Common Workflow Language, v1.0

DOI:
10.6084/m9.figshare.3115156.v2

Document Version
Final published version

Common Workflow Language (CWL) Command Line Tool Description, v1.0

This version:

- https://w3id.org/cwl/v1.0/ (https://w3id.org/cwl/v1.0/)

Current version:

- https://w3id.org/cwl/ (https://w3id.org/cwl/)

Authors:

- Peter Amstutz peter.amstutz@curoverse.com (mailto:peter.amstutz@curoverse.com), Arvados Project, Curoverse
- Michael R. Crusoe michael.crusoe@gmail.com (mailto:michael.crusoe@gmail.com), Common Workflow Language project
- Nebojša Tijanić nebojsa.tijanic@sbgenomics.com (mailto:nebojsa.tijanic@sbgenomics.com), Seven Bridges Genomics

Contributors:

- Brad Chapman bchapman@hsph.harvard.edu (mailto:bchapman@hsph.harvard.edu), Harvard Chan School of Public Health
- John Chilton jmchilton@gmail.com (mailto:jmchilton@gmail.com), Galaxy Project, Pennsylvania State University
- Michael Heuer heuermh@berkeley.edu, (mailto:heuermh@berkeley.edu,),UC Berkeley AMPLab
- Andrey Kartashov Andrey.Kartashov@cchmc.org (mailto:Andrey.Kartashov@cchmc.org), Cincinnati Children's Hospital
- Dan Leehr dan.leehr@duke.edu (mailto:dan.leehr@duke.edu), Duke University
- Hervé Ménager herve.menager@gmail.com (mailto:herve.menager@gmail.com), Institut Pasteur
- Maya Nedeljkovich maja.nedeljkovic@sbgenomics.com (mailto:maja.nedeljkovic@sbgenomics.com), Seven Bridges Genomics
- Matt Scales mscales@icr.ac.uk (mailto:mscales@icr.ac.uk), Institute of Cancer Research, London
- Stian Soiland-Reyes soiland-reyes@cs.manchester.ac.uk (mailto:soiland-reyes@cs.manchester.ac.uk), University of Manchester
- Luka Stojanovic luka.stojanovic@sbgenomics.com (mailto:luka.stojanovic@sbgenomics.com), Seven Bridges Genomics

Abstract

A Command Line Tool is a non-interactive executable program that reads some input, performs a computation, and terminates after producing some output. Command line programs are a flexible unit of code sharing and reuse, unfortunately the syntax and input/output semantics among command line programs is extremely heterogeneous. A common layer for describing the syntax and semantics of programs can reduce this incidental complexity by providing a consistent way to connect programs together. This specification defines the Common Workflow Language (CWL) Command Line Tool Description, a vendor-neutral standard for describing the syntax and input/output semantics of command line programs.

Status of this document

This document is the product of the Common Workflow Language working group (https://groups.google.com/forum/#!forum/common-workflow-language). The latest version of this document is available in the "v1.0" directory at

https://github.com/common-workflow-language/common-workflow-language (https://github.com/common-workflow-language/common-workflow-language)

Table of contents

1. Introduction

The Common Workflow Language (CWL) working group is an informal, multi-vendor working group consisting of various organizations and individuals that have an interest in portability of data analysis workflows. The goal is to create specifications like this one that enable data scientists to describe analysis tools and workflows that are powerful, easy to use, portable, and support reproducibility.

1.1 Introduction to v1.0

This specification represents the first full release from the CWL group. Since draft-3, version 1.0 introduces the following changes and additions:

- The Directory type.
- Syntax simplifcations: denoted by the `map<>` syntax. Example: inputs contains a list of items, each with an id. Now one can specify a mapping of that identifier to the corresponding `CommandInputParamater` .

```
inputs:
 - id: one
   type: string
   doc: First input parameter
 - id: two
   type: int
   doc: Second input parameter
```

  can be

```
inputs:
 one:
  type: string
  doc: First input parameter
 two:
  type: int
  doc: Second input parameter
```

- InitialWorkDirRequirement: list of files and subdirectories to be present in the output directory prior to execution.
- Shortcuts for specifying the standard output and/or error streams as a (streamable) File output.
- SoftwareRequirement for describing software dependencies of a tool.
- The common `description` field has been renamed to `doc` .

1.2 Errata

Post v1.0 release changes to the spec.

- 13 July 2016: Mark `baseCommand` as optional and update descriptive text.

1.3 Purpose

Standalone programs are a flexible and interoperable form of code reuse. Unlike monolithic applications, applications and analysis workflows which are composed of multiple separate programs can be written in multiple languages and execute concurrently on multiple hosts. However, POSIX does not dictate computer-readable grammar or semantics for program input and output, resulting in extremely heterogeneous command line grammar and input/output semantics among

program. This is a particular problem in distributed computing (multi-node compute clusters) and virtualized environments (such as Docker containers) where it is often necessary to provision resources such as input files before executing the program.

Often this gap is filled by hard coding program invocation and implicitly assuming requirements will be met, or abstracting program invocation with wrapper scripts or descriptor documents. Unfortunately, where these approaches are application or platform specific it creates a significant barrier to reproducibility and portability, as methods developed for one platform must be manually ported to be used on new platforms. Similarly it creates redundant work, as wrappers for popular tools must be rewritten for each application or platform in use.

The Common Workflow Language Command Line Tool Description is designed to provide a common standard description of grammar and semantics for invoking programs used in data-intensive fields such as Bioinformatics, Chemistry, Physics, Astronomy, and Statistics. This specification defines a precise data and execution model for Command Line Tools that can be implemented on a variety of computing platforms, ranging from a single workstation to cluster, grid, cloud, and high performance computing platforms.

1.4 References to other specifications

Javascript Object Notation (JSON): http://json.org (http://json.org)

JSON Linked Data (JSON-LD): http://json-ld.org (http://json-ld.org)

YAML: http://yaml.org (http://yaml.org)

Avro: https://avro.apache.org/docs/1.8.1/spec.html (https://avro.apache.org/docs/1.8.1/spec.html)

Uniform Resource Identifier (URI) Generic Syntax: https://tools.ietf.org/html/rfc3986 (https://tools.ietf.org/html/rfc3986))

Internationalized Resource Identifiers (IRIs): https://tools.ietf.org/html/rfc3987 (https://tools.ietf.org/html/rfc3987)

Portable Operating System Interface (POSIX.1-2008): http://pubs.opengroup.org/onlinepubs/9699919799/ (http://pubs.opengroup.org/onlinepubs/9699919799/)

Resource Description Framework (RDF): http://www.w3.org/RDF/ (http://www.w3.org/RDF/)

1.5 Scope

This document describes CWL syntax, execution, and object model. It is not intended to document a CWL specific implementation, however it may serve as a reference for the behavior of conforming implementations.

1.6 Terminology

The terminology used to describe CWL documents is defined in the Concepts section of the specification. The terms defined in the following list are used in building those definitions and in describing the actions of a CWL implementation:

may: Conforming CWL documents and CWL implementations are permitted but not required to behave as described.

must: Conforming CWL documents and CWL implementations are required to behave as described; otherwise they are in error.

error: A violation of the rules of this specification; results are undefined. Conforming implementations may detect and report an error and may recover from it.

fatal error: A violation of the rules of this specification; results are undefined. Conforming implementations must not continue to execute the current process and may report an error.

at user option: Conforming software may or must (depending on the modal verb in the sentence) behave as described; if it does, it must provide users a means to enable or disable the behavior described.

deprecated: Conforming software may implement a behavior for backwards compatibility. Portable CWL documents should not rely on deprecated behavior. Behavior marked as deprecated may be removed entirely from future revisions of the CWL specification.

2. Data model

2.1 Data concepts

An **object** is a data structure equivalent to the "object" type in JSON, consisting of a unordered set of name/value pairs (referred to here as **fields**) and where the name is a string and the value is a string, number, boolean, array, or object.

A **document** is a file containing a serialized object, or an array of objects.

A **process** is a basic unit of computation which accepts input data, performs some computation, and produces output data. Examples include CommandLineTools, Workflows, and ExpressionTools.

An **input object** is an object describing the inputs to an invocation of a process.

An **output object** is an object describing the output resulting from an invocation of a process.

An **input schema** describes the valid format (required fields, data types) for an input object.

An **output schema** describes the valid format for an output object.

Metadata is information about workflows, tools, or input items.

2.2 Syntax

CWL documents must consist of an object or array of objects represented using JSON or YAML syntax. Upon loading, a CWL implementation must apply the preprocessing steps described in the Semantic Annotations for Linked Avro Data (SALAD) Specification (SchemaSalad.html). An implementation may formally validate the structure of a CWL document using SALAD schemas located at https://github.com/common-workflow-language/common-workflow-language/tree/master/v1.0 (https://github.com/common-workflow-language/common-workflow-language/tree/master/v1.0)

2.3 Identifiers

If an object contains an `id` field, that is used to uniquely identify the object in that document. The value of the `id` field must be unique over the entire document. Identifiers may be resolved relative to either the document base and/or other identifiers following the rules are described in the Schema Salad specification (SchemaSalad.html#Identifier_resolution).

An implementation may choose to only honor references to object types for which the `id` field is explicitly listed in this specification.

2.4 Document preprocessing

An implementation must resolve $import (SchemaSalad.html#Import) and $include (SchemaSalad.html#Import) directives as described in the Schema Salad specification (SchemaSalad.html).

Another transformation defined in Schema salad is simplification of data type definitions. Type `<T>` ending with `?` should be transformed to `[<T>, "null"]` . Type `<T>` ending with `[]` should be transformed to `{"type": "array", "items": <T>}`

2.5 Extensions and metadata

Input metadata (for example, a lab sample identifier) may be represented within a tool or workflow using input parameters which are explicitly propagated to output. Future versions of this specification may define additional facilities for working with input/output metadata.

Implementation extensions not required for correct execution (for example, fields related to GUI presentation) and metadata about the tool or workflow itself (for example, authorship for use in citations) may be provided as additional fields on any object. Such extensions fields must use a namespace prefix listed in the `$namespaces` section of the document as described in the Schema Salad specification (SchemaSalad.html#Explicit_context).

Implementation extensions which modify execution semantics must be listed in the `requirements` field.

3. Execution model

3.1 Execution concepts

A **parameter** is a named symbolic input or output of process, with an associated datatype or schema. During execution, values are assigned to parameters to make the input object or output object used for concrete process invocation.

A **CommandLineTool** is a process characterized by the execution of a standalone, non-interactive program which is invoked on some input, produces output, and then terminates.

A **workflow** is a process characterized by multiple subprocess steps, where step outputs are connected to the inputs of downstream steps to form a directed acylic graph, and independent steps may run concurrently.

A **runtime environment** is the actual hardware and software environment when executing a command line tool. It includes, but is not limited to, the hardware architecture, hardware resources, operating system, software runtime (if applicable, such as the specific Python interpreter or the specific Java virtual machine), libraries, modules, packages, utilities, and data files required to run the tool.

A **workflow platform** is a specific hardware and software implementation capable of interpreting CWL documents and executing the processes specified by the document. The responsibilities of the workflow platform may include scheduling process invocation, setting up the necessary runtime environment, making input data available, invoking the tool process, and collecting output.

A workflow platform may choose to only implement the Command Line Tool Description part of the CWL specification.

It is intended that the workflow platform has broad leeway outside of this specification to optimize use of computing resources and enforce policies not covered by this specification. Some areas that are currently out of scope for CWL specification but may be handled by a specific workflow platform include:

- Data security and permissions
- Scheduling tool invocations on remote cluster or cloud compute nodes.
- Using virtual machines or operating system containers to manage the runtime (except as described in DockerRequirement (CommandLineTool.html#DockerRequirement)).
- Using remote or distributed file systems to manage input and output files.
- Transforming file paths.
- Determining if a process has previously been executed, and if so skipping it and reusing previous results.
- Pausing, resuming or checkpointing processes or workflows.

Conforming CWL processes must not assume anything about the runtime environment or workflow platform unless explicitly declared though the use of process requirements.

3.2 Generic execution process

The generic execution sequence of a CWL process (including workflows and command line line tools) is as follows.

1. Load, process and validate a CWL document, yielding a process object.
2. Load input object.
3. Validate the input object against the `inputs` schema for the process.
4. Validate process requirements are met.
5. Perform any further setup required by the specific process type.
6. Execute the process.
7. Capture results of process execution into the output object.
8. Validate the output object against the `outputs` schema for the process.
9. Report the output object to the process caller.

3.3 Requirements and hints

A **process requirement** modifies the semantics or runtime environment of a process. If an implementation cannot satisfy all requirements, or a requirement is listed which is not recognized by the implementation, it is a fatal error and the implementation must not attempt to run the process, unless overridden at user option.

A **hint** is similar to a requirement; however, it is not an error if an implementation cannot satisfy all hints. The implementation may report a warning if a hint cannot be satisfied.

Requirements are inherited. A requirement specified in a Workflow applies to all workflow steps; a requirement specified on a workflow step will apply to the process implementation of that step and any of its substeps.

If the same process requirement appears at different levels of the workflow, the most specific instance of the requirement is used, that is, an entry in `requirements` on a process implementation such as CommandLineTool will take precedence over an entry in `requirements` specified in a workflow step, and an entry in `requirements` on a workflow step takes precedence over the workflow. Entries in `hints` are resolved the same way.

Requirements override hints. If a process implementation provides a process requirement in `hints` which is also provided in `requirements` by an enclosing workflow or workflow step, the enclosing `requirements` takes precedence.

3.4 Parameter references

Parameter references are denoted by the syntax `$(...)` and may be used in any field permitting the pseudo-type `Expression`, as specified by this document. Conforming implementations must support parameter references. Parameter references use the following subset of Javascript/ECMAScript 5.1 (http://www.ecma-international.org/ecma-262/5.1/) syntax, but they are designed to not require a Javascript engine for evaluation.

In the following BNF grammar, character classes, and grammar rules are denoted in '{}', '-' denotes exclusion from a character class, '(())' denotes grouping, '|' denotes alternates, trailing '*' denotes zero or more repeats, '+' denote one or more repeats, '/' escapes these special characters, and all other characters are literal values.

symbol::	{Unicode alphanumeric}+
singleq::	[' (({character - '} | \'))* ']
doubleq::	[" (({character - "} | \"))* "]
index::	[{decimal digit}+]
segment::	. {symbol} | {singleq} | {doubleq} | {index}
parameter reference::	$({symbol} {segment}*)

Use the following algorithm to resolve a parameter reference:

1. Match the leading symbol as the key
2. Look up the key in the parameter context (described below) to get the current value. It is an error if the key is not found in the parameter context.
3. If there are no subsequent segments, terminate and return current value
4. Else, match the next segment
5. Extract the symbol, string, or index from the segment as the key
6. Look up the key in current value and assign as new current value. If the key is a symbol or string, the current value must be an object. If the key is an index, the current value must be an array or string. It is an error if the key does not match the required type, or the key is not found or out of range.
7. Repeat steps 3-6

The root namespace is the parameter context. The following parameters must be provided:

- `inputs` : The input object to the current Process.
- `self` : A context-specific value. The contextual values for 'self' are documented for specific fields elsewhere in this specification. If a contextual value of 'self' is not documented for a field, it must be 'null'.
- `runtime` : An object containing configuration details. Specific to the process type. An implementation may provide opaque strings for any or all fields of `runtime`. These must be filled in by the platform after processing the Tool but before actual execution. Parameter references and expressions may only use the literal string value of the field and must not perform computation on the contents, except where noted otherwise.

If the value of a field has no leading or trailing non-whitespace characters around a parameter reference, the effective value of the field becomes the value of the referenced parameter, preserving the return type.

If the value of a field has non-whitespace leading or trailing characters around a parameter reference, it is subject to string interpolation. The effective value of the field is a string containing the leading characters, followed by the string value of the parameter reference, followed by the trailing characters. The string value of the parameter reference is its textual JSON representation with the following rules:

- Leading and trailing quotes are stripped from strings
- Objects entries are sorted by key

Multiple parameter references may appear in a single field. This case must be treated as a string interpolation. After interpolating the first parameter reference, interpolation must be recursively applied to the trailing characters to yield the final string value.

3.5 Expressions

An expression is a fragment of Javascript/ECMAScript 5.1 (http://www.ecma-international.org/ecma-262/5.1/) code evaluated by the workflow platform to affect the inputs, outputs, or behavior of a process. In the generic execution sequence, expressions may be evaluated during step 5 (process setup), step 6 (execute process), and/or step 7 (capture output). Expressions are distinct from regular processes in that they are intended to modify the behavior of the workflow itself rather than perform the primary work of the workflow.

To declare the use of expressions, the document must include the process requirement `InlineJavascriptRequirement`. Expressions may be used in any field permitting the pseudo-type `Expression`, as specified by this document.

Expressions are denoted by the syntax `$(...)` or `${...}`. A code fragment wrapped in the `$(...)` syntax must be evaluated as a ECMAScript expression (http://www.ecma-international.org/ecma-262/5.1/#sec-11). A code fragment wrapped in the `${...}` syntax must be evaluated as a EMACScript function body (http://www.ecma-international.org/ecma-262/5.1/#sec-13) for an anonymous, zero-argument function. Expressions must return a valid JSON data type: one of null, string, number, boolean, array, object. Other return values must result in a `permanentFailure`. Implementations must permit any syntactically valid Javascript and account for nesting of parenthesis or braces and that strings that may contain parenthesis or braces when scanning for expressions.

The runtime must include any code defined in the "expressionLib" field of InlineJavascriptRequirement prior to executing the actual expression.

Before executing the expression, the runtime must initialize as global variables the fields of the parameter context described above.

The effective value of the field after expression evaluation follows the same rules as parameter references discussed above. Multiple expressions may appear in a single field.

Expressions must be evaluated in an isolated context (a "sandbox") which permits no side effects to leak outside the context. Expressions also must be evaluated in Javascript strict mode (http://www.ecma-international.org/ecma-262/5.1/#sec-4.2.2).

The order in which expressions are evaluated is undefined except where otherwise noted in this document.

An implementation may choose to implement parameter references by evaluating as a Javascript expression. The results of evaluating parameter references must be identical whether implemented by Javascript evaluation or some other means.

Implementations may apply other limits, such as process isolation, timeouts, and operating system containers/jails to minimize the security risks associated with running untrusted code embedded in a CWL document.

Exceptions thrown from an exception must result in a `permanentFailure` of the process.

3.6 Executing CWL documents as scripts

By convention, a CWL document may begin with `#!/usr/bin/env cwl-runner` and be marked as executable (the POSIX "+x" permission bits) to enable it to be executed directly. A workflow platform may support this mode of operation; if so, it must provide `cwl-runner` as an alias for the platform's CWL implementation.

A CWL input object document may similarly begin with `#!/usr/bin/env cwl-runner` and be marked as executable. In this case, the input object must include the field `cwl:tool` supplying an IRI to the default CWL document that should be executed using the fields of the input object as input parameters.

3.7 Discovering CWL documents on a local filesystem

To discover CWL documents look in the following locations:

`/usr/share/commonwl/`

`/usr/local/share/commonwl/`

`$XDG_DATA_HOME/commonwl/` (usually `$HOME/.local/share/commonwl`)

`$XDF_DATA_HOME` is from the XDG Base Directory Specification (http://standards.freedesktop.org/basedir-spec/basedir-spec-0.6.html)

4. Running a Command

To accommodate the enormous variety in syntax and semantics for input, runtime environment, invocation, and output of arbitrary programs, a CommandLineTool defines an "input binding" that describes how to translate abstract input parameters to an concrete program invocation, and an "output binding" that describes how to generate output parameters from program output.

4.1 Input binding

The tool command line is built by applying command line bindings to the input object. Bindings are listed either as part of an input parameter using the `inputBinding` field, or separately using the `arguments` field of the CommandLineTool.

The algorithm to build the command line is as follows. In this algorithm, the sort key is a list consisting of one or more numeric or string elements. Strings are sorted lexicographically based on UTF-8 encoding.

1. Collect `CommandLineBinding` objects from `arguments` . Assign a sorting key `[position, i]` where `position` is `CommandLineBinding.position` and `i` is the index in the `arguments` list.
2. Collect `CommandLineBinding` objects from the `inputs` schema and associate them with values from the input object. Where the input type is a record, array, or map, recursively walk the schema and input object, collecting nested `CommandLineBinding` objects and associating them with values from the input object.
3. Create a sorting key by taking the value of the `position` field at each level leading to each leaf binding object. If `position` is not specified, it is not added to the sorting key. For bindings on arrays and maps, the sorting key must include the array index or map key following the position. If and only if two bindings have the same sort key, the tie must be broken using the ordering of the field or parameter name immediately containing the leaf binding.
4. Sort elements using the assigned sorting keys. Numeric entries sort before strings.
5. In the sorted order, apply the rules defined in `CommandLineBinding` to convert bindings to actual command line elements.
6. Insert elements from `baseCommand` at the beginning of the command line.

4.2 Runtime environment

All files listed in the input object must be made available in the runtime environment. The implementation may use a shared or distributed file system or transfer files via explicit download to the host. Implementations may choose not to provide access to files not explicitly specified in the input object or process requirements.

Output files produced by tool execution must be written to the **designated output directory**. The initial current working directory when executing the tool must be the designated output directory.

Files may also be written to the **designated temporary directory**. This directory must be isolated and not shared with other processes. Any files written to the designated temporary directory may be automatically deleted by the workflow platform immediately after the tool terminates.

For compatibility, files may be written to the **system temporary directory** which must be located at `/tmp` . Because the system temporary directory may be shared with other processes on the system, files placed in the system temporary directory are not guaranteed to be deleted automatically. A tool must not use the system temporary directory as a backchannel communication with other tools. It is valid for the system temporary directory to be the same as the designated temporary directory.

When executing the tool, the tool must execute in a new, empty environment with only the environment variables described below; the child process must not inherit environment variables from the parent process except as specified or at user option.

- `HOME` must be set to the designated output directory.
- `TMPDIR` must be set to the designated temporary directory.
- `PATH` may be inherited from the parent process, except when run in a container that provides its own `PATH` .
- Variables defined by EnvVarRequirement
- The default environment of the container, such as when using DockerRequirement

An implementation may forbid the tool from writing to any location in the runtime environment file system other than the designated temporary directory, system temporary directory, and designated output directory. An implementation may provide read-only input files, and disallow in-place update of input files. The designated temporary directory, system temporary directory and designated output directory may each reside on different mount points on different file systems.

An implementation may forbid the tool from directly accessing network resources. Correct tools must not assume any network access. Future versions of the specification may incorporate optional process requirements that describe the networking needs of a tool.

The `runtime` section available in parameter references and expressions contains the following fields. As noted earlier, an implementation may perform deferred resolution of runtime fields by providing opaque strings for any or all of the following fields; parameter references and expressions may only use the literal string value of the field and must not perform computation on the contents.

- `runtime.outdir` : an absolute path to the designated output directory
- `runtime.tmpdir` : an absolute path to the designated temporary directory
- `runtime.cores` : number of CPU cores reserved for the tool process
- `runtime.ram` : amount of RAM in mebibytes (2**20) reserved for the tool process
- `runtime.outdirSize` : reserved storage space available in the designated output directory
- `runtime.tmpdirSize` : reserved storage space available in the designated temporary directory

For `cores` , `ram` , `outdirSize` and `tmpdirSize` , if an implementation can't provide the actual number of reserved cores during the expression evaluation time, it should report back the minimal requested amount.

See ResourceRequirement for details on how to describe the hardware resources required by a tool.

The standard input stream and standard output stream may be redirected as described in the `stdin` and `stdout` fields.

4.3 Execution

Once the command line is built and the runtime environment is created, the actual tool is executed.

The standard error stream and standard output stream (unless redirected by setting `stdout` or `stderr`) may be captured by platform logging facilities for storage and reporting.

Tools may be multithreaded or spawn child processes; however, when the parent process exits, the tool is considered finished regardless of whether any detached child processes are still running. Tools must not require any kind of console, GUI, or web based user interaction in order to start and run to completion.

The exit code of the process indicates if the process completed successfully. By convention, an exit code of zero is treated as success and non-zero exit codes are treated as failure. This may be customized by providing the fields `successCodes` , `temporaryFailCodes` , and `permanentFailCodes` . An implementation may choose to default unspecified non-zero exit codes to either `temporaryFailure` or `permanentFailure` .

4.4 Output binding

If the output directory contains a file named "cwl.output.json", that file must be loaded and used as the output object. Otherwise, the output object must be generated by walking the parameters listed in `outputs` and applying output bindings to the tool output. Output bindings are associated with output parameters using the `outputBinding` field. See `CommandOutputBinding` for details.

5. CommandLineTool

This defines the schema of the CWL Command Line Tool Description document.

Fields

field	type	required	description
`inputs`	array<CommandInputParameter> | map<CommandInputParameter.id, CommandInputParameter.type> | map<CommandInputParameter.id, CommandInputParameter>	True	Defines the input parameters of the process. The process is ready to run when all required input parameters are associated with concrete values. Input parameters include a schema for each parameter which is used to validate the input object. It may also be used to build a user interface for constructing the input object.
`outputs`	array<CommandOutputParameter> | map<CommandOutputParameter.id, CommandOutputParameter.type> | map<CommandOutputParameter.id, CommandOutputParameter>	True	Defines the parameters representing the output of the process. May be used to generate and/or validate the output object.
`class`	string	True	
`id`	string	False	The unique identifier for this process object.
`requirements`	array<InlineJavascriptRequirement | SchemaDefRequirement | DockerRequirement | SoftwareRequirement | InitialWorkDirRequirement | EnvVarRequirement | ShellCommandRequirement | ResourceRequirement>	False	Declares requirements that apply to either the runtime environment or the workflow engine that must be met in order to execute this process. If an implementation cannot satisfy all requirements, or a requirement is listed which is not recognized by the implementation, it is a fatal error and the implementation must not attempt to run the process, unless overridden at user option.
`hints`	array<Any>	False	Declares hints applying to either the runtime environment or the workflow engine that may be helpful in executing this process. It is not an error if an implementation cannot satisfy all hints, however the implementation may report a warning.
`label`	string	False	A short, human-readable label of this process object.

`doc`	string	False	A long, human-readable description of this process object.
`cwlVersion`	CWLVersion	False	CWL document version. Always required at the document root. Not required for a Process embedded inside another Process.
`baseCommand`	string | array<string>	False	Specifies the program to execute. If an array, the first element of the array is the command to execute, and subsequent elements are mandatory command line arguments. The elements in `baseCommand` must appear before any command line bindings from `inputBinding` or `arguments`. If `baseCommand` is not provided or is an empty array, the first element of the command line produced after processing `inputBinding` or `arguments` must be used as the program to execute. If the program includes a path separator character it must be an absolute path, otherwise it is an error. If the program does not include a path separator, search the `$PATH` variable in the runtime environment of the workflow runner find the absolute path of the executable.
`arguments`	array<string | Expression | CommandLineBinding>	False	Command line bindings which are not directly associated with input parameters.
`stdin`	string | Expression	False	A path to a file whose contents must be piped into the command's standard input stream.
`stderr`	string | Expression	False	Capture the command's standard error stream to a file written to the designated output directory. If `stderr` is a string, it specifies the file name to use. If `stderr` is an expression, the expression is evaluated and must return a string with the file name to use to capture stderr. If the return value is not a string, or the resulting path contains illegal characters (such as the path separator `/`) it is an error.
`stdout`	string | Expression	False	Capture the command's standard output stream to a file written to the designated output directory. If `stdout` is a string, it specifies the file name to use. If `stdout` is an expression, the expression is evaluated and must return a string with the file name to use to capture stdout. If the return value is not a string, or the resulting path contains illegal characters (such as the path separator `/`) it is an error.
`successCodes`	array<int>	False	Exit codes that indicate the process completed successfully.
`temporaryFailCodes`	array<int>	False	Exit codes that indicate the process failed due to a possibly temporary condition, where executing the process with the same runtime environment and inputs may produce different results.
`permanentFailCodes`	array<int>	False	Exit codes that indicate the process failed due to a permanent logic error, where executing the process with the same runtime environment and same inputs is expected to always fail.

5.1 CommandInputParameter

An input parameter for a CommandLineTool.

Fields

field	type	required	description
`id`	string	True	The unique identifier for this parameter object.
`label`	string	False	A short, human-readable label of this object.
`secondaryFiles`	string | Expression | array<string | Expression>	False	Only valid when `type: File` or is an array of `items: File`. Describes files that must be included alongside the primary file(s). If the value is an expression, the value of `self` in the expression must be the primary input or output File to which this binding applies. If the value is a string, it specifies that the following pattern should be applied to the primary file: 1. If string begins with one or more caret `^` characters, for each caret, remove the last file extension from the path (the last period `.` and all following characters). If there are no file extensions, the path is unchanged. 2. Append the remainder of the string to the end of the file path.
`format`	string | array<string> | Expression	False	Only valid when `type: File` or is an array of `items: File`. For input parameters, this must be one or more IRIs of concept nodes that represents file formats which are allowed as input to this parameter, preferrably defined within an ontology. If no ontology is available, file formats may be tested by exact match. For output parameters, this is the file format that will be assigned to the output parameter.
`streamable`	boolean	False	Only valid when `type: File` or is an array of `items: File`. A value of `true` indicates that the file is read or written sequentially without seeking. An implementation may use this flag to indicate whether it is valid to stream file contents using a named pipe. Default: `false`.
`doc`	string | array<string>	False	A documentation string for this type, or an array of strings which should be concatenated.
`inputBinding`	CommandLineBinding	False	Describes how to handle the inputs of a process and convert them into a concrete form for execution, such as command line parameters.
`default`	Any	False	The default value for this parameter if not provided in the input object.
`type`	CWLType | CommandInputRecordSchema | CommandInputEnumSchema | CommandInputArraySchema | string | array<CWLType | CommandInputRecordSchema | CommandInputEnumSchema | CommandInputArraySchema | string>	False	Specify valid types of data that may be assigned to this parameter.

5.1.1 Expression

'Expression' is not a real type. It indicates that a field must allow runtime parameter references. If InlineJavascriptRequirement is declared and supported by the platform, the field must also allow Javascript expressions.

Symbols

symbol	description
`ExpressionPlaceholder`	

5.1.2 CommandLineBinding

When listed under `inputBinding` in the input schema, the term "value" refers to the the corresponding value in the input object. For binding objects listed in `CommandLineTool.arguments`, the term "value" refers to the effective value after evaluating `valueFrom`.

The binding behavior when building the command line depends on the data type of the value. If there is a mismatch between the type described by the input schema and the effective value, such as resulting from an expression evaluation, an implementation must use the data type of the effective value.

- **string**: Add `prefix` and the string to the command line.
- **number**: Add `prefix` and decimal representation to command line.
- **boolean**: If true, add `prefix` to the command line. If false, add nothing.

- **File**: Add `prefix` and the value of `File.path` to the command line.
- **array**: If `itemSeparator` is specified, add `prefix` and the join the array into a single string with `itemSeparator` separating the items. Otherwise first add `prefix`, then recursively process individual elements.
- **object**: Add `prefix` only, and recursively add object fields for which `inputBinding` is specified.
- **null**: Add nothing.

Fields

field	type	required	description
`loadContents`	boolean	False	Only valid when `type: File` or is an array of `items: File`. Read up to the first 64 KiB of text from the file and place it in the "contents" field of the file object for use by expressions.
`position`	int	False	The sorting key. Default position is 0.
`prefix`	string	False	Command line prefix to add before the value.
`separate`	boolean	False	If true (default), then the prefix and value must be added as separate command line arguments; if false, prefix and value must be concatenated into a single command line argument.
`itemSeparator`	string	False	Join the array elements into a single string with the elements separated by by `itemSeparator`.
`valueFrom`	string | Expression	False	If `valueFrom` is a constant string value, use this as the value and apply the binding rules above. If `valueFrom` is an expression, evaluate the expression to yield the actual value to use to build the command line and apply the binding rules above. If the inputBinding is associated with an input parameter, the value of `self` in the expression will be the value of the input parameter. When a binding is part of the `CommandLineTool.arguments` field, the `valueFrom` field is required.
`shellQuote`	boolean	False	If `ShellCommandRequirement` is in the requirements for the current command, this controls whether the value is quoted on the command line (default is true). Use `shellQuote: false` to inject metacharacters for operations such as pipes.

5.1.3 Any

The **Any** type validates for any non-null value.

Symbols

symbol	description
`Any`	

5.1.4 CWLType

Extends primitive types with the concept of a file and directory as a builtin type.

Symbols

symbol	description
`null`	no value
`boolean`	a binary value
`int`	32-bit signed integer
`long`	64-bit signed integer
`float`	single precision (32-bit) IEEE 754 floating-point number
`double`	double precision (64-bit) IEEE 754 floating-point number
`string`	Unicode character sequence
`File`	A File object
`Directory`	A Directory object

5.1.5 File

Represents a file (or group of files if `secondaryFiles` is specified) that must be accessible by tools using standard POSIX file system call API such as open(2) and read(2).

Fields

field	type	required	description
`class`	File_class	True	Must be `File` to indicate this object describes a file.
`location`	string	False	An IRI that identifies the file resource. This may be a relative reference, in which case it must be resolved using the base IRI of the document. The location may refer to a local or remote resource; the implementation must use the IRI to retrieve file content. If an implementation is unable to retrieve the file content stored at a remote resource (due to unsupported protocol, access denied, or other issue) it must signal an error. If the `location` field is not provided, the `contents` field must be provided. The implementation must assign a unique identifier for the `location` field. If the `path` field is provided but the `location` field is not, an implementation may assign the value of the `path` field to `location`, then follow the rules above.
`path`	string	False	The local host path where the File is available when a CommandLineTool is executed. This field must be set by the implementation. The final path component must match the value of `basename`. This field must not be used in any other context. The command line tool being executed must be able to to access the file at `path` using the POSIX `open(2)` syscall. As a special case, if the `path` field is provided but the `location` field is not, an implementation may assign the value of the `path` field to `location`, and remove the `path` field. If the `path` contains POSIX shell metacharacters (http://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html#tag_18_02) (`|`, `&`, `;`, `<`, `>`, `(`, `)`, `$`, `` ` ``, `\`, `"`, `'`, `<space>`, `<tab>`, and `<newline>`) or characters not allowed (http://www.iana.org/assignments/idna-tables-6.3.0/idna-tables-6.3.0.xhtml) for Internationalized Domain Names for Applications (https://tools.ietf.org/html/rfc6452) then implementations may terminate the process with a `permanentFailure`.
`basename`	string	False	The base name of the file, that is, the name of the file without any leading directory path. The base name must not contain a slash `/`. If not provided, the implementation must set this field based on the `location` field by taking the final path component after parsing `location` as an IRI. If `basename` is provided, it is not required to match the value from `location`. When this file is made available to a CommandLineTool, it must be named with `basename`, i.e. the final component of the `path` field must match `basename`.
`dirname`	string	False	The name of the directory containing file, that is, the path leading up to the final slash in the path such that `dirname + '/' + basename == path`. The implementation must set this field based on the value of `path` prior to evaluating parