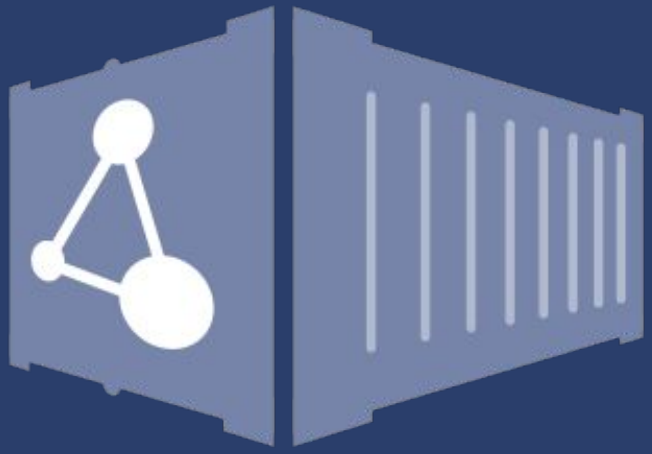


Dockstore Fundamentals:

Introduction to Docker and Descriptors for Reproducible Analysis

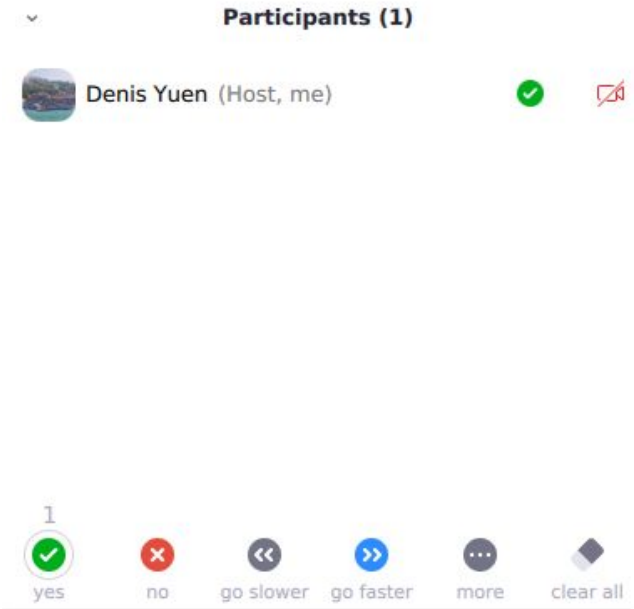
Louise Cabansay, Software Engineer, UC Santa Cruz Genomics Institute
Andrew Duncan, Software Engineer, Ontario Institute for Cancer Research
Denis Yuen, Senior Software Engineer, Ontario Institute for Cancer Research

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?

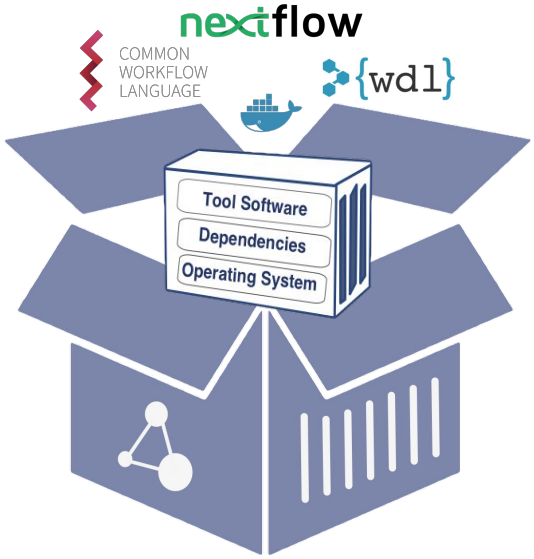
Dockstore is a free and open source platform for sharing scientific tools and workflows.

Portability

Interoperability

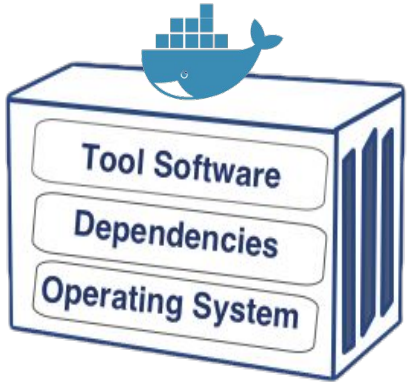
Reproducibility

“An app store for bioinformatics”



Portability:

Container



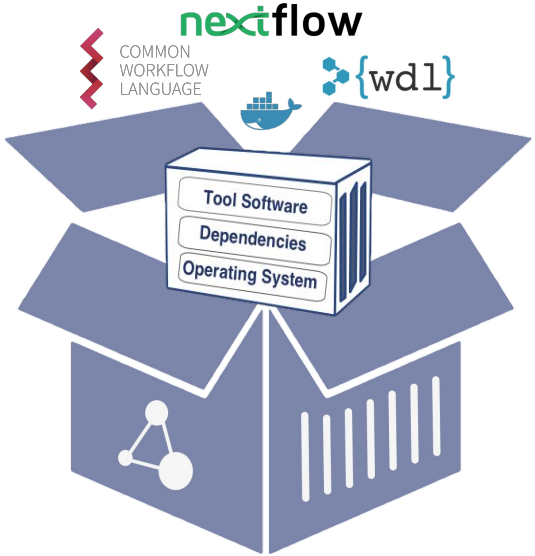
+

Descriptor

COMMON WORKFLOW LANGUAGE

{wdl}

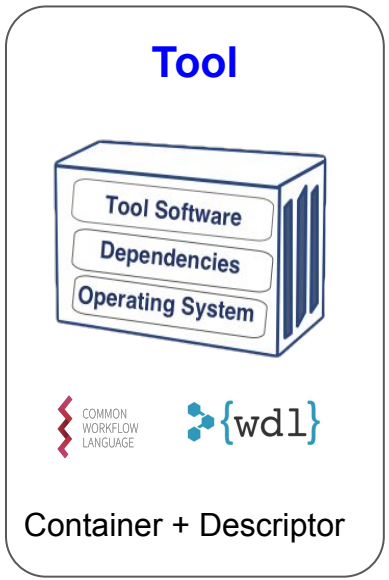
nextflow



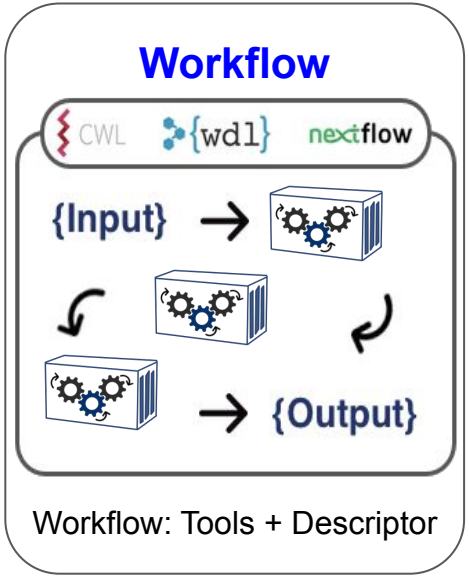
Software is “packaged” using container technology and described using descriptor languages

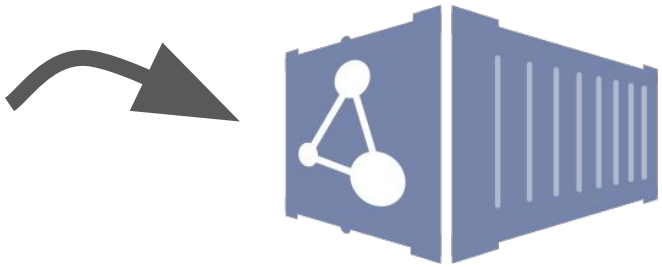
- Analysis can be moved from environment-to-environment (local machines, clouds, servers) and yet be guaranteed to run on anything that supports Docker

What's on Dockstore? tools and workflows



OR





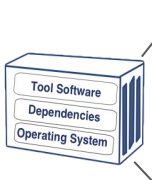
A tool uses a single container and performs a single action or step that is outlined by a descriptor.

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

Example:

Tool

BWA-MEM



Debian OS
binaries
C
GNU Make
GCC
zlibs

COMMON WORKFLOW LANGUAGE

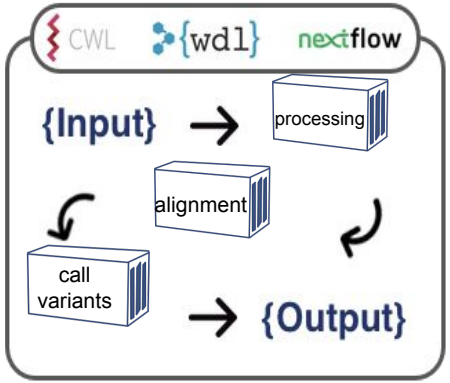
{wdl}

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

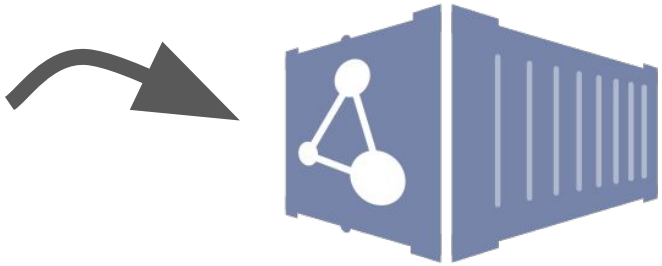
OR

Workflow

Variant Calling Pipeline



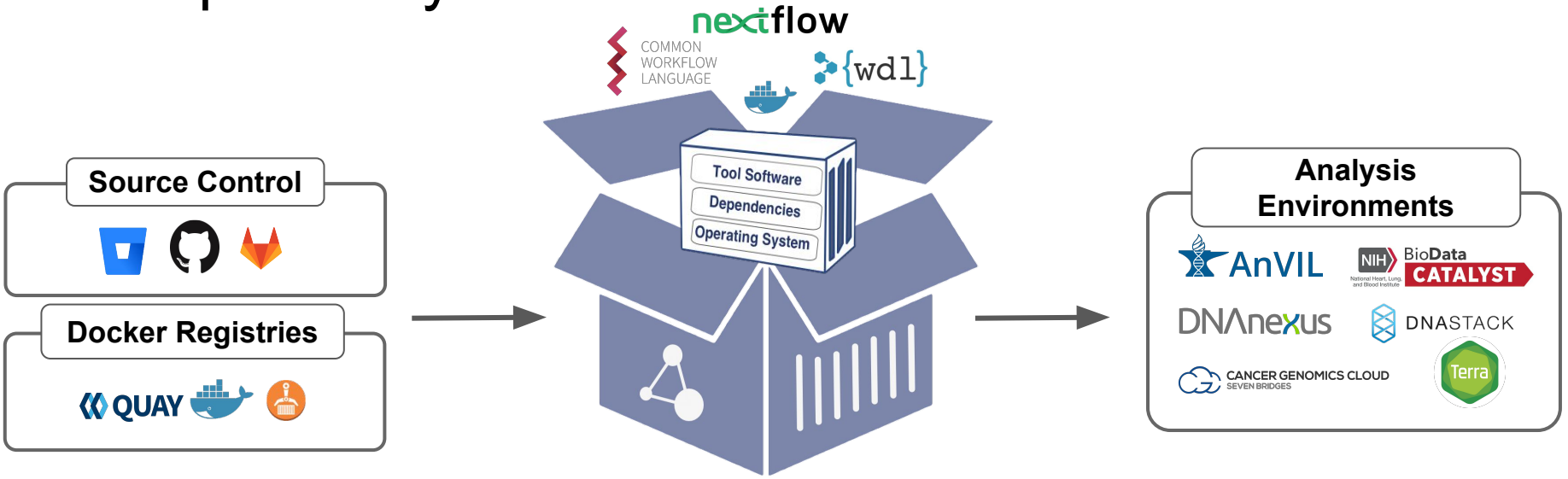
A workflow can use multiple containers and executes multiple actions or steps, still outlined by a descriptor



*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

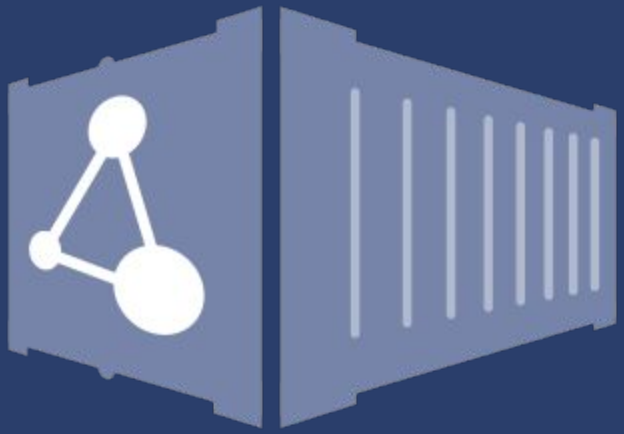
Global Alliance for Genomics & Health
Collaborate. Innovate. Accelerate.

