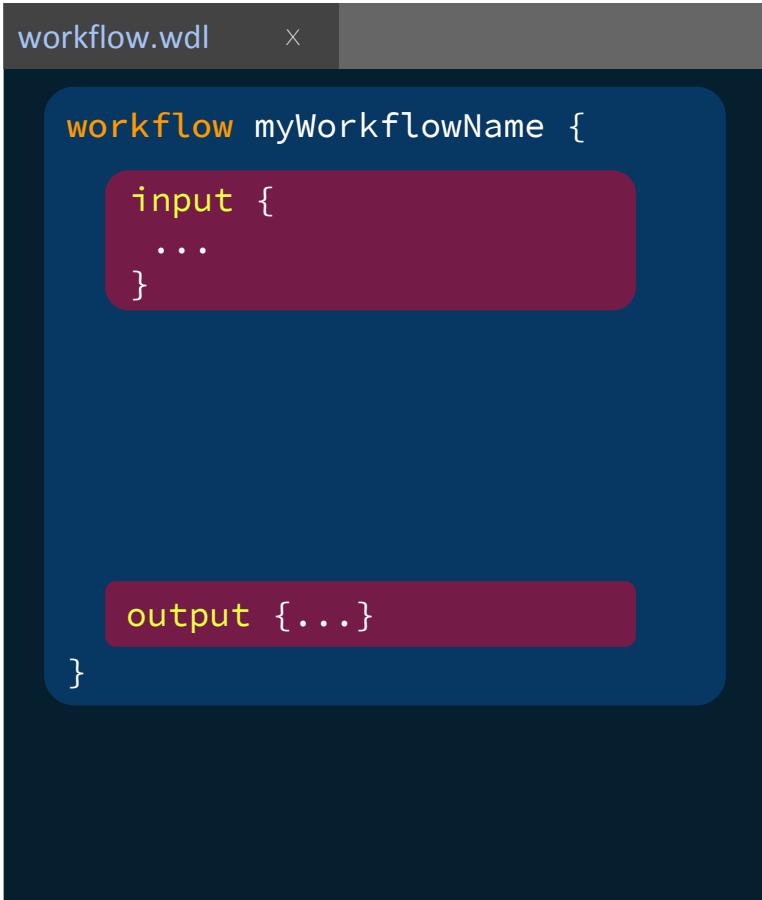
A screenshot of a code editor window titled "workflow.wdl". The code block contains a single line of WDL syntax:

```
workflow myWorkflowName { }
```

The word "workflow" is highlighted in orange, indicating it is a keyword. The code is displayed in a dark-themed editor.

Workflow: Code block that defines the overall workflow. You can think of it as an outline.

Top-level Components - Workflow



A screenshot of a code editor window titled "workflow.wdl". The code is a WDL workflow definition:

```
workflow myWorkflowName {
    input {
        ...
    }

    output {...}
}
```

The "input" block and the "output" block are highlighted with a red background.

Workflow: Code block that defines the overall workflow. You can think of it as an outline.

- **Inputs (optional)**
 - Ex: use when building more complex workflows that will re-use inputs for multiple purposes
- **Outputs**
 - Specify output you want to keep from run of entire workflow
 - Signals to Cromwell to keep track of these outputs and save them somewhere

Top-level Components - Call

```
workflow.wdl      x  
  
workflow myWorkflowName {  
    input {  
        ...  
    }  
  
    call task_A  
  
    call task_B {  
        input: ...  
    }  
  
    output {...}  
}
```

Call: Component that defines which tasks the workflow will run

- Located within workflow block
- can also specify input parameters to pass to that task (optional)

Top-level Components - Task

```
workflow.wdl      x  
  
workflow myWorkflowName {  
    input {  
        ...  
    }  
  
    call task_A  
  
    call task_B {  
        input: ...  
    }  
  
    output {...}  
}  
  
task task_A { ... }  
  
task task_B { ... }
```

Task: Defines all the information necessary to perform an action.

- Tasks are referred to within a call, but are actually defined *outside* of the workflow block

What's in a Task?

Task: Defines all the information necessary to perform an action



A screenshot of a code editor window titled "task.wdl". The editor displays a single line of WDL code: "task doSomething {". A large, semi-transparent gray rectangle covers the majority of the editor area, starting from the opening brace of the task definition and extending downwards.

```
task.wdl x
task doSomething {
```

What's in a Task? Command

Task: Defines all the information necessary to perform an action

- Command ('the action') - **required!**
 - Defines the command(s) that will be run in the execution environment
 - Can be multiple lines/commands



A screenshot of a code editor window titled "task.wdl". The code defines a task named "doSomething" which contains a command block. The command block runs "echo Hello World!" and "cat \${myName}".

```
task.wdl    x
task doSomething {
    command {
        echo Hello World!
        cat ${myName}
    }
}
```

What's in a Task? Inputs

Task: Defines all the information necessary to perform an action

- Inputs
 - Optional, only required if the task will have inputs
 - All inputs must be typed
 - string, int, file, etc
 - Individual inputs can also be optional, denoted by '?':
`input { File? myName }`
 - Can also set a default values:
`input { String? myName="Foobar" }`



The screenshot shows a code editor window titled "task.wdl". The code defines a task named "doSomething" with the following structure:

```
task doSomething {
    input { File myName }

    command {
        echo Hello World!
        cat ${myName}
    }
}
```

What's in a Task? Outputs

Task: Defines all the information necessary to perform an action

- Outputs
 - The outputs section defines which values should be exposed as outputs after a successful run of the task
 - Especially useful when using output of one task as input of another
 - Outputs must be typed
 - ex: string, int, file, etc

The screenshot shows a code editor window titled "task.wdl". The code is written in WDL (Workflow Description Language) and defines a task named "doSomething". The task has an "input" section with a single file named "myName". It contains a "command" block that runs two shell commands: "echo Hello World! > Hello.txt" and "cat \${myName} >> Hello.txt". The task also has an "output" section defining a file named "outFile" with the value "Hello.txt". The code is presented in three blue-highlighted sections corresponding to the WDL blocks.

```
task doSomething {
    input { File myName }

    command {
        echo Hello World! > Hello.txt
        cat ${myName} >> Hello.txt
    }

    output {
        File outFile = "Hello.txt"
    }
}
```

Simple Example: HelloWorld.wdl

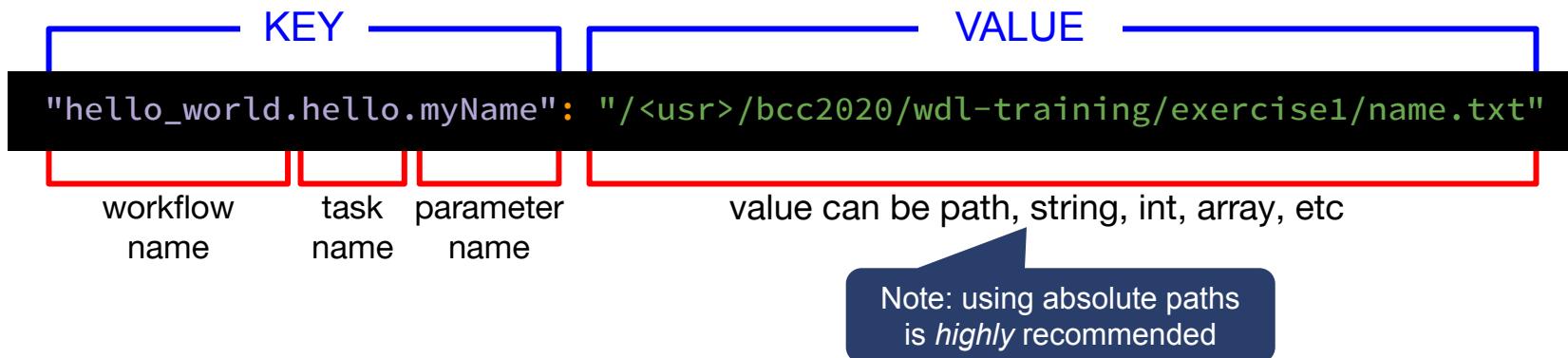
```
HelloWorld.wdl  x

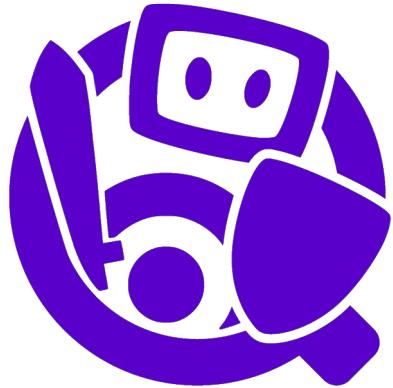
1 version 1.0
2 # add and name a workflow block
3 workflow hello_world {
4     call hello
5     # important: add output for whole workflow
6     output {
7         File helloFile = hello.outFile
8     }
9 }
10
11 # define the 'hello' task
12 task hello {
13     input { File myName }
14     command {
15         echo Hello World! > Hello.txt
16         cat ${myName} >> Hello.txt
17     }
18     output { File outFile = "Hello.txt" }
19 }
20 }
```

Parameter JSON (simple):

- A parameter JSON specifies the actual input values to fill-in for the WDL parameters when running the workflow

```
hello.json    x
1 {
2   "hello_world.hello.myName": "/<usr>/bcc2020/wdl-training/exercise1/name.txt"
3 }
```





instruqt



Exercise #1: Run your first wdl

- Single task workflow
 - Workflow block not required

```
HelloWorld.wdl ×  
1 version 1.0  
2 workflow hello_world {  
3     call hello  
4     output { File helloFile = hello.outFile }  
5 }  
6  
7 task hello {  
8     input { File myName }  
9     command {  
10        echo Hello World! > Hello.txt  
11        cat ${myName} >> Hello.txt  
12    }  
13    output { File outFile = "Hello.txt" }  
14 }  
15 }
```

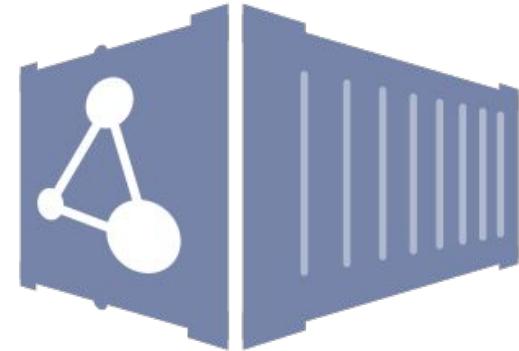
Run with DockstoreCLI

```
dockstore workflow launch --local-entry HelloWorld.wdl --json hello.json
```

Overview: Dockstore CLI

A handy command line resource to help users develop content locally.

- Run descriptors by automatically calling to Cromwell or CWLtool
 - Local descriptors
 - Remote descriptors pulled from Dockstore
- Generate a JSON parameter template based on a given descriptor
- Built-in plugins enable fetching remote input data (http, s3, gs)



Example execution with the Dockstore Command Line Interface (CLI):

```
dockstore workflow launch --local-entry HelloWorld.wdl --json hello.json
```

What's in a Task? Runtime

Task: Defines all the information necessary to perform an action

- Runtime
 - Defines context/environment
 - docker containers
 - compute resources
 - For cloud platforms, runtime parameters can be used to allocate/configure resources when spinning up a compute instance
 - CPU
 - Disk
 - Memory
 - Instance types
 - Region
 - Preemptibility

The screenshot shows a code editor window titled "task.wdl". The code is written in WDL (Workflow Description Language) and defines a task named "doSomething". The task has an input parameter "myName" of type File, a command to echo "Hello World!" to a file and append the value of "myName" to it, an output file "outFile" set to "Hello.txt", and a runtime configuration specifying "ubuntu:latest" as the Docker image and "1GB" of memory.

```
task doSomething {
    input { File myName }

    command {
        echo Hello World! > Hello.txt
        cat ${myName} >> Hello.txt
    }

    output {
        File outFile = "Hello.txt"
    }

    runtime {
        docker: "ubuntu:latest"
        memory: "1GB"
    }
}
```

What's in a Task? Parameterization

Task: Defines all the information necessary to perform an action

- Parameterized values
 - Variables that define placeholder requirements
 - flexibility, reuse, repurposing
 - ***Pretty much everything can be parameterized***
 - Commands
 - Inputs, outputs
 - Runtime requirements

But is this always a best practice?

The screenshot shows a code editor window titled "task.wdl". The code is written in WDL (Workflow Description Language) and defines a task named "doSomething". The task has four main sections: input, command, output, and runtime.

```
task doSomething {
    input {
        File myName
        String outFile
    }

    command {
        echo Hello World! > ${outFile}
        cat ${myName} >> ${outFile}
    }

    output {
        File outFile = "${outFile}"
    }

    runtime {
        docker: docker_image
        memory: "${memory_gb}"
    }
}
```

Non-Input Declarations

A task can have declarations which are intermediate values rather than inputs.

- Think of these as variables that you can define to help execute a task
- Typed: String, Int, File, etc
- Can use or build upon:
 - the input values given to task
 - methods from WDL standard library
 - other non-input declarations
- After defining, used in other sections: command, outputs, runtime

task.wdl x

```
task doSomething {
```

```
    input { String myName }
```

```
#creating non-input declaration
String myString = "hi " + ${myName}
String outFile = ${myName} + ".out"
```

```
# example usage in command
command {
    echo ${myString} > ${outFile}
}
```

```
# example usage in output
output {
    File outFile = "${outFile}"
}
```

WDL Standard Library (Take Home)

Built-in functions or methods provided by the core WDL language

- Helpful in writing more complex workflows
- Examples:
 - File handling (read, write different kinds files: JSON, tsv, etc)
 - Working with stdin, stdout
 - Data manipulation/handling: arrays, objects, strings, etc
 - Mapping values
 - Much more!

WDL Standard Library (Simple)

```
task.wdl    x

task hello {
    input { File myName }

    command {
        echo Hello World! > Hello.txt
        cat ${myName} >> Hello.txt
    }

    output {
        File outFile = "Hello.txt"
    }
}
```

Output:
“hello.outFile” : “.../Hello.txt”

```
task.wdl    x

task hello {
    input { File myName }

    command {
        echo Hello World!
        cat ${myName}
    }

    output {
        File outFile = stdout()
    }
}
```

Output:
“hello.outFile” : “.../stdout”

Primer Exercise #2

For our second exercise we're going to **parameterize** a simple workflow.

Goal: generate statistics about an alignment file

- Software: samtools (flagstat)
- Input: sam file
- Output: alignment statistics

There will be multiple ways to solve this assignment. This is a chance for you to apply the things we've learned to a real bioinformatics workflow.



Exercise #2: Complete metrics.wdl

1. Set runtime to use the samtools docker container:
quay.io/lcabansay/samtools:latest
2. Parameterize the samtools command in the flagstat task
3. (optional) If you make any new inputs, be sure update the metrics.json

```
runtime {  
    docker: "ubuntu:latest"  
    memory: "1GB"  
}
```

```
metrics.wdl x  
1 version 1.0  
2  
3 workflow metrics {  
4     call flagstat  
5     output { File align_metrics = flagstat.metrics }  
6 }  
7  
8 task flagstat {  
9     input { File input_sam }  
10  
11     # non-parameterized flagstat command  
12     command {  
13         samtools flagstat mini.sam > mini.sam.metrics  
14     }  
15     output {  
16         File metrics = "mini.sam.metrics"  
17     }  
18  
19     # set some parameterized runtime parameters  
20     runtime {  
21         docker: # set  
22     }  
23 }  
24 }
```

Exercise #2: Solution* - Descriptors

*note: this is one example solution, multiple are possible

metrics.wdl

```
1 version 1.0
2
3 workflow metrics {
4     call Flagstat
5     output { File align_metrics = flagstat.metrics }
6 }
7
8 task flagstat {
9     input { File input_sam }
10
11    # non-parameterized flagstat command
12    command {
13        samtools flagstat mini.sam > mini.sam.metrics
14    }
15
16    output {
17        File metrics = "mini.sam.metrics"
18    }
19
20    # set some parameterized runtime parameters
21    runtime {
22        docker: # set
23    }
24 }
```

metrics.wdl

```
1 version 1.0
2
3 workflow metrics {
4     call Flagstat
5     output { File align_metrics = flagstat.metrics }
6 }
7
8 task flagstat {
9     input { File input_sam }
10
11    # slightly parameterized flagstat command
12    command {
13        samtools flagstat ${input_sam} > mini.sam.metrics
14    }
15
16    output { File metrics = "mini.sam.metrics" }
17
18    # set docker runtime
19    runtime {
20        docker: "quay.io/ldcabansay/samtools:latest"
21    }
22 }
23
24 }
```

Exercise #2: Solution vs Solution2

*note: this is one example solution, multiple are possible

metrics.wdl

```
1 version 1.0
2
3 workflow metrics {
4     call Flagstat
5     output { File align_metrics = flagstat.metrics }
6 }
7
8 task flagstat {
9     input { File input_sam }
10
11    # slightly parameterized flagstat command
12    command {
13        samtools flagstat ${input_sam} > mini.sam.metrics
14    }
15
16    output { File metrics = "mini.sam.metrics" }
17
18    # set docker runtime
19    runtime {
20        "quay.io/ldcabansay/samtools:latest"
21    }
22}
23
24}
```

metrics.wdl

```
1 version 1.0
2
3 workflow metrics {
4     call Flagstat
5     output { File align_metrics = flagstat.metrics }
6 }
7
8 task flagstat {
9     input {
10        File input_sam
11        String docker_image
12    }
13    # create a string to help parameterize command
14    String stats = basename(input_sam) + ".metrics"
15
16    command {
17        samtools flagstat ${input_sam} > ${stats}
18    }
19    output { File metrics = "${stats}" }
20
21    # set a parameterized docker runtime
22    runtime {
23        docker: docker_image
24    }
25}
```

Break

Multi-task workflows:

- In real use cases, workflows are typically multi-task pipelines that build upon individual steps
 - The output of one task can serve as the input of another
- Each task in workflow can have their own isolated runtime environment
 - Docker image
 - Task specific compute requirements

Example: Multi-task workflow

HelloWorld.wdl

```
1 version 1.0
2 workflow hello_world {
3     call hello
4         output { File helloFile = hello.outFile }
5 }
6
7 task hello {
8     input { File myName }
9
10    command {
11        echo Hello World!
12        cat ${myName}
13    }
14    output { File outFile = stdout() }
15 }
```

GoodbyeWorld.wdl

```
1 version 1.0
2 workflow goodbye_world {
3     call goodbye
4         output { File byeFile = goodbye.outFile }
5 }
6
7 task goodbye {
8     input { File greeting }
9
10    command {
11        cat ${greeting}
12        echo See you later!
13    }
14    output { File outFile = stdout() }
15 }
```

Example: Multi-task workflow - HelloGoodbye.wdl

HelloGoodbye.wdl ×

```
1 version 1.0
2 workflow HelloGoodbye {
3     call hello
4     call goodbye {
5         input: greeting = hello.outFile
6     }
7     output { File hello_goodbye = goodbye.outFile }
8 }
9 task hello {
10     input { File myName }
11     command {
12         echo Hello World!
13         cat ${myName}
14     }
15     output { File outFile = stdout() }
16 }
17 task goodbye {
18     input { File greeting }
19     command {
20         cat ${greeting}
21         echo See you later!
22     }
23     output { File outFile = stdout() }
24 }
```

HelloGoodbye.json ×

```
1 [
2 "HelloGoodbye.hello.myName": 
3 "/root/bcc2020-training/wdl-training/exercise3/
4 hello_examples/name.txt"
5 ]
```

WDL Imports

A WDL file may contain import statements to include WDL code from other sources.

- Useful in keeping large multi-task workflows organized
- Helpful way to reuse and build modular workflows

Imports: Concepts

- Primary Descriptor
 - The ‘main’ descriptor file, anchors relevant imports
- Sub-workflow aka sub-wdl
 - the imported, external WDL
- Namespaces
 - prevents name collisions, organizes tasks/workflows into groups
- Aliases
 - A custom name given to a namespace (optional)
 - Not specifying an alias defaults to namespace to filename without ‘.wdl’

```
primary-descriptor.wdl  ×  
  
import "<resource>" as <alias>  
  
workflow primary {  
    ...  
  
    call task_A  
  
    call <alias>.taskOne {  
        input: ...  
    }  
    ...  
}  
  
task task_A { ... }
```

JSON mapping:

```
"primary.taskOne.param_name": "<value of param or path if file>"
```



Example: No Imports vs Imports

HelloGoodbye.wdl ×

```
1 version 1.0
2 workflow HelloGoodbye {
3     call hello
4     call goodbye {
5         input: greeting = hello.outFile
6     }
7     output { File hello_goodbye = goodbye.outFile }
8 }
9 task hello {
10     input { File myName }
11     command {
12         echo Hello World!
13         cat ${myName}
14     }
15     output { File outFile = stdout() }
16 }
17 task goodbye {
18     input { File greeting }
19     command {
20         cat ${greeting}
21         echo See you later!
22     }
23     output { File outFile = stdout() }
24 }
```

HelloGoodbye_imports.wdl ×

```
1 version 1.0
2 # add import statements to bring in sub-workflows
3 # if not given, namespace/alias = file minus '.wdl'
4 import "HelloWorld.wdl"
5
6 # otherwise, namespace = <alias>
7 import "GoodbyeWorld.wdl" as bye
8
9 workflow HelloGoodbye {
10
11     #call the hello task, syntax: <alias>.taskname
12     call HelloWorld.hello
13
14     #call the goodbye task, syntax: <alias>.taskname
15     call bye.goodbye {
16         input: greeting = hello.outFile
17     }
18     #same as before, define workflow outputs
19     output { File hello_goodbye = goodbye.outFile }
20 }
21
22
23
24 }
```

Example: No Imports vs Imports

HelloGoodbye.wdl ×

```
1 version 1.0
2 workflow HelloGoodbye {
3     call hello
4     call goodbye {
5         input: greeting = hello.outFile
6     }
7     output { File hello_goodbye = goodbye.outFile }
8 }
9 task hello {
10     input { File myName }
11     command {
12         echo Hello World!
13         cat ${myName}
14     }
15     output { File outFile = stdout() }
16 }
17 task goodbye {
18     input { File greeting }
19     command {
20         cat ${greeting}
21         echo See you later!
22     }
23     output { File outFile = stdout() }
24 }
```

HelloGoodbye_imports.wdl ×

```
1 version 1.0
2 # add import statements to bring in sub-workflows
3 # if not given, namespace/alias = file minus '.wdl'
4 import "HelloWorld.wdl"
5
6 # otherwise, namespace = <alias>
7 import "GoodbyeWorld.wdl" as bye
8
9 workflow HelloGoodbye {
10
11     #call the hello task, syntax: <alias>.taskname
12     call HelloWorld.hello
13
14     #call the goodbye task, syntax: <alias>.taskname
15     call bye.goodbye {
16         input: greeting = hello.outFile
17     }
18     #same as before, define workflow outputs
19     output { File hello_goodbye = goodbye.outFile }
20 }
21
22
23
24 }
```

Do we have to
change the JSON
when running
HelloGoodbye
using imports?

Example: No Imports vs Imports (no comments)

HelloGoodbye.wdl ×

```
1 version 1.0
2 workflow HelloGoodbye {
3     call hello
4     call goodbye {
5         input: greeting = hello.outFile
6     }
7     output { File hello_goodbye = goodbye.outFile }
8 }
9 task hello {
10    input { File myName }
11    command {
12        echo Hello World!
13        cat ${myName}
14    }
15    output { File outFile = stdout() }
16 }
17 task goodbye {
18    input { File greeting }
19    command {
20        cat ${greeting}
21        echo See you later!
22    }
23    output { File outFile = stdout() }
24 }
```

HelloGoodbye_imports.wdl ×

```
1 version 1.0
2 import "HelloWorld.wdl"
3 import "GoodbyeWorld.wdl" as bye
4
5 workflow HelloGoodbye {
6     call HelloWorld.hello
7     call bye.goodbye {
8         input: greeting = hello.outFile
9     }
10    output { File hello_goodbye = goodbye.outFile }
11 }
12
13
14
15
16
17
18
19
20
21
22
23
24 }
```

Primer for Exercise #3: (if time permits)

- Aligner (bwa)
 - Align a sequence files (FASTQs) to a reference
 - Produces an alignment file: sam
- Metrics (samtools flagstat)
 - Evaluates an alignment file (sam or bam)
 - Reports statistics on about the alignment
- What we'll make:
 - A workflow that does both of these tasks, first aligns, then generates statistics about the alignment
 - Without imports
 - With imports

Ex: BWA Aligner

- Specify WDL version
- Define workflow, call, and task(s)
- Define parameters
 - Input and output
 - Parameterized command(s)
 - Runtime environment
 - Compute resource requirements
- Metadata
 - Authorship, contact information, etc

1 version 1.0
2 workflow alignReads {
3 call bwa_align
4 output { File output_sam = bwa_align.output_sam }
5 }
6 task bwa_align {
7 input {
8 String sample_name
9 String docker_image
10 String? bwa_options
11 File read1_fastq
12 File read2_fastq
13 File ref_fasta
14 File ref_fasta_fai
15 File ref_fasta_amb
16 File ref_fasta_ann
17 File ref_fasta_bwt
18 File ref_fasta_pac
19 File ref_fasta_sa
20 }
21 String output_sam = "\${sample_name}" + ".sam"
22
23 command {
24 bwa mem \${bwa_options} \${ref_fasta} \
25 \${read1_fastq} \${read2+fastq} > \${output_sam}
26 }
27 output { File output_sam = "\${output_sam}" }
28 runtime {
29 docker: docker_image
30 memory: "\${memory_gb}" + "GB"
31 }
32 meta {
33 author: "Foo Bar"
34 email: "foobar@university.edu"
35 }
36 }
37 }

Example: Metrics.wdl (samtools flagstat)

Same as the solution to
exercise #2

```
metrics.wdl
1 version 1.0
2
3 workflow metrics {
4     call Flagstat
5     output { File align_metrics = Flagstat.metrics }
6 }
7
8 task Flagstat {
9     input {
10     File input_sam
11     String docker_image
12 }
13 # create a string to help parameterize command
14 String stats = basename(input_sam) + ".metrics"
15 command {
16     samtools flagstat ${input_sam} > ${stats}
17 }
18 output { File metrics = "${stats}" }
19
20 # set some parameterized runtime parameters
21 runtime {
22     docker: docker_image
23 }
24 }
```



Exercise#3: Writing a multi-task workflow (if time permits)

Create a multi-task workflow: align_and_metrics.wdl

- aligner.wdl
 - Generates a sam file from given fastq sequence files and reference files
- metrics.wdl
 - Generates alignment statistics of a given sam/bam file.
- You may create align_and_metrics.wdl with or without imports. A skeleton is provided to you for each type.

Importing workflows: Best Practices & Tips (Take home)

- Each import should perform a specific action and be runnable on its own (given the correct input)
- Use descriptive aliases for namespaces
- Use descriptive names for tasks and workflows
- Imports can be local file paths or HTTP(s) paths
 - HTTP(s) example: grabbing a raw file from a repository like GitHub

Importing workflows: Caveats (Take home)

- Not all workflow engines or platforms support imports (ex. DNAnexus)
- Levels of support also varies, some features don't work
 - Ex. Terra only recently started supporting local file path imports and only if the descriptor is in GitHub
- Learn more about which platforms support imports:
<https://docs.dockstore.org/en/develop/end-user-topics/language-support.html>

Metadata and Parameter Metadata (Take Home)

Metadata

- **meta** section within the workflow section
- Optional key/value pairs
- Useful for author, email, description

Parameter Metadata

- **parameter_meta** section within a task
- Optional key/value pairs
- Describe parameters
- Key **must** map to a task input or output

The screenshot shows a code editor window titled "task.wdl". The code is written in WDL (Workflow Description Language) and defines a task named "doSomething". The task has an input parameter "myName" of type File. It contains a command block that echoes "Hello World!" to a file named "Hello.txt" and then appends the value of "myName" to the same file. The task also has an output parameter "outFile" of type File, which is set to "Hello.txt". The entire task definition is enclosed in a brace at the bottom.

```
task.wdl x

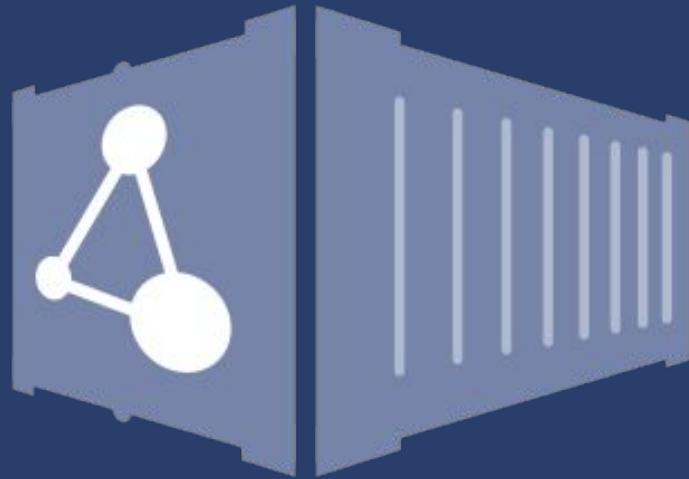
task doSomething {
    input { File myName }

    command {
        echo Hello World! > Hello.txt
        cat ${myName} >> Hello.txt
    }

    output {
        File outFile = "Hello.txt"
    }
}
```

More trainings and tutorial content:

- docs.dockstore.org
 - [Getting started with CWL](#)
 - [Getting started with WDL](#)
 - [Getting started with Nextflow](#)
- As provided by each language
 - [Common Workflow Language User Guide](#)
 - [learn-wdl](#) - educational materials for learning WDL from [openwdl](#)
 - [WDL documentation](#) and tutorials from [Terra](#)
 - [Get started — Nextflow 20.04.1 documentation](#)
 - [Nexflow-tutorial](#) from Seqera Labs



Summary, Dockstore, and next steps!

What is Dockstore?

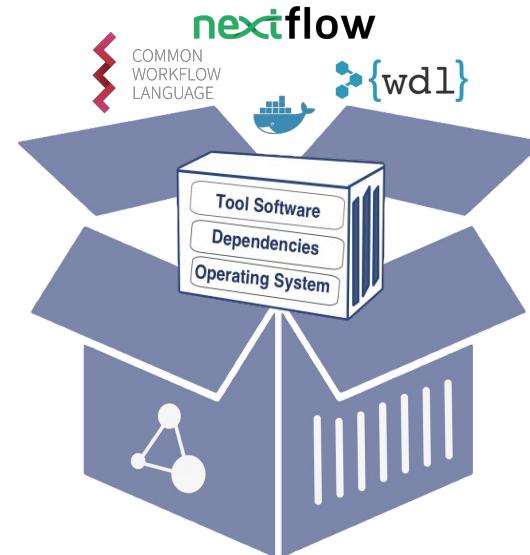
Dockstore is a free and open source platform for sharing scientific tools and workflows.

Portability

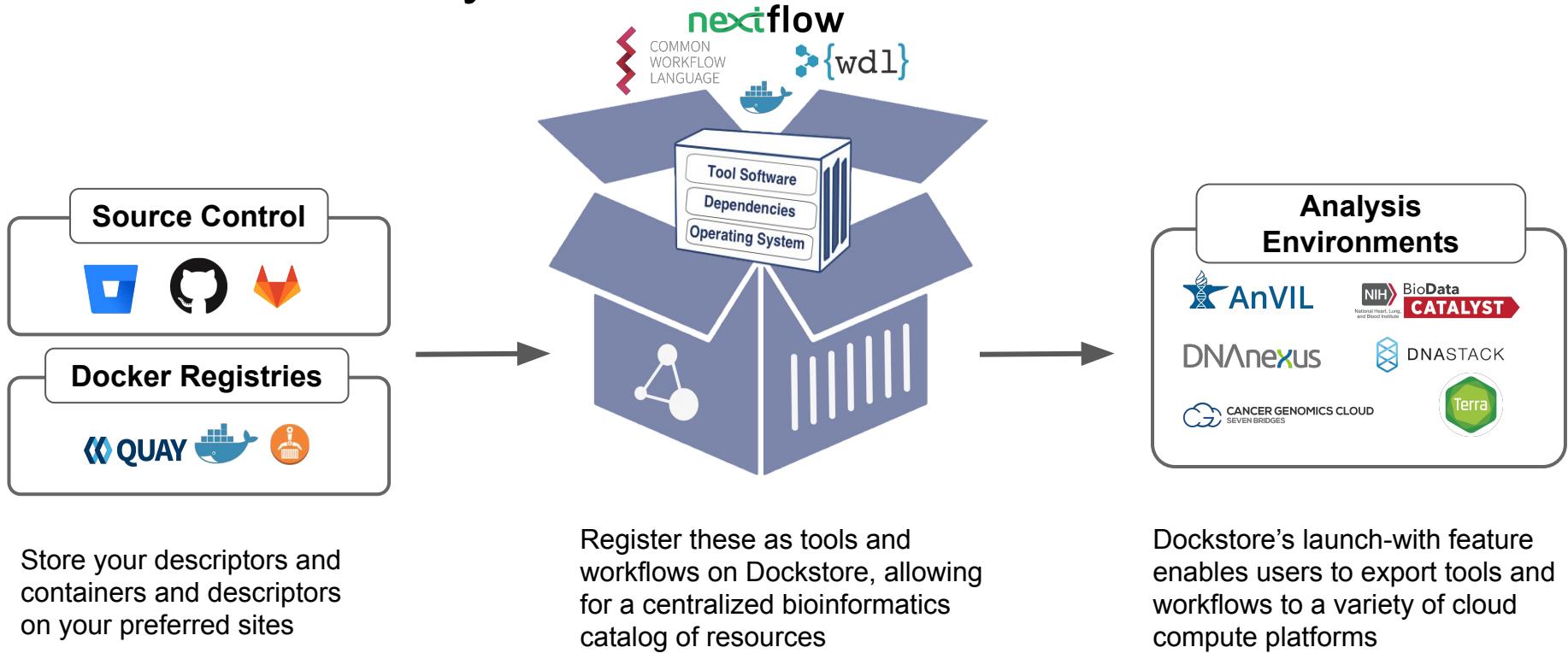
Interoperability

Reproducibility

“An app store for bioinformatics”



Dockstore Ecosystem



Launching Analysis

The screenshot shows the BioData Catalyst platform interface. On the left, a workflow descriptor file named `vg_map_call_sv.wdl` is displayed, showing its version 1.0 and configuration details. On the right, a 'Launch with' panel lists several partner platforms: DNAstack, DNAexus, Terra, CGC (Cloud), AnVIL, and NHLBI BioData Catalyst. The 'DNAstack' button is highlighted.

```
github.com/vgteam/vg_wdl/vg_map_call_sv:svpack
Last Modified: 169 days ago
genome-graph genomics genotyping structural-variant sv variation-graph

Descriptor Files Test Parameter Files
vg_map_call_sv.wdl

version 1.0
workflow vgMapCallsV {
    meta {
        author: "Jean Monlong"
        email: "jmonlong@ucsc.edu"
        description: "Read mapping and SV genotyping using vg. It takes as input sample 1st and 2nd read pairs, a CRAM file, a genome FASTA file, and an index of the FASTA file associated with it. It outputs a VCF file, a GCSA index file, a GCSA.lcp index file, a snarl index file, a GBWT index file, and a PATH.LIST file. The number of reads contained in each read pair is specified by the READS_PER_CHUNK parameter. The number of cores used for CRAM conversion is specified by the CRAM_CONVERT_CORES parameter. The number of cores used for splitting reads is specified by the SPLIT_READ_CORES parameter. The number of cores used for mapping is specified by the MAP_CORES parameter. The number of memory cores used for merging GAM files is specified by the MERGE_GAM_CORES parameter. The number of disk cores used for merging GAM files is specified by the MERGE_GAM_DISK parameter. The number of memory cores used for merging GAM files is specified by the MERGE_GAM_MEM parameter. The time taken for merging GAM files is specified by the MERGE_GAM_TIME parameter. The number of cores used for VGCALL is specified by the VGCALL_CORES parameter." }
    input {
        String SAMPLE_NAME           # The sample name
        File? INPUT_READ_FILE_1      # Input sample 1st read pair
        File? INPUT_READ_FILE_2      # Input sample 2nd read pair
        File? INPUT_CRAM_FILE        # Input CRAM file
        File? CRAM_REF               # Genome fasta file associated with the CRAM file
        File? CRAM_REF_INDEX         # Index of the fasta file associated with the CRAM file
        String VG_CONTAINER = "quay.io/vgteam/vg:v1.19.0" # VG Container used for running the workflow
        File XG_FILE                 # Path to .xg index file
        File GCSA_INDEX_FILE         # Path to .gcsa index file
        File GCSA_LCP_INDEX_FILE     # Path to .gcsa.lcp index file
        File? SNARL_INDEX_FILE       # (OPTIONAL) Path to snarl index file
        File? GBWT_INDEX_FILE        # (OPTIONAL) Path to gbwt index file
        File? PATH_LIST_FILE         # (OPTIONAL) Text file where each line contains a path to a BAM file
        Int READS_PER_CHUNK = 20000000
        Int CRAM_CONVERT_CORES = 16
        Int CRAM_CONVERT_DISK = 200
        Int SPLIT_READ_CORES = 32
        Int SPLIT_READ_DISK = 200
        Int MAP_CORES = 32
        Int MAP_DISK = 100
        Int MAP_MEM = 100
        Int MERGE_GAM_CORES = 64
        Int MERGE_GAM_DISK = 400
        Int MERGE_GAM_MEM = 100
        Int MERGE_GAM_TIME = 1200
        Int VGCALL_CORES = 10
    }
}
```

Partner Platforms

- DNAstack
- DNAexus
- Terra
- CGC : Cancer Genomics Cloud (Seven Bridges)
- AnVIL
- BioData Catalyst

Structural Variant Calling using Graph Genomes

Contributed by: Jean Monlong and Charles Markello
(VG Team, UC Santa Cruz, Genomics Institute)

Launching Analysis - Example



The screenshot displays the AnVIL Import Workflow interface. At the top, there's a header with a menu icon, the AnVIL logo, and a "IMPORT WORKFLOW" button. Below the header, on the left, is a panel titled "Importing from Dockstore" showing the GitHub repository "github.com/klarman-cell-observatory/cumulus/Cumulus" and version "V 0.15.0". It includes a warning message about Dockstore guarantees and a note to review the WDL. On the right, there's a configuration panel with fields for "Workflow Name" (set to "Cumulus") and "Destination Workspace" (set to "Dockstore-Webinar"). A large "IMPORT" button is at the bottom of this panel. The background features a light gray hexagonal grid pattern.

Cumulus Workflow: <https://dockstore.org/workflows/github.com/klarman-cell-observatory/cumulus/Cumulus:0.15.0?tab=info>

AnVIL Organization, Cumulus Collection: <https://dockstore.org/organizations/anvil/collections/Cumulus>

Contributed by: Bo Li & Yiming Yang (Cumulus Team, Broad Institute)

General Best Practices

- Include authorship and contact information in the primary descriptor file
- Include workflow description either in the README or descriptor file
- Use releases/tags (ex. v1.1) instead of branches for versioning
- Test parameter files associated with workflows to provide samples that users can try out
 - Use publicly available data
 - Real world examples or simple test data
- Add labels to your workflow to improve findability
- Checker workflows to test compatibility with different environments
- Use the snapshot and DOI feature to improve reproducibility

DOIs

Create snapshots and digital object identifiers for your workflows to permanently capture the state of a workflow for publication

[Creating Snapshots and Requesting DOIs — Dockstore documentation](#)

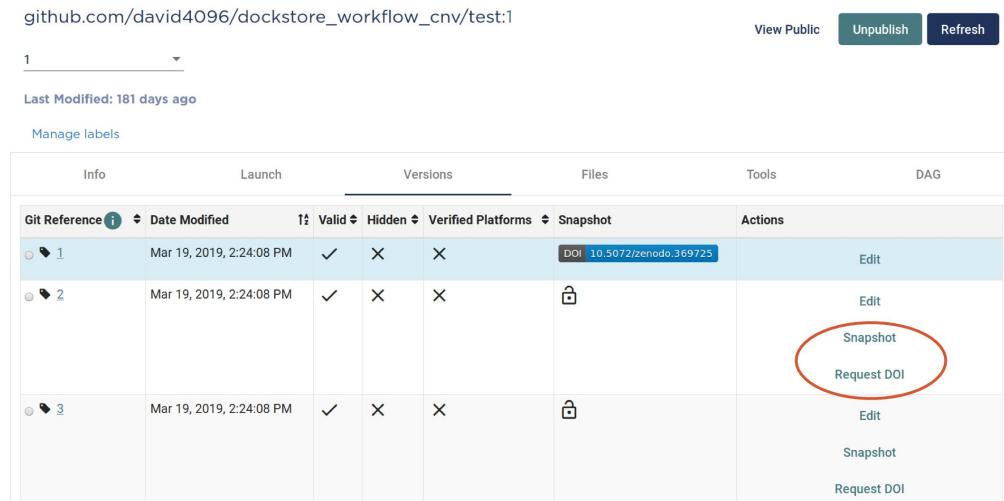
Examples:

Forward

<https://doi.org/10.5281/zenodo.3889018>

Backward

https://dockstore.org/workflows/github.com/dockstore/hello_world:master?tab=versions



The screenshot shows a Dockstore interface for a workflow named "dockstore_workflow_cnv/test:1". At the top right are buttons for "View Public", "Unpublish", and "Refresh". Below the URL is a dropdown menu set to "1". A note says "Last Modified: 181 days ago". There's a "Manage labels" link. A table lists three workflow versions. The first version has a DOI link (10.5072/zenodo.369725) and edit, snapshot, and request DOI buttons. The second version has an edit button and snapshot, request DOI buttons. The third version has an edit button and snapshot, request DOI buttons. A red oval highlights the "Snapshot" and "Request DOI" buttons for the second version.

Git Reference	Date Modified	Valid	Hidden	Verified Platforms	Snapshot	Actions
1	Mar 19, 2019, 2:24:08 PM	✓	✗	✗	DOI: 10.5072/zenodo.369725	Edit
2	Mar 19, 2019, 2:24:08 PM	✓	✗	✗	🔒	Edit Snapshot Request DOI
3	Mar 19, 2019, 2:24:08 PM	✓	✗	✗	🔒	Edit Snapshot Request DOI

Organizations

Landing page to showcase tools and workflows

- You can organize and collect workflows from other people
- Same workflow may be in an organization based on lab, funding source
- [Organizations and Collections — Dockstore documentation](#)

Example: a COVID-19 collection that submits to Nextstrain

<https://dockstore.org/organizations/BroadInstitute/organizations/pgs>

- Contributed by: Daniel Park (Viral Genomics Group, Broad Institute)

The screenshot shows the Dockstore interface for the 'Collections' section of the Broad Institute's Viral Genomics organization. At the top, there is a search bar, an 'Organizations' dropdown, and a 'Docs' link. The user 'denis-yuen' is logged in. Below the header, the 'Collections' section is displayed.

BROAD INSTITUTE **Broad Institute of MIT and Harvard / Viral Genomics**
Viral Genomic Tools

Viral NGS Workflows

The workflows in this collection provide the ability for users to perform viral genomic data analysis. These workflows enable users to perform assembly, QC, kraken metagenomics and aggregate statistics. Additionally, we've provided workflows for users to go from raw reads (uBAM), through to producing a phylogenetic tree.

Overview of analytical workflows available

```
graph TD; FASTQ --> ImportFASTQ[Import FASTQ Data]; SRAID --> ImportSRA[Import SRA Data]; ImportFASTQ --> uBAM; ImportSRA --> uBAM; uBAM --> Kraken[Kraken Classifier]; uBAM --> ViralAssembly[Viral Assembly]; Kraken --> Fasta[FASTA]; ViralAssembly --> Fasta; Fasta --> AugurTree[Build Augur Tree]; AugurTree --> Nextstrain[Nextstrain]
```

Tutorials

The following Terra workspaces outline in detail the steps to set up and execute the listed workflows and they additionally contain example inputs and references.

[COVID-19_fread_Viral_NGS](#)
[COVID-19](#)

Currently maintained by Broad Viral Genomics & Data Sciences Platform.

github.com/broadinstitute/viral-pipelines/augur_from_newick
Last updated Jun 10, 2020

github.com/broadinstitute/viral-pipelines/augur_from_beast_mcc
Last updated Jun 10, 2020

github.com/broadinstitute/viral-pipelines/augur_from_msra
Last updated Jun 10, 2020

github.com/broadinstitute/viral-pipelines/assemble_refbased
Last updated Jun 10, 2020

github.com/broadinstitute/viral-pipelines/merge_vcfs_and_annotate
Last updated Jun 11, 2020

github.com/broadinstitute/viral-pipelines/fetch_sra_to_bam
Last updated May 12, 2020

github.com/broadinstitute/viral-pipelines/genbank

github.com/broadinstitute/viral-pipelines/mafft

Getting Help on Dockstore

User forum at <https://discuss.dockstore.org/>

- Topics embedded with each tool, workflow, and documentation page.
- Talk about bioinformatics, workflows, and get help on development



The screenshot shows the Dockstore user forum interface. At the top, there is a navigation bar with icons for search, organizations, docs, and help. The 'Help' icon is highlighted with a red box. Below the navigation bar, there is a search bar and a list of categories. The categories include General, Automatic Tools and Workflows, Dockstore, Tools and Workflows, Documentation, Staff, Staging Automatic Tools and Workflows, and Dev Dockstore. Each category has a count of topics and a brief description. To the right of the categories, there is a 'Latest' section displaying a list of threads with their titles, counts, and timestamps.

Category	Topics	Latest
General	1	Github.com/broadinstitute/gatk-svmodule07 · 0 22h
Automatic Tools and Workflows	782	Github.com/broadinstitute/gatk-svmodule07-preprocessing · 0 22h
Dockstore	9	github.com/CRI-Atlas/fatlas-workflows/synapse-EPIC · 0 22h
Tools and Workflows	7	Registry.hub.docker.com/sagebionetworks/synapsepythonclient/synapse-get-sts-tool · 0 1d
Documentation	75	Registry.hub.docker.com/sagebionetworks/synapsepythonclient/synapse-sync-to-synapse-tool · 0 1d
Staff	6	Registry.hub.docker.com/sagebionetworks/synapsepythonclient/synapse-query-tool · 0 1d
Staging Automatic Tools and Workflows	151	Registry.hub.docker.com/sagebionetworks/synapsepythonclient/synapse-store-tool · 0 1d
Dev Dockstore	22	

Documentation and Tutorials

- Example Topics:
 - Launching Tools and Workflows
 - Writing checker workflows
 - Developing File Provisioning Plugins
 - Creating Organizations
 - And many more!

<https://docs.dockstore.org/>

Developer Tutorial
Go through the process of creating a tool and registering it on Dockstore.

End User Tutorials
Learn how to use Dockstore from the perspective of a user who runs tools and workflows.

Advanced Tutorials
A collection of articles and tutorials regarding advanced Dockstore topics

The screenshot shows the Dockstore documentation website. At the top, there's a navigation bar with links for 'Docs', 'Edit on GitHub', and 'Organizations and Collections'. Below the navigation, the main content area has a heading 'Organizations and Collections' and a sub-section 'Organizations'. It explains that organizations are landing pages for collaborations, institutions, consortiums, companies, etc., that allow users to showcase tools and workflows. Collections are groupings of related tools and workflows. The page also mentions that collections can be thought of as a playlist on a music streaming service where tools and workflows are analogous to individual songs. It notes that they can be shared publicly, and the user does not need to own them. Under 'Creating an organization', it says to go to the 'organizations' page and select 'Create Organization Request'. A note states that any user can request to create an organization by filling out the following form. It emphasizes that the request must be approved by a Dockstore curator in order to be public. Until it is approved, you are still able to edit it, add collections, add members, etc.

Create Organization Request

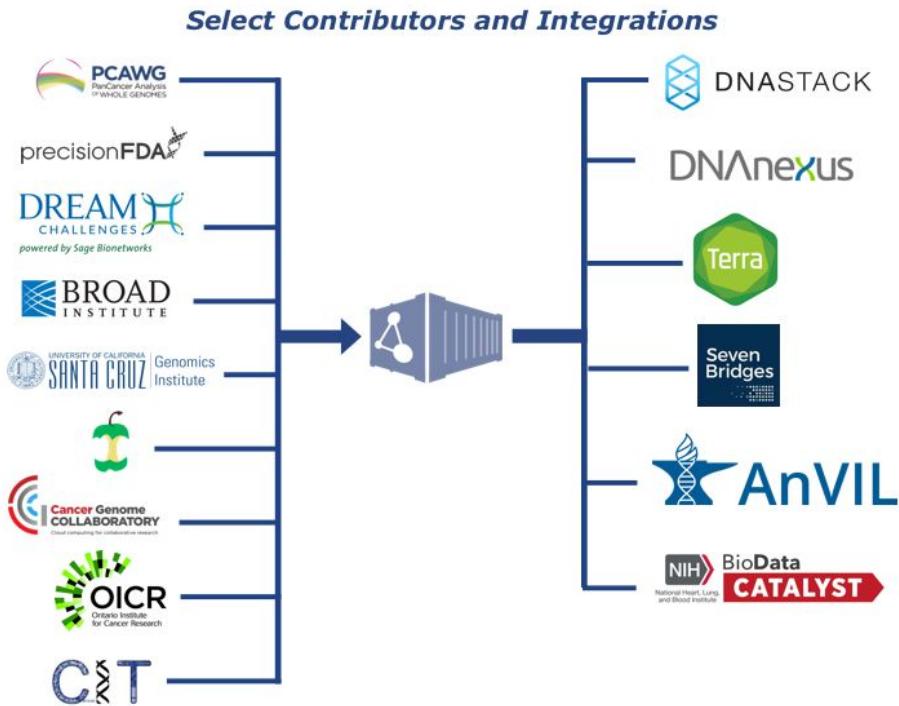
Fill out the form to send an organization request to a Dockstore curator to approve. Once approved, the organization will be publicly visible.

Name *
OICR
The name of the organization
Display Name *
Ontario Institute for Cancer Research
The display name of the organization
Topic *
OICR is a collaborative, not-for-profit research institute accelerating the development of n
A short description of the organization
Organization website
https://oicr.on.ca/
Link to organization website
Link to organization website
Toronto, ON
The location of the organization
Contact Email Address
Once approved, this email address will be publicly visible
Image URL
https://oicr.on.ca/wp-content/uploads/2017/01/OICR_Logo.png
Link to your organization's logo. Link must end in jpg, jpeg, png, or gif

Create Organization Request

Dockstore Ecosystem

Dockstore is thankful to its many contributors, users, and partners. This community has pulled together a library of over **700** tools and workflows. In the diagram to the right we've highlighted a few select contributors to give a sense of what has been occurring in this space.



The Dockstore Team



UNIVERSITY OF CALIFORNIA
SANTA CRUZ | Genomics
Institute

Lincoln Stein

Denis Yuen

Andrew Duncan

Gary Luu

Gregory Hogue

Benedict Paten

Elnaz Sarbar

Charles Overbeck

Walt Shands

David Steinberg

Nneka Olunwa

Louise Cabansay

Natalie Perez

Melaina Legaspi

Charles Reid

Emily Soth

Andy Chen

Acknowledgements



This work was funded by the Government of Canada through Genome Canada and the Ontario Genomics Institute (OGI-168).

Ontario



Funded by:



Extra Slides for Q&A

Additional Readings

- Bind Mounts - <https://docs.docker.com/storage/bind-mounts/>
- Volumes - <https://docs.docker.com/storage/volumes/>

Note: -v has historically been how volumes are mounted, however --mount is an equivalent option with a different syntax

Exercise #1a: Using Docker

Docker has a whole library of commands, here are some basic examples:

Display system-wide information about your installation of docker:

```
docker info
```

Managing docker images:

```
docker image help
```

Managing docker containers:

```
docker container help
```

Run the official hello-world docker container from dockerhub:

```
docker container run hello-world
```

Exercise #1b: Explore the Dockstore CLI (Take Home)

Make a JSON template based off a *local* WDL:

```
dockstore workflow convert wdl2json --wdl hello-task.wdl > convert.json
```

Make a JSON template based off descriptor *located remotely* on dockstore:

```
dockstore workflow convert entry2json --entry [ dockstore identifier ] > [ parameter.json ]
```

Run a descriptor located remotely on dockstore:

```
dockstore workflow launch --entry [ dockstore identifier ] --json [ parameter.json ]
```

Scatter Gather (take home reading)

Scatter

- Given an array of values, run the same task on each value in parallel (ex. Array of Files)

Gather

- Collect the results of running each scatter command in an array

Beginner Example - [Scatter Gather Pipeline](#)

Advanced Example - [Use scatter-gather to joint call genotypes](#)

What's in a WDL? Top-level Summary

3 top-level components that are part of the core structure of a WDL script

The screenshot shows a code editor window titled "workflow.wdl". The code is color-coded: "workflow" and "call" are orange, "input" and "output" are yellow, and "task" is blue. The code structure is as follows:

```
workflow myWorkflowName {
    input {
        ...
    }
    call task_A
    call task_B {
        input: ...
    }
    output {...}
}

task task_A { ... }

task task_B { ... }
```

Workflow: Code block that defines the overall workflow.

- think of it as an ‘outline’

Call: Defines *which* tasks to run

- can also specify input parameters to pass to that task.
- Located within workflow block

Task: Defines all the information necessary to perform an action.

- Tasks to run are specified by a ‘call’ inside the workflow block
- Full definition of task is done *outside* of the workflow block

What's in a Task? Summary

Task: Defines *all* the information necessary to perform an action in a **parameterized** way.

- Command ('the action')
 - the shell command(s) that will be executed when task is ran
- Inputs and Outputs
 - All inputs and outputs must be typed (ex: string, int, file, etc)
- Non-input declarations
 - Intermediate variables to help run task
- Runtime
 - Defines context/environment
 - container
 - compute resources

The screenshot shows a code editor window titled "task.wdl". The code defines a task named "doSomething" with the following structure:

```
task doSomething {
    input {
        File myName
        String outFile
    }

    command {
        echo Hello World! > {outFile}
        cat ${myName} >> {outFile}
    }

    output {
        File outFile = "{outFile}"
    }

    runtime {
        docker: docker_image
        memory: "${memory_gb}"
    }
}
```

Summary

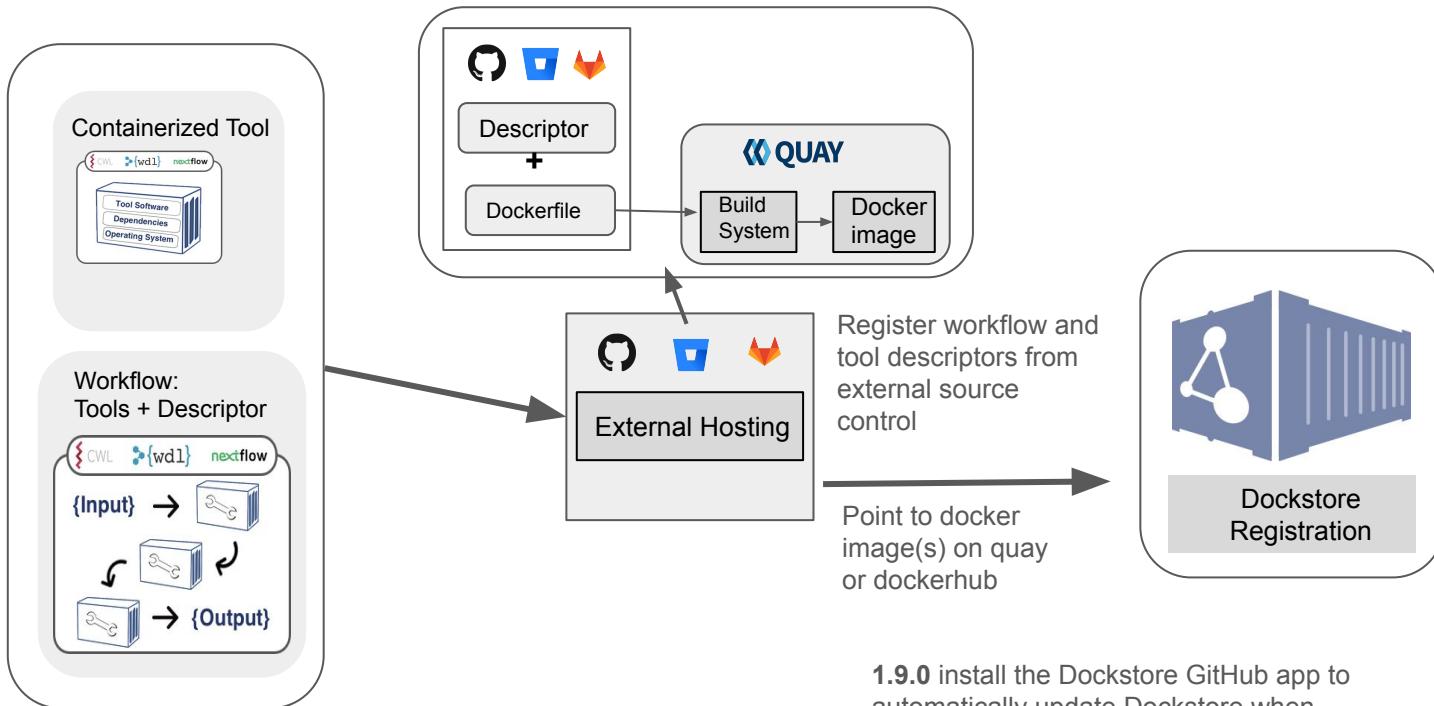
A workflow:

- A `workflow` block (with inputs and outputs)
- A `call` section to define which `task(s)` to run
- One or more `task(s)` defining what the workflow will do
- A `meta` section

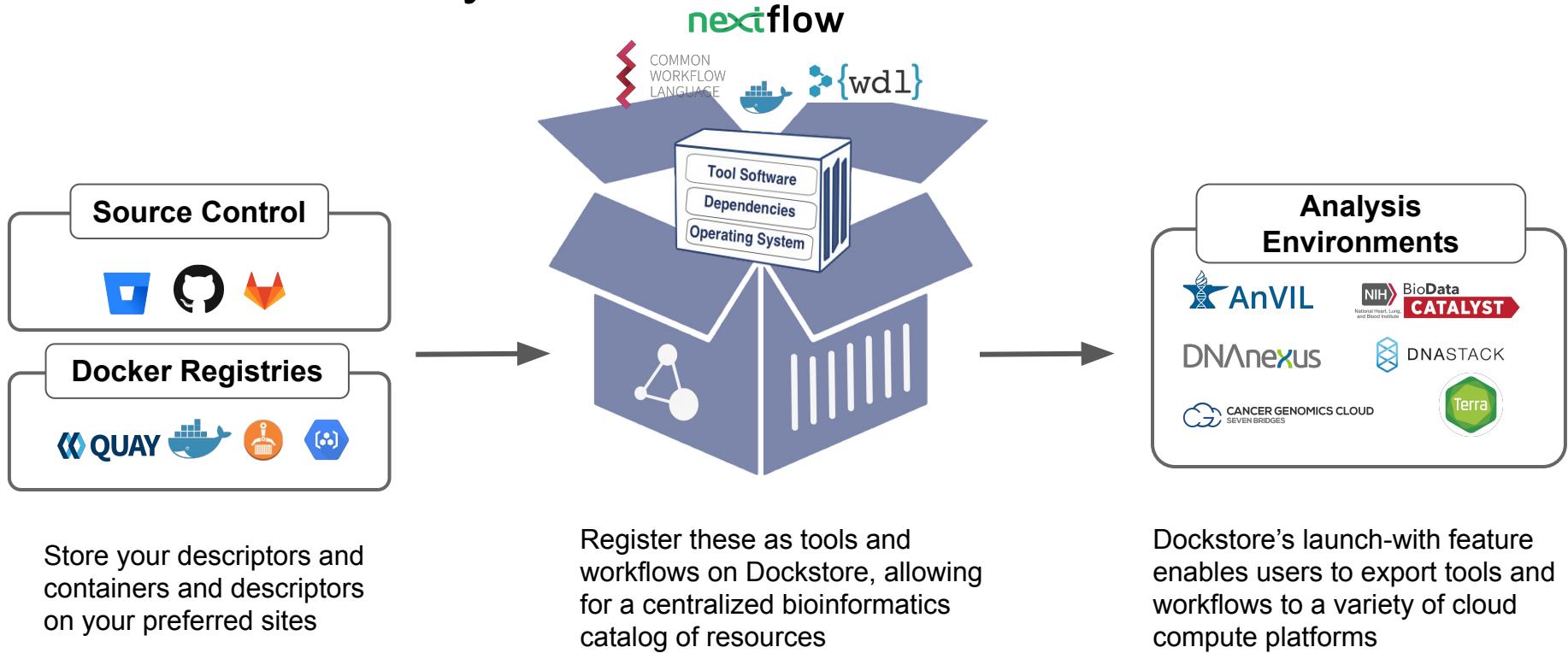
A task:

- An `input` section (required if the task will have inputs)
- Non-input declarations (as many as needed, optional)
- A `command` section (required)
- A `runtime` section (optional)
- An `output` section (required if the task will have outputs)
- A `parameter_meta` section (optional)

Ways to Register to Dockstore



Dockstore Ecosystem



Language Support

Feature	CWL	WDL	Nextflow
Dockstore site			
Tool registration	Yes	Yes	No
Workflow registration	Yes	Yes	Yes
Hosted Workflows	Yes	Yes	Yes
DAG Display	Yes (cwl version=>1.0) ^[0]	Yes (wdl version<=draft-2) ^[1]	Limited support
Tool Tab Display	Yes (cwl version=>1.0)	Yes (wdl version<=draft-2)	Yes
Launch-with Platforms	Not yet!	FireCloud (workflows only) ^[2] DNAstack (workflows only) ^[3] DNAnexus (workflows only) Terra (workflows only)	Not yet!
Metadata Display	Yes	Yes (wdl version<=draft-2)	Yes
Dockstore CLI			
Local workflow engines	cwltool, Cromwell	Cromwell ^[4]	Nextflow
File Provisioning In	Local File System HTTP FTP S3 via plugins Data Object Service	Local File System HTTP FTP S3 via plugins Data Object Service	Local File System HTTP FTP S3
Plugins Support	s3 s3cmd icgc-get Data Object Service	s3 s3cmd icgc-get Data Object Service	No
File Provisioning Out	Local File System HTTP FTP S3 via plugins	Local File System	Local File System S3
Notifications	Yes	Yes	No

More info:

<https://docs.dockstore.org/en/develop/end-user-topics/language-support.html>