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

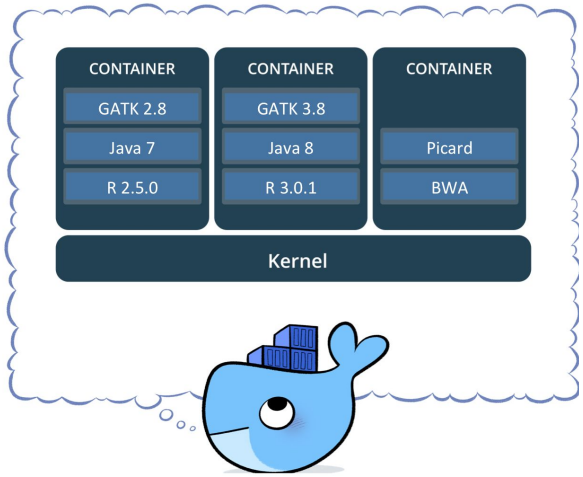
*as used on Dockstore

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.

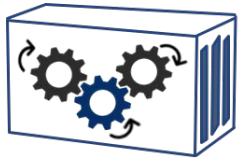


Image:

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

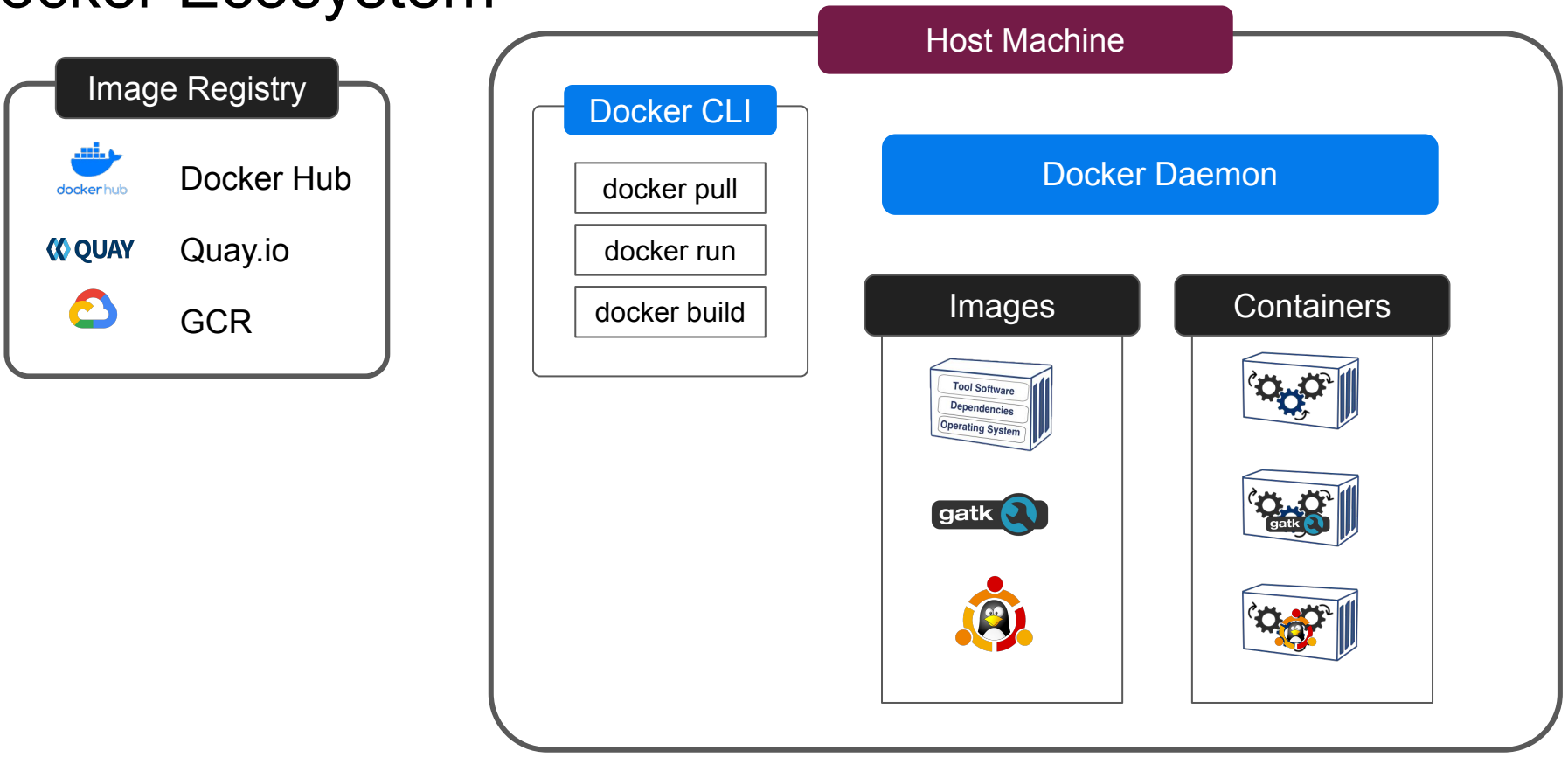
Registry:

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

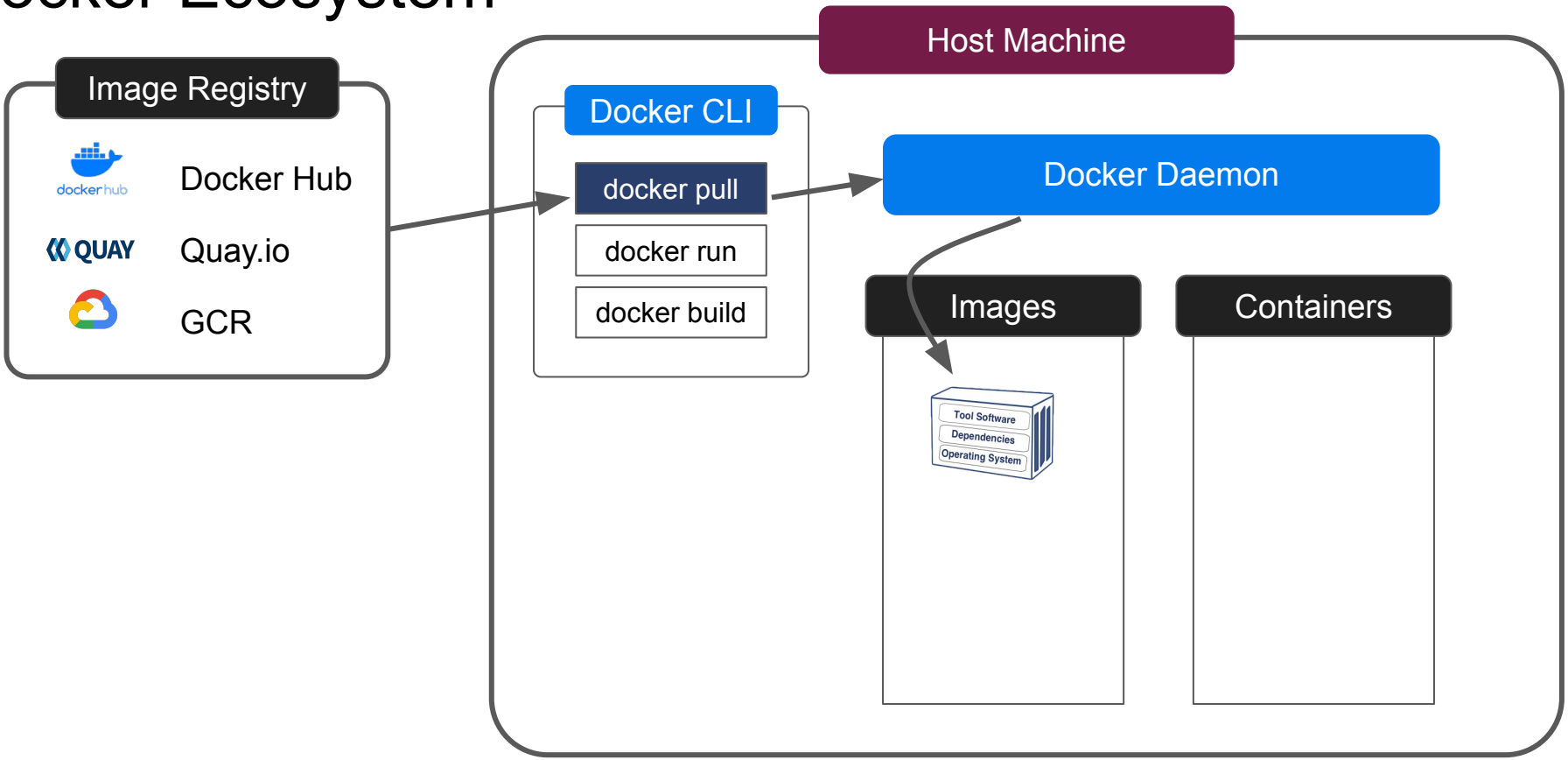
Dockstore itself does not host images, but rather gets them from Image Registries:

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

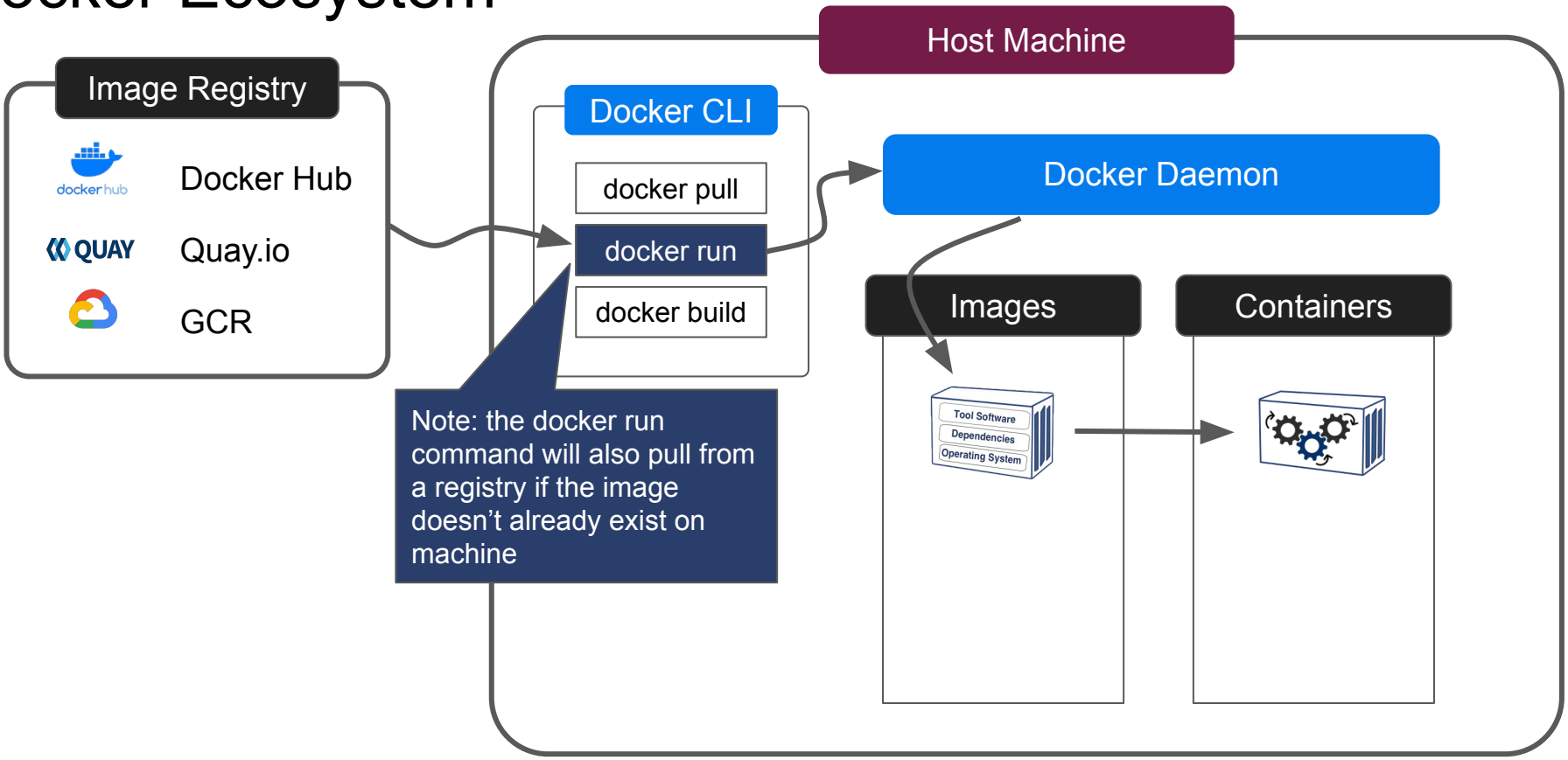
Docker Ecosystem



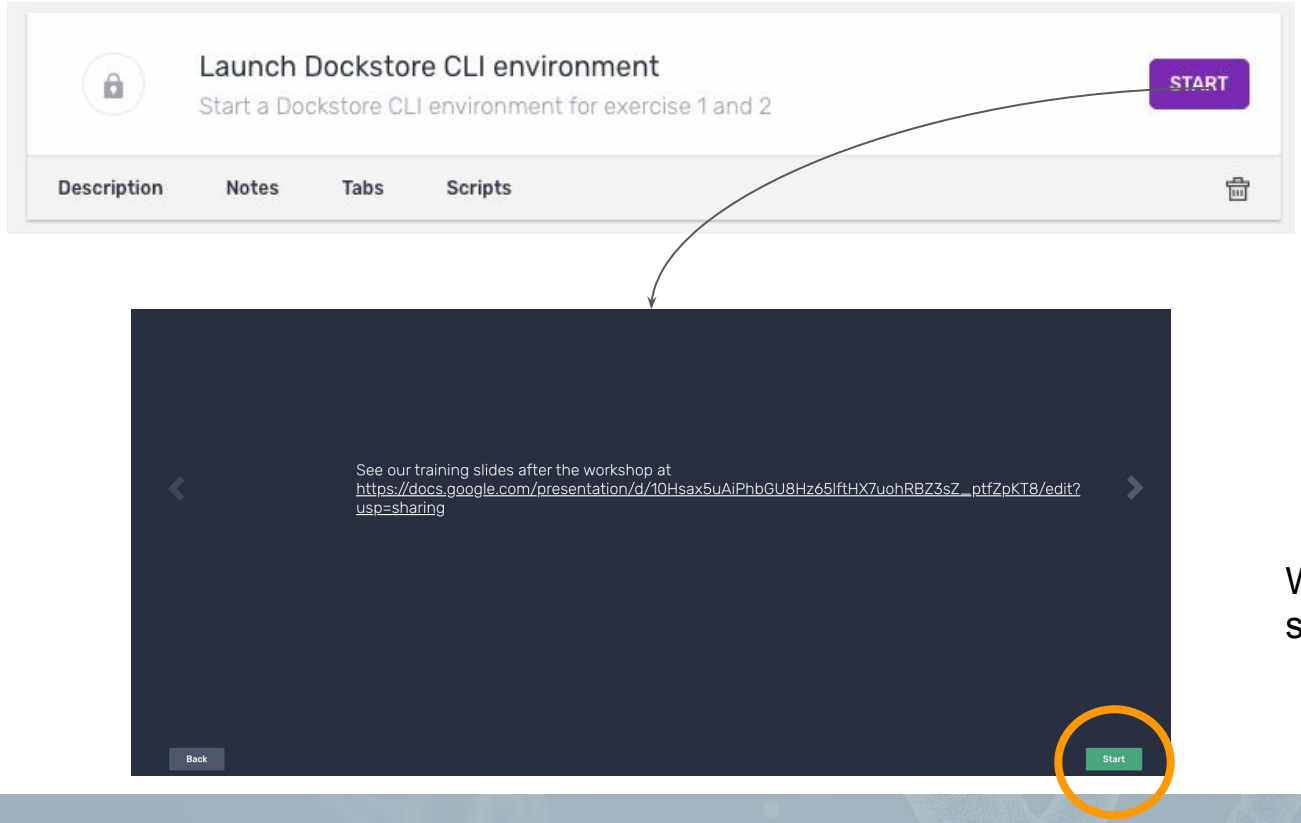
Docker Ecosystem



Docker Ecosystem



Start Up Instruqt



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!!

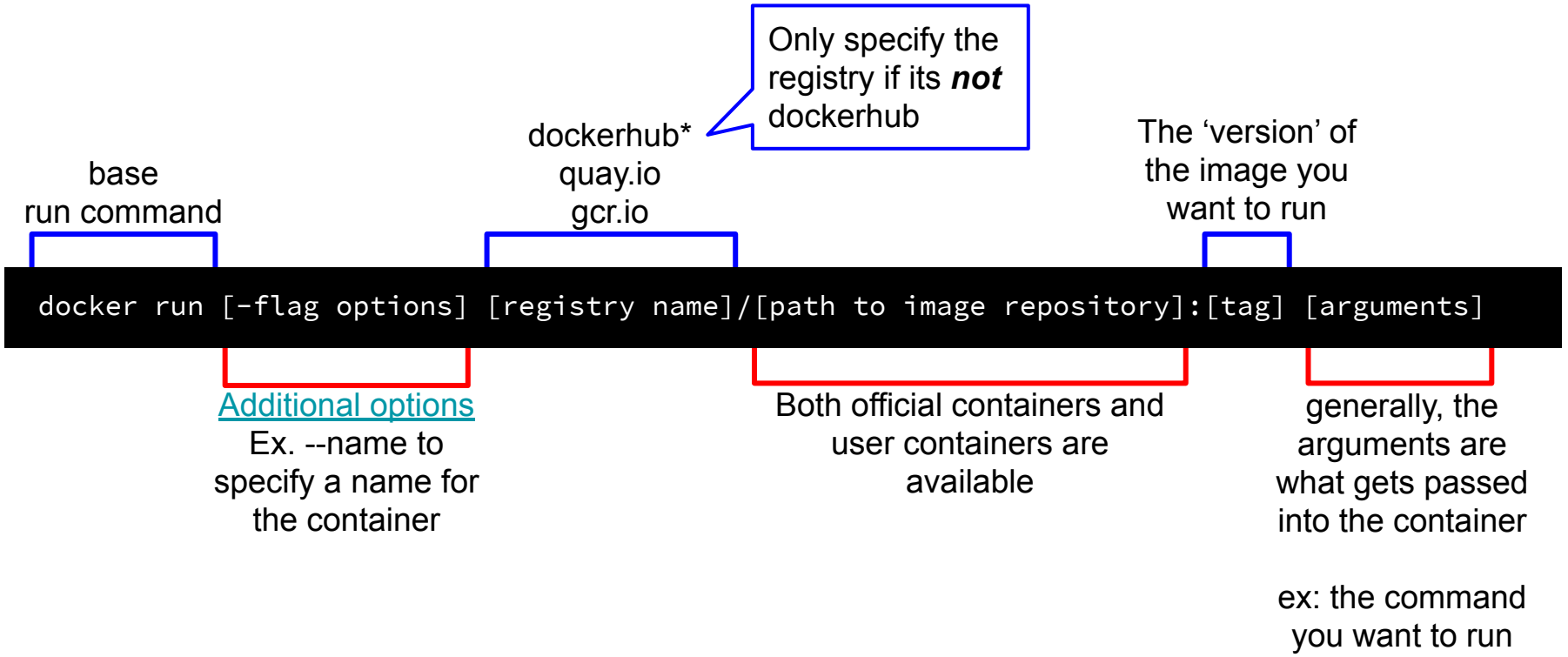
Run and done

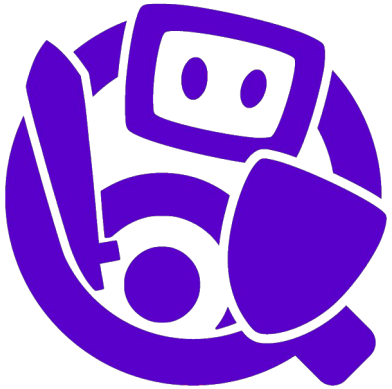
1. Execute docker run command
2. Actions executed in container (stuff happens)
3. Container stops running after completion

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?





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:

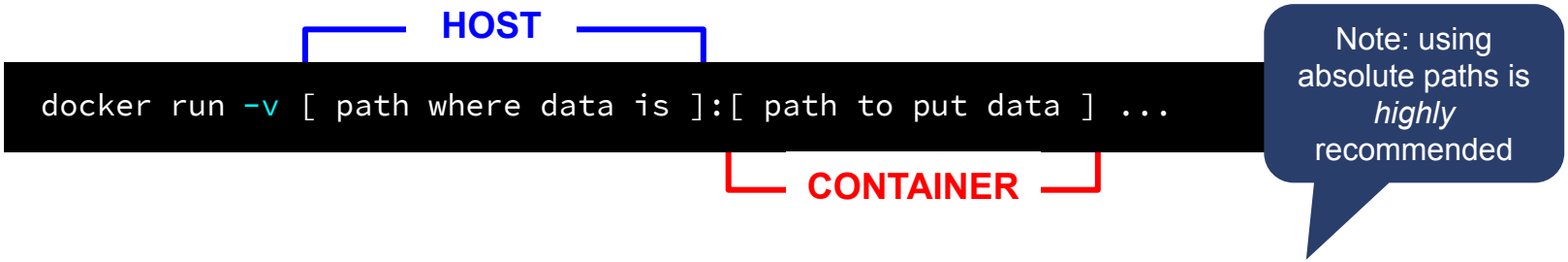
Enter the samtools container and confirm that samtools is installed

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

Sharing data between host and container

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

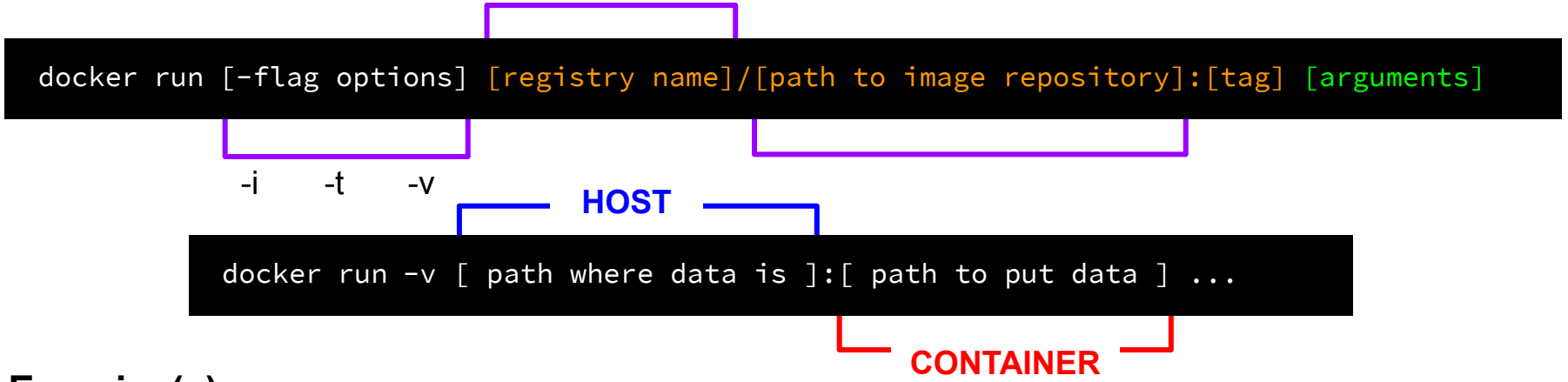
- Map a host directory to a directory in a container
- Any files added to either directory is available in the other



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



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

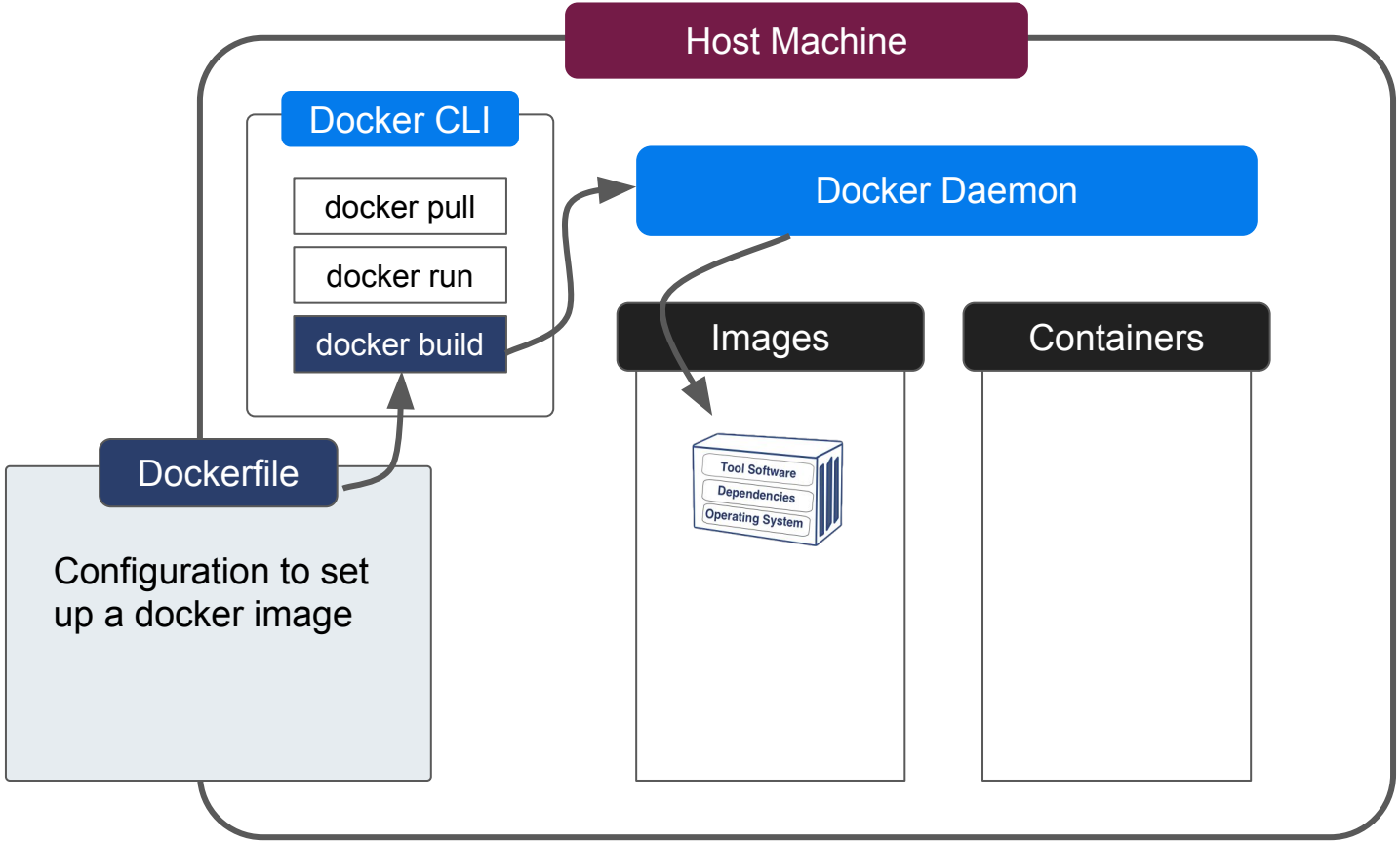
Image built from the dockerfile can then be used for 'off-the-shelf' software usage by others.

Dockerfiles Overview:

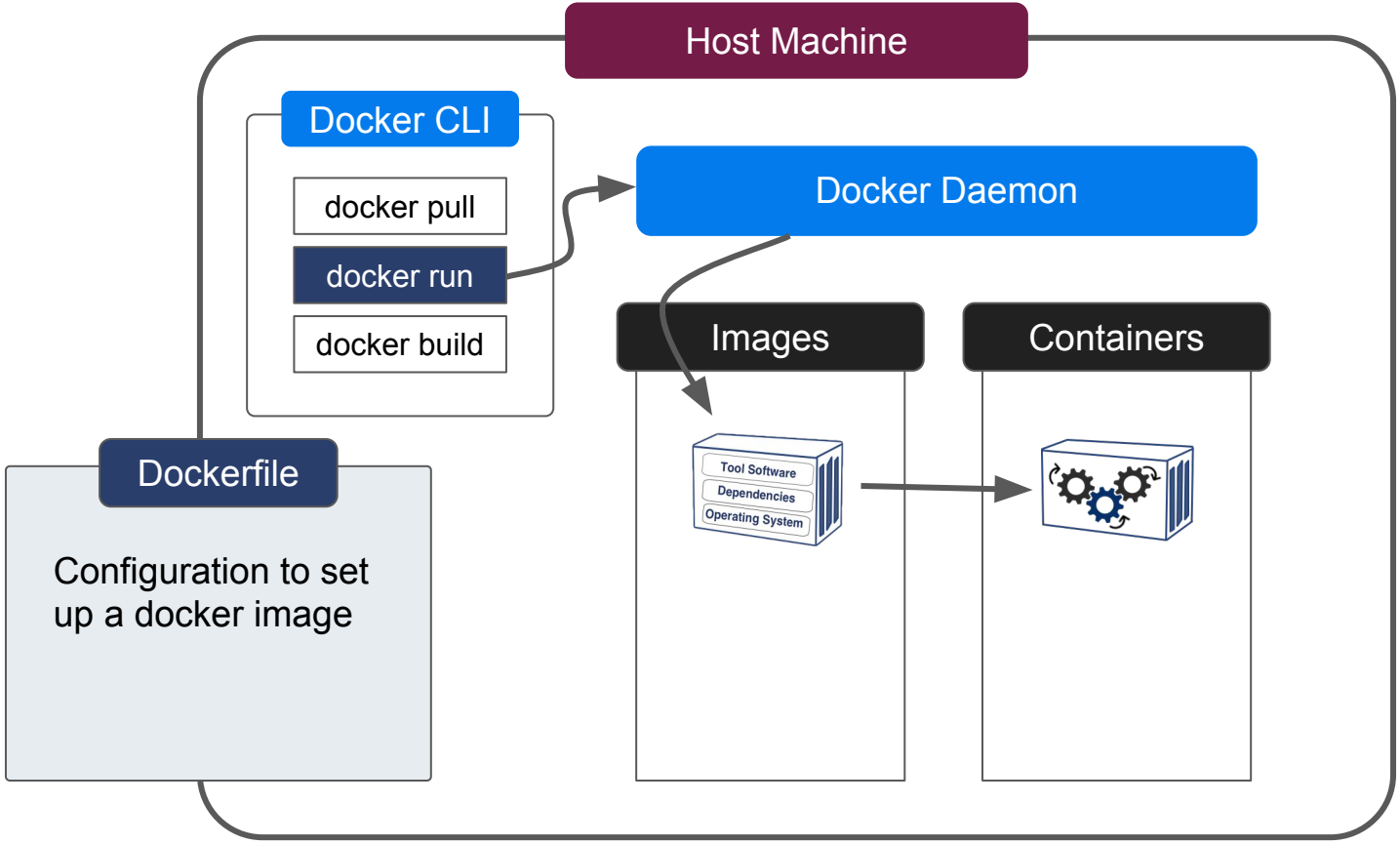
A simple text file with instructions to build an image:

- 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

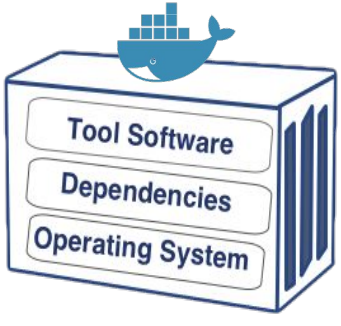


Dockerfiles - local



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

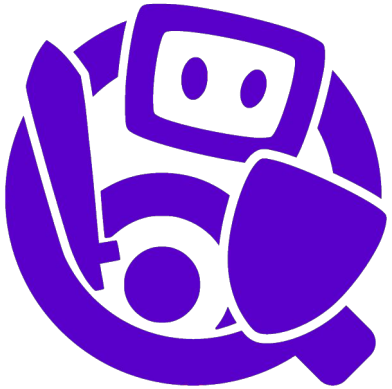
```
docker build -t bwa:v1.0 .
```

(-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

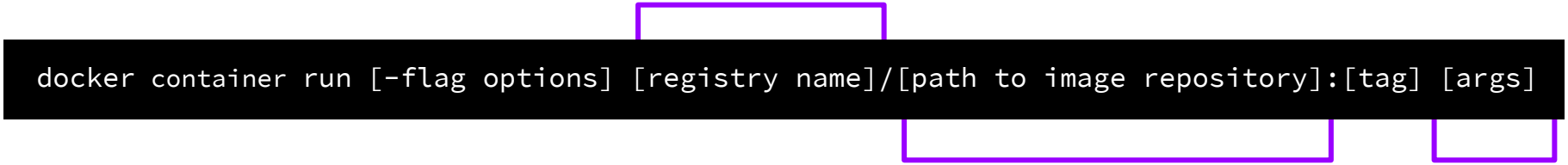
Dockerfile

```
# Start with a base image
FROM { base image name }

# Add file author/maintainer and contact info (optional)
MAINTAINER {your name} <youremail@research.edu>

# set user you want to install packages as
USER root

# update package manager & install dependencies (if any)
RUN apt update

# install analysis software from package manager
RUN apt install -y { software package name }
```

Build an image from Dockerfile:

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

Exercise #2b: Try out your new container!



Exercises:

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

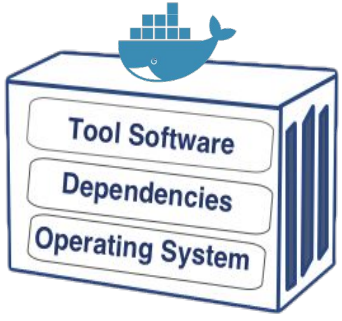
```
docker image ls
```

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

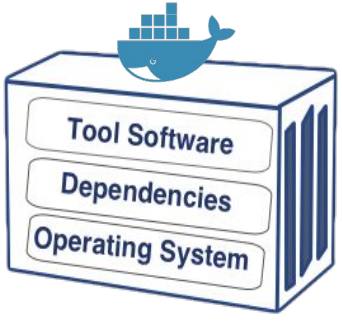


Dockerfile

```
# Start with a base image
FROM ubuntu:14.04

# Add file author/maintainer and contact info (optional)
MAINTAINER Brian OConnor <briandoconnor@gmail.com>

# install software dependencies
USER root
RUN apt-get -m update && apt-get install -y wget unzip \
    openjdk-7-jre zip


# manual software installation from source
# get the tool and install it in /usr/local/bin
RUN wget -q http://downloads.sourceforge.net/project
/bamstats/BAMStats-1.25.zip

# commands/scripts to finish software setup
RUN unzip BAMStats-1.25.zip && \
    rm BAMStats-1.25.zip && \
    mv BAMStats-1.25 /opt/

COPY bin/bamstats /usr/local/bin/
RUN chmod a+x /usr/local/bin/bamstats

# switch back to the ubuntu user so this tool (and the
files written) are not owned by root
RUN groupadd -r -g 1000 ubuntu && useradd -r -g ubuntu -u
1000 -m ubuntu
USER ubuntu

# command /bin/bash is executed when container starts
CMD ["/bin/bash"]
```

Example: samtools (compile from source files)

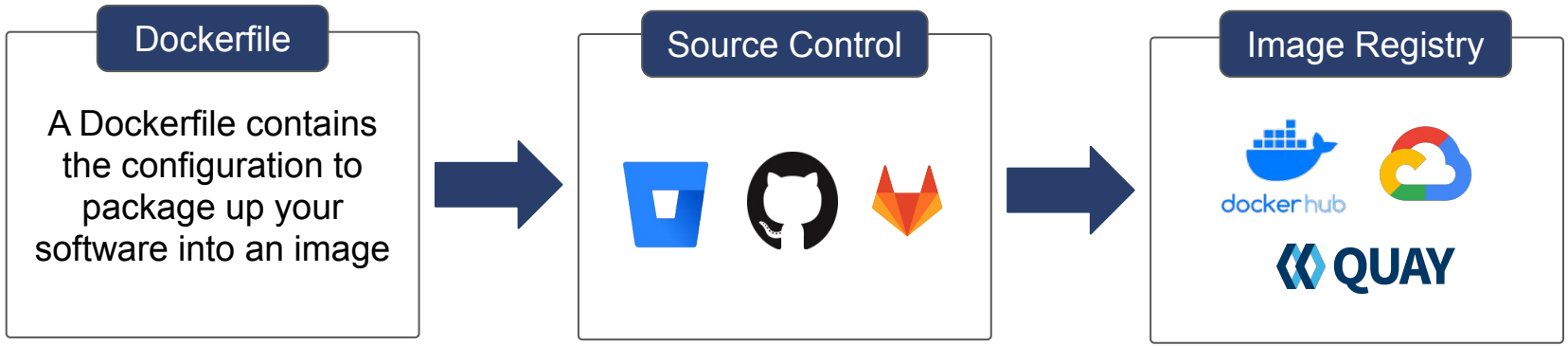
- Images of containers are built from Dockerfiles
- Dockerfiles describe the packaged up environment:
  - operating system or base image to build upon
  - dependencies needed for the software
  - the actual analysis software
  - Scripts or commands to complete environment set- up



Dockerfile

```
# Start with a base image
FROM ubuntu:18.04

# Add file author/maintainer and contact info (optional)
MAINTAINER Louise Cabansay <lcabansa@ucsc.edu>

# install software dependencies
RUN apt update && apt -y upgrade && apt install -y \
    wget build-essential libncurses5-dev zlib1g-dev \
    libbz2-dev liblzma-dev libcurl3-dev \

WORKDIR /usr/src

# get the software source files
RUN wget https://github.com/samtools/samtools/releases/
download/1.10/samtools-1.10.tar.bz2


# installation commands to compile source files
tar xjf samtools-1.10.tar.bz2 && \
    rm samtools-1.10.tar.bz2 && \
    cd samtools-1.10 && \
    ./configure --prefix $(pwd) && \
    make

# add newly built executables to path
ENV PATH="/usr/src/samtools-1.10:${PATH}"
```

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!

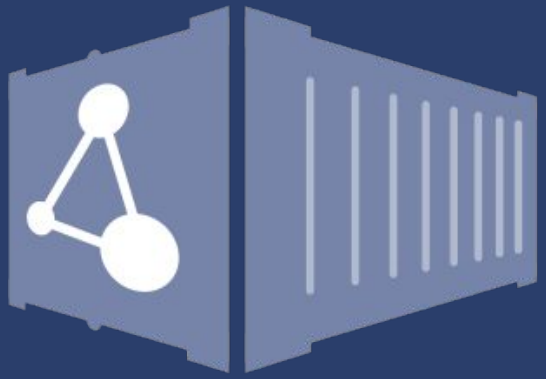
Break

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!

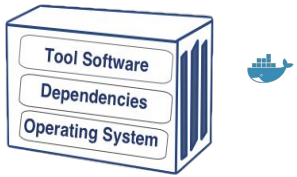


Intro to Descriptors (WDL)

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

COMMON WORKFLOW LANGUAGE

{wdl}

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.bw
a.CEU.low_coverage.20121211.bam"
    },
    "bamstats_report": {
        "class": "File",
        "path": "/tmp/bamstats_report.zip"
    }
}
```

CWL: Common Workflow Language

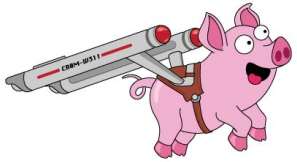
CWL: Common Workflow Language

- 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

COMMON WORKFLOW LANGUAGE

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

Running nextflow workflows

- Works on local machines, HPC, AWS, Google Cloud
- Cloud support via Sequera Labs

WDL: Workflow Description Language

WDL: Workflow Description Language

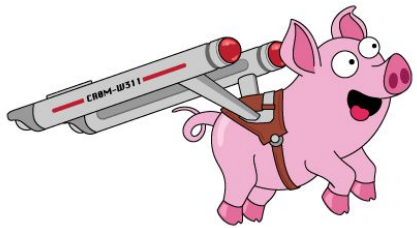
- human-readable and writable descriptor language

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

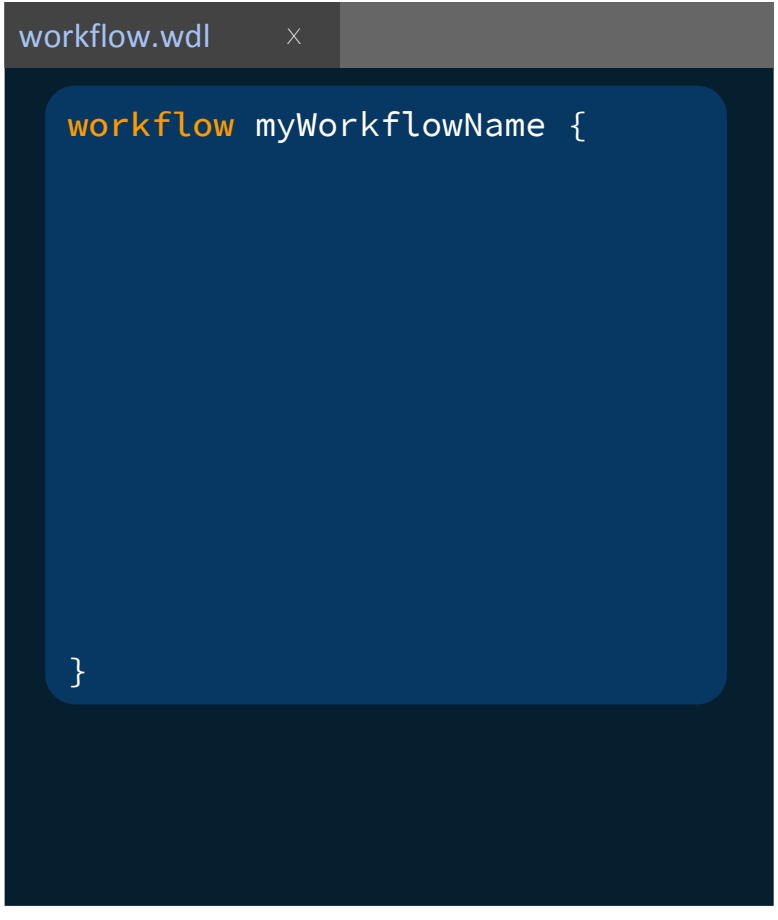


What's in a WDL? Top-level Components

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

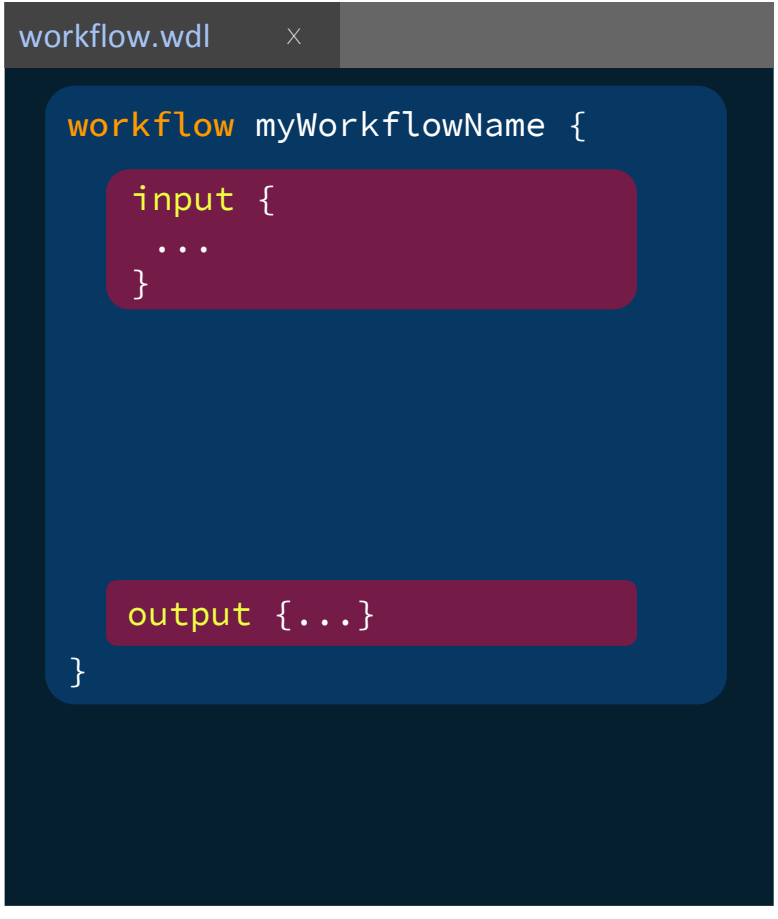
- Workflow
- Call
- Task

Top-level Components - Workflow



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

Top-level Components - Workflow



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

```
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

```
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

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

task.wdl

```
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" }
```

task.wdl

```
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

task.wdl

```
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

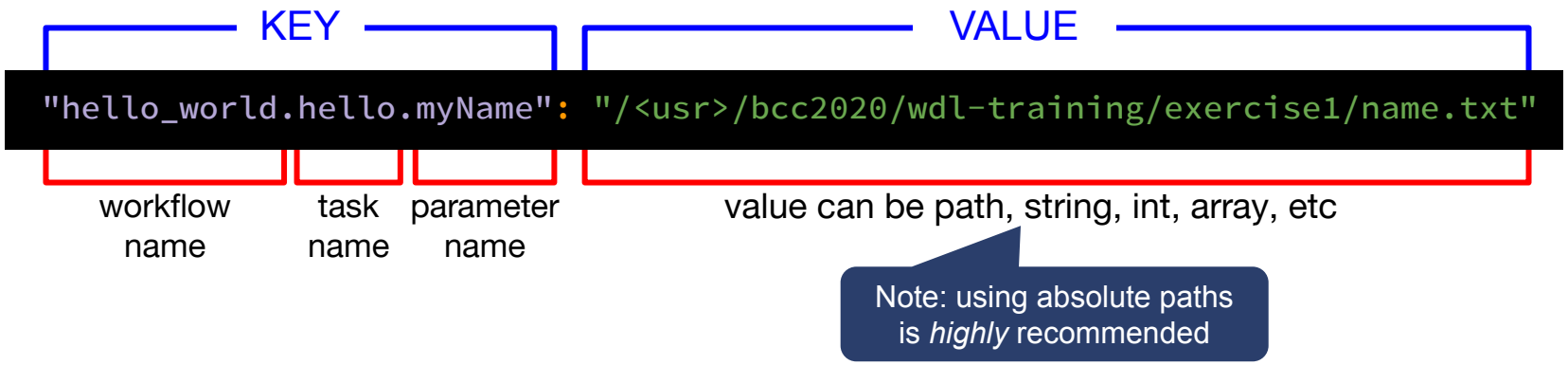
```
version 1.0
# add and name a workflow block
workflow hello_world {
   call hello
   # important: add output for whole workflow
   output {
     File helloFile = hello.outFile
   }
}

# define the ‘hello’ task
task hello {
    input { File myName }
    command {
        echo Hello World! > Hello.txt
        cat ${myName} >> Hello.txt
    }
    output { File outFile = “Hello.txt” }
}

```

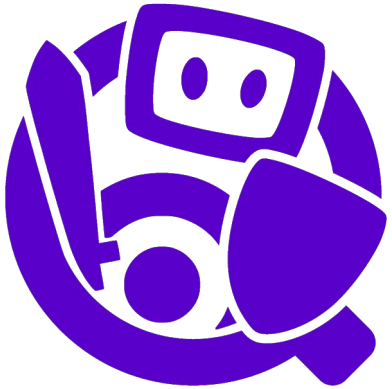
Parameter JSON (simple):

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

hello.json

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





instruqt

Exercise #1: Run your first wdl

- Single task workflow
  - Workflow block not required

HelloWorld.wdl

```
version 1.0
workflow hello_world {
   call hello
   output { File helloFile = hello.outFile }
}

task hello {
    input { File myName }
    command {
        echo Hello World! > Hello.txt
        cat ${myName} >> Hello.txt
    }
       output { File outFile = “Hello.txt” }
}

```

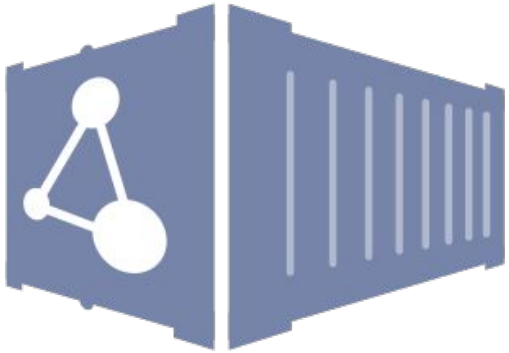
Run with DockstoreCLI

```
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

task.wdl

```
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?

task.wdl

```
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

```
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

```
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

```
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/ldcabansay/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

```
version 1.0

workflow metrics {
   call flagstat
   output { File align_metrics = flagstat.metrics }
}

task flagstat {
    input { File input_sam }

    # non-parameterized flagstat command
    command {
      samtools flagstat mini.sam > mini.sam.metrics
    }
    output {
      File metrics = “mini.sam.metrics”
    }

    # set some parameterized runtime parameters
    runtime {
    docker: # set
    }
}

```

Exercise #2: Solution* - Descriptors

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

metrics.wdl

```
version 1.0

workflow metrics {
   call Flagstat
   output { File align_metrics = flagstat.metrics }
}

task flagstat {
    input { File input_sam }

    # non-parameterized flagstat command
    command {
      samtools flagstat mini.sam > mini.sam.metrics
    }

    output {
      File metrics = “mini.sam.metrics”
    }

    # set some parameterized runtime parameters
    runtime {
    docker: # set
    }
}
```

metrics.wdl

```
version 1.0

workflow metrics {
   call Flagstat
   output { File align_metrics = flagstat.metrics }
}

task flagstat {
    input { File input_sam }

    # slightly parameterized flagstat command
    command {
      samtools flagstat ${input_sam} > mini.sam.metrics
    }

    output { File metrics = “mini.sam.metrics” }

    # set docker runtime
    runtime {
    docker: “quay.io/ldcabansay/samtools:latest”
    }
}
```

Exercise #2: Solution vs Solution2

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

metrics.wdl

```
version 1.0

workflow metrics {
   call Flagstat
   output { File align_metrics = flagstat.metrics }
}

task flagstat {
    input { File input_sam }

    # slightly parameterized flagstat command
    command {
       samtools flagstat ${input_sam} > mini.sam.metrics
    }

    output { File metrics = “mini.sam.metrics” }

    # set docker runtime
    runtime {
    “quay.io/ldcabansay/samtools:latest”
    }
}
```

metrics.wdl

```
version 1.0

workflow metrics {
   call Flagstat
   output { File align_metrics = flagstat.metrics }
}

task flagstat {
    input {
      File input_sam
      String docker_image
    }
    # create a string to help parameterize command
    String stats = basename(input_sam) + “.metrics”
    command {
       samtools flagstat ${input_sam} > ${stats}
    }
    output { File metrics = “${stats}” }

    # set a parameterized docker runtime
    runtime {
    docker: docker_image
    }
}
```

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

```
version 1.0
workflow hello_world {
   call hello
   output { File helloFile = hello.outFile }
}

task hello {
    input { File myName }

    command {
        echo Hello World!
        cat ${myName}
    }
    output { File outFile = stdout() }
}
```

GoodbyeWorld.wdl

```
version 1.0
workflow goodbye_world {
   call goodbye
   output { File byeFile = goodbye.outFile }
}

task goodbye {
    input { File greeting }

    command {
        cat ${greeting}
        echo See you later!
    }
    output { File outFile = stdout() }
}
```

Example: Multi-task workflow - HelloGoodbye.wdl

HelloGoodbye.wdl

```
version 1.0
workflow HelloGoodbye {
    call hello
    call goodbye {
      input: greeting = hello.outFile
    }
    output { File  hello_goodbye = goodbye.outFile }
}
task hello {
    input { File myName }
    command {
        echo Hello World!
        cat ${myName}
    }
    output { File outFile = stdout() }
}
task goodbye {
    input { File greeting }
    command {
        cat ${greeting}
        echo See you later!
    }
    output { File outFile = stdout() }
}
```

HelloGoodbye.json

```
{
"HelloGoodbye.hello.myName":
"/root/bcc2020-training/wdl-training/exercise3/
hello_examples/name.txt"
}
```

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>"
```

workflow name | task name | parameter name

Example: No Imports vs Imports

HelloGoodbye.wdl

```
version 1.0
workflow HelloGoodbye {
    call hello
    call goodbye {
      input: greeting = hello.outFile
    }
    output { File hello_goodbye = goodbye.outFile }
}
task hello {
    input { File myName }
    command {
        echo Hello World!
        cat ${myName}
    }
    output { File outFile = stdout() }
}
task goodbye {
    input { File greeting }
    command {
        cat ${greeting}
        echo See you later!
    }
    output { File outFile = stdout() }
}
```

HelloGoodbye_imports.wdl

```
version 1.0
# add import statements to bring in sub-workflows
# if not given, namespace/alias = file minus '.wdl'
import "HelloWorld.wdl"

# otherwise, namespace = <alias>
import "GoodbyeWorld.wdl" as bye

workflow HelloGoodbye {

    #call the hello task, syntax: <alias>.taskname
    call HelloWorld.hello

    #call the goodbye task, syntax: <alias>.taskname
    call bye.goodbye {
      input: greeting = hello.outFile
    }
    #same as before, define workflow outputs
    output { File hello_goodbye = goodbye.outFile }
}
```

Example: No Imports vs Imports

HelloGoodbye.wdl

```
version 1.0
workflow HelloGoodbye {
    call hello
    call goodbye {
      input: greeting = hello.outFile
    }
    output { File hello_goodbye = goo
}
task hello {
    input { File myName }
    command {
        echo Hello World!
        cat ${myName}
    }
    output { File outFile = stdout() }
}
task goodbye {
    input { File greeting }
    command {
        cat ${greeting}
        echo See you later!
    }
    output { File outFile = stdout() }
}
```

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

HelloGoodbye_imports.wdl

```
version 1.0
# add import statements to bring in sub-workflows
# if not given, namespace/alias = file minus '.wdl'
import "HelloWorld.wdl"

# otherwise, namespace = <alias>
import "GoodbyeWorld.wdl" as bye

workflow HelloGoodbye {

    #call the hello task, syntax: <alias>.taskname
    call HelloWorld.hello

    #call the goodbye task, syntax: <alias>.taskname
    call bye.goodbye {
      input: greeting = hello.outFile
    }
    #same as before, define workflow outputs
    output { File hello_goodbye = goodbye.outFile }
}
```

Example: No Imports vs Imports (no comments)

HelloGoodbye.wdl

```
version 1.0
workflow HelloGoodbye {
   call hello
   call goodbye {
     input: greeting = hello.outFile
   }
   output { File hello_goodbye = goodbye.outFile }
}
task hello {
    input { File myName }
    command {
        echo Hello World!
        cat ${myName}
    }
    output { File outFile = stdout() }
}
task goodbye {
    input { File greeting }
    command {
        cat ${greeting}
        echo See you later!
    }
    output { File outFile = stdout() }
}
```

HelloGoodbye_imports.wdl

```
version 1.0
import “HelloWorld.wdl”
import “GoodbyeWorld.wdl” as bye

workflow HelloGoodbye {
   call HelloWorld.hello
   call bye.goodbye {
     input: greeting = hello.outFile
   }
   output { File hello_goodbye = goodbye.outFile }
}
```

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

aligner.wdl

```
version 1.0
workflow alignReads {
    call bwa_align
    output { File output_sam = bwa_align.output_sam }
}
task bwa_align {
    input {
        String sample_name
        String docker_image
        String? bwa_options
        File read1_fastq
        File read2_fastq
        File ref_fasta
        File ref_fasta_fai
        File ref_fasta_amb
        File ref_fasta_ann
        File ref_fasta_bwt
        File ref_fasta_pac
        File ref_fasta_sa
    }
    String output_sam = "${sample_name}" + ."sam"

    command {
        bwa mem ${bwa_options} ${ref_fasta} \
        ${read1_fastq} ${read2+fastq} > ${output_sam}
    }
    output { File output_sam = "${output_sam}" }
    runtime {
       docker: docker_image
       memory: "${memory_gb}" + "GB"
    }
    meta {
    author: "Foo Bar"
    email: "foobar@university.edu"
    }
}
```

Example: Metrics.wdl (samtools flagstat)

Same as the solution to exercise #2

metrics.wdl

```
version 1.0

workflow metrics {
   call Flagstat
   output { File align_metrics = Flagstat.metrics }
}

task Flagstat {
    input {
      File input_sam
      String docker_image
    }
    # create a string to help parameterize command
    String stats = basename(input_sam) + ".metrics"
    command {
       samtools flagstat ${input_sam} > ${stats}
    }
    output { File metrics = "${stats}" }

    # set some parameterized runtime parameters
    runtime {
    docker: docker_image
    }
}
```

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

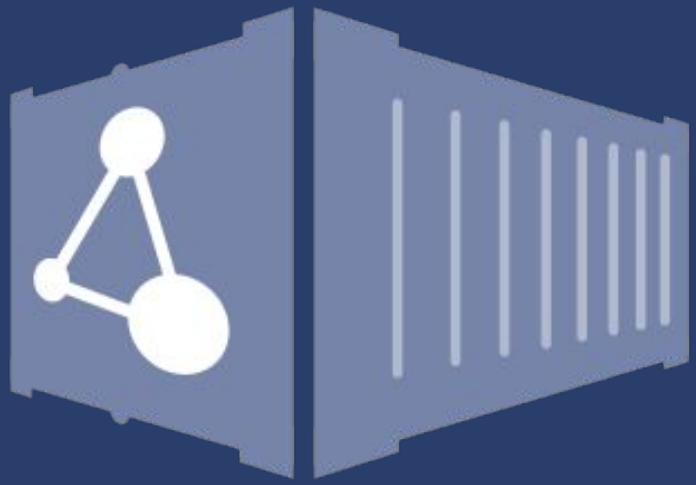
- **parameter_meta** section within a task
- Optional key/value pairs
- Describe parameters
- Key **must** map to a task input or output

task.wdl

```
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?

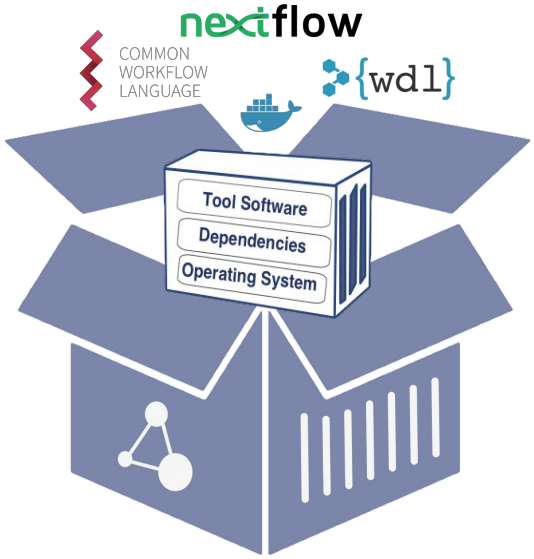
Dockstore is a free and open source platform for sharing scientific tools and workflows.

Portability

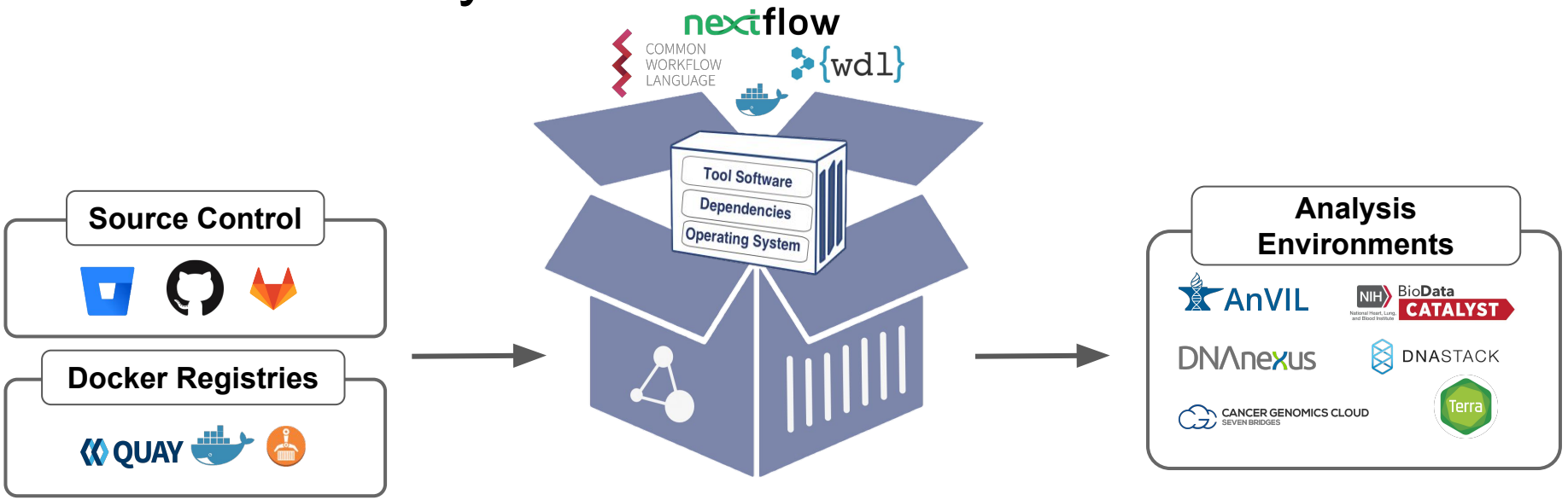
Interoperability

Reproducibility

“An app store for bioinformatics”



Dockstore Ecosystem

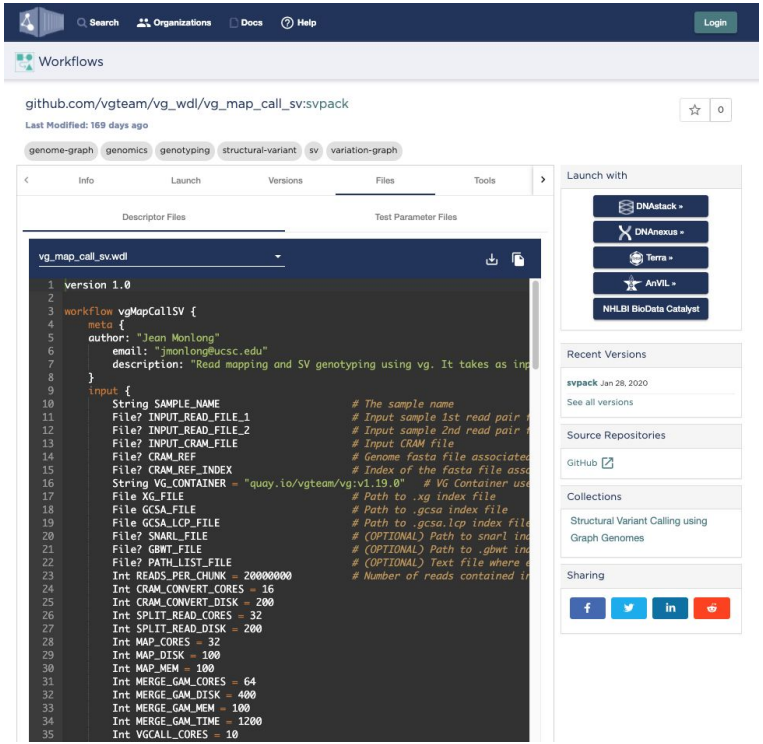


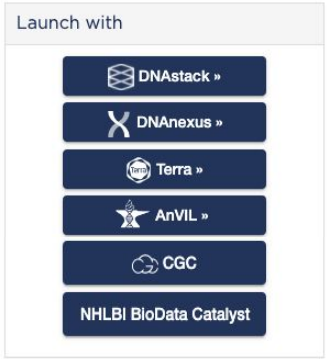
Store your descriptors and containers and descriptors on your preferred sites

Register these as tools and workflows on Dockstore, allowing for a centralized bioinformatics catalog of resources

Dockstore's launch-with feature enables users to export tools and workflows to a variety of cloud compute platforms

Launching Analysis





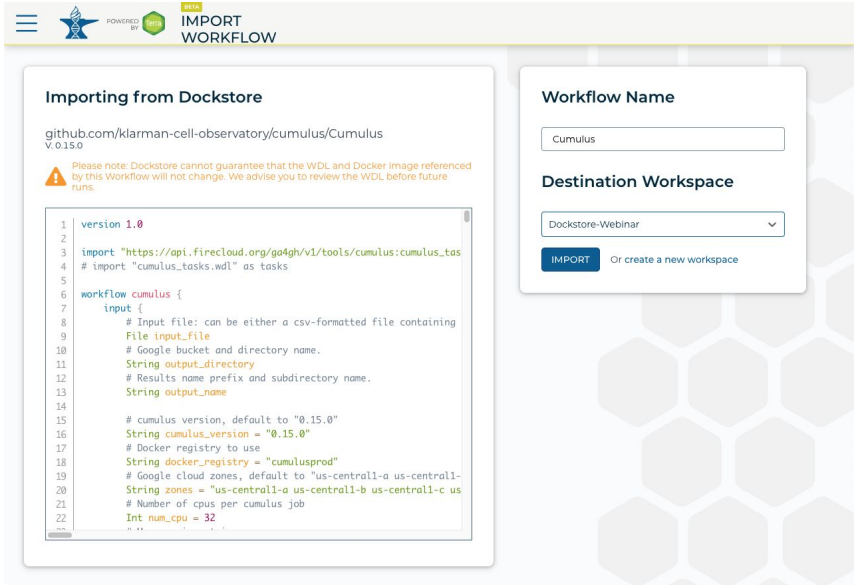
Partner Platforms

- DNAstack
- DNAnexus
- 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



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

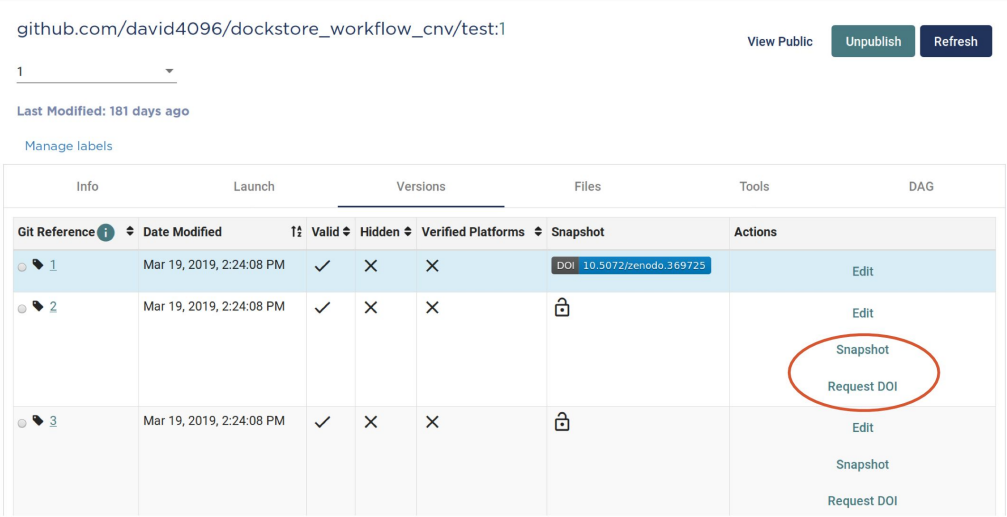
Contributed by: Bo Li & Yiming Yang (Cumulus Team, Broad Institute)

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

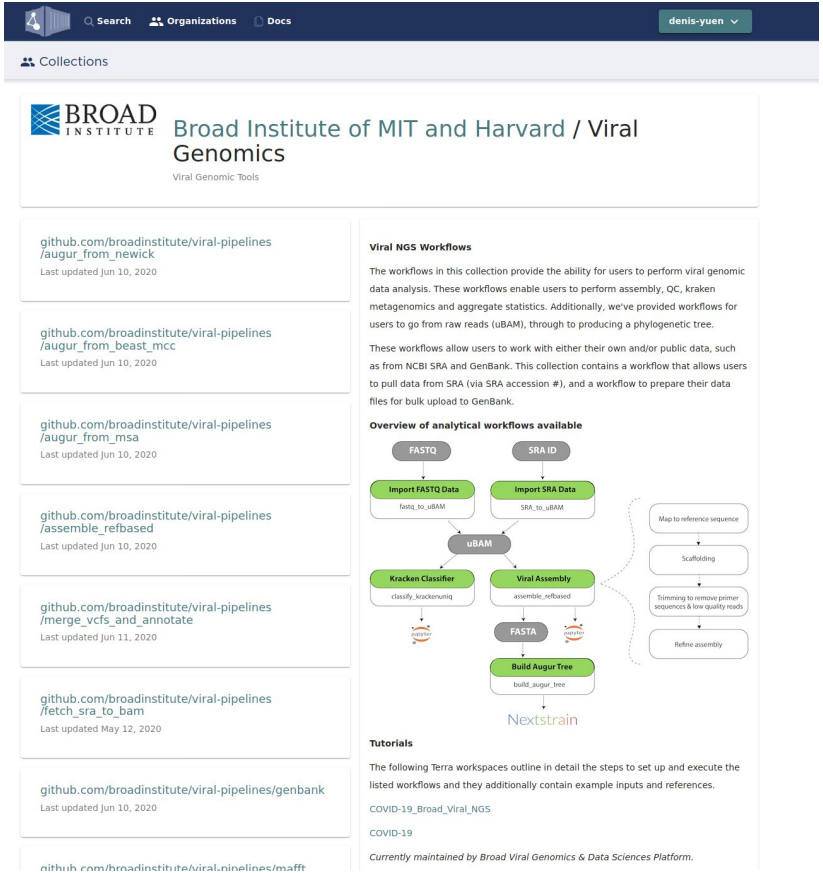
Organizations

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/collections/pgs

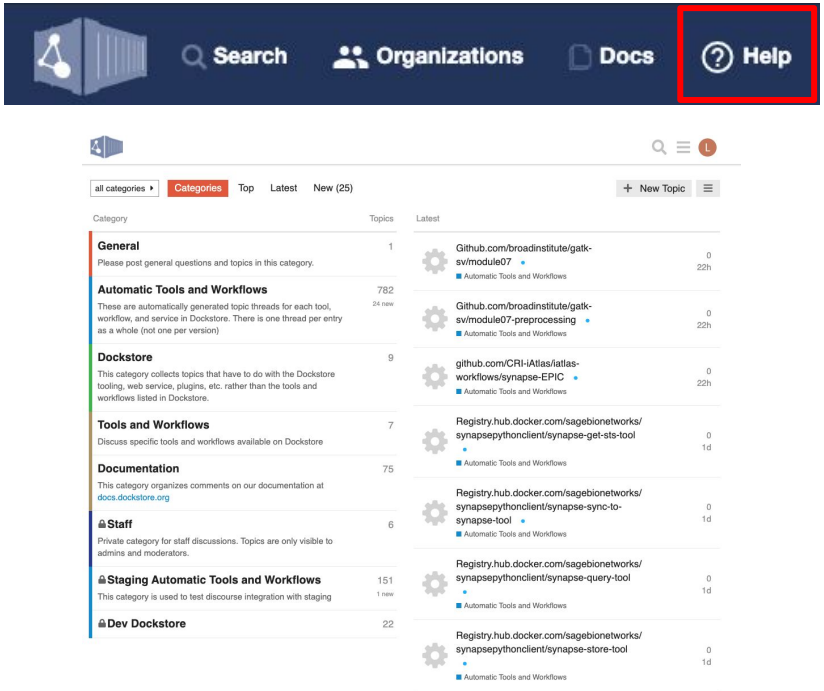
- Contributed by: Daniel Park (Viral Genomics Group, Broad Institute)



Getting Help on Dockstore

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

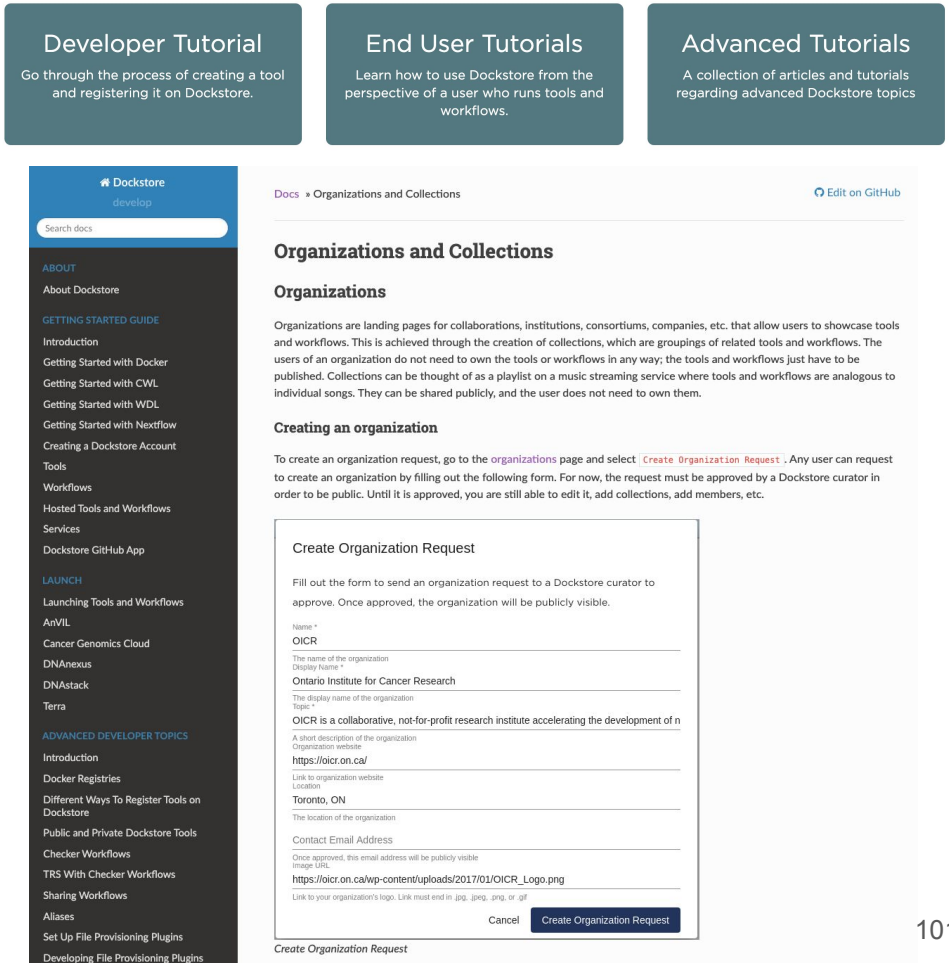
- Topics embedded with each tool, workflow, and documentation page.
- Talk about bioinformatics, workflows, and get help on development



Documentation and Tutorials

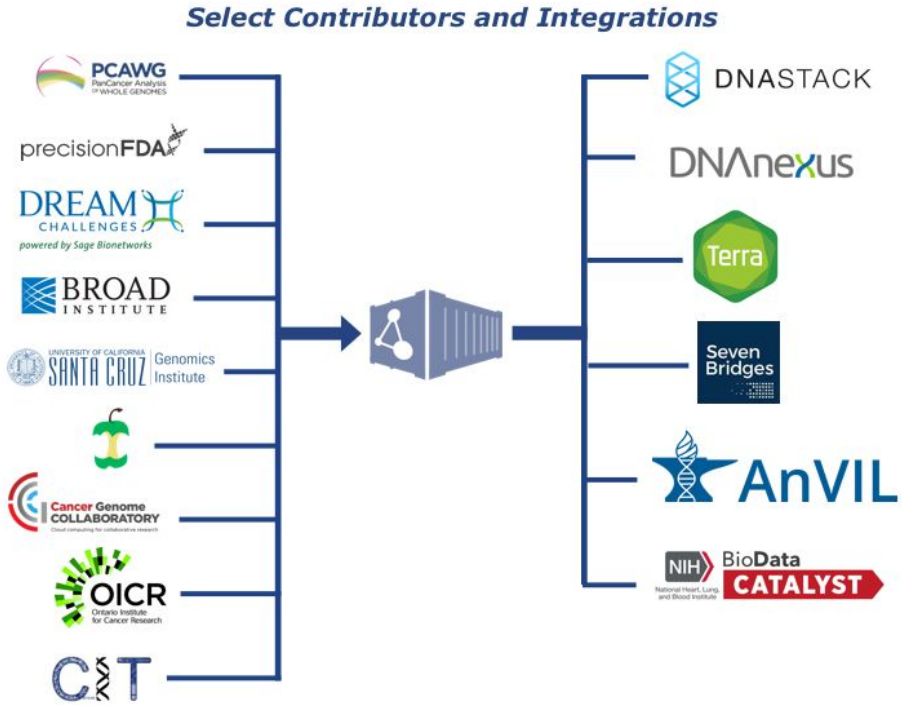
- Example Topics:
  - Launching Tools and Workflows
  - Writing checker workflows
  - Developing File Provisioning Plugins
  - Creating Organizations
  - And many more!

https://docs.dockstore.org/



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 occuring in this space.



The Dockstore Team

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).

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

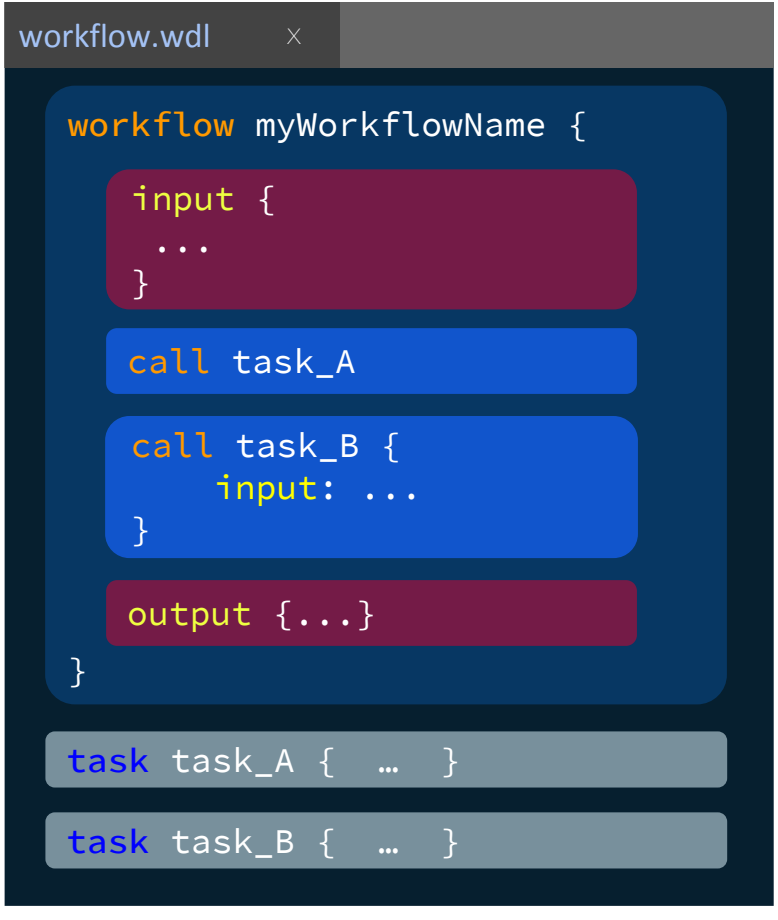
- Collect the results of running each scatter command in an array

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



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

task.wdl

```
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

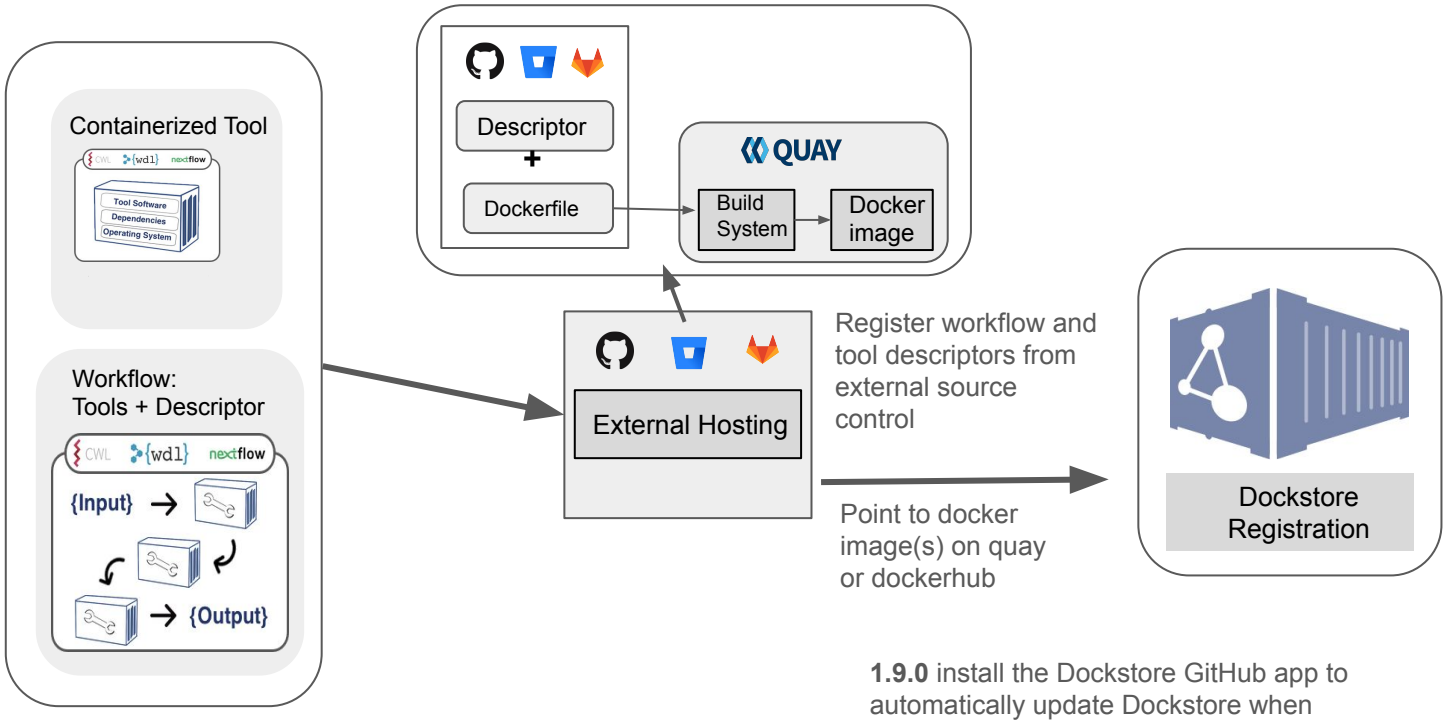
A workflow:

- 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:

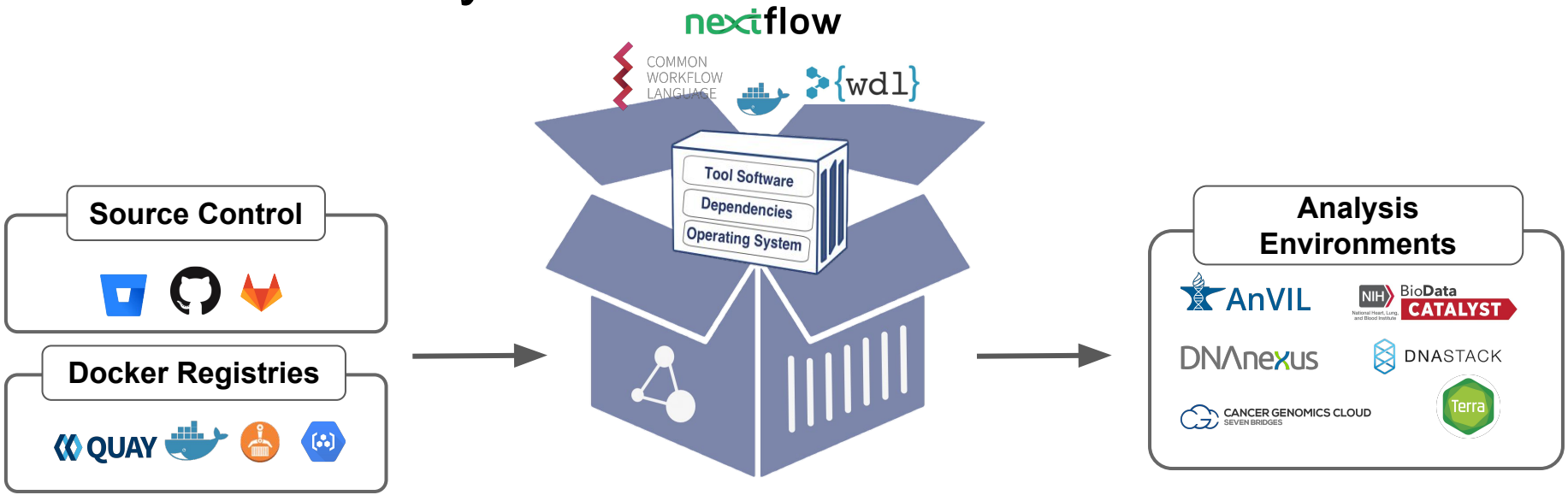
- An `input` section (required if the task will have inputs)
- Non-input declarations (as many as needed, optional)
- A `command` section (required)
- A `runtime` section (optional)
- An `output` section (required if the task will have outputs)
- A `parameter_meta` section (optional)

Ways to Register to Dockstore



1.9.0 install the Dockstore GitHub app to automatically update Dockstore when workflow is updated on GitHub

Dockstore Ecosystem



Store your descriptors and containers and descriptors on your preferred sites

Register these as tools and workflows on Dockstore, allowing for a centralized bioinformatics catalog of resources

Dockstore's launch-with feature enables users to export tools and workflows to a variety of cloud compute platforms

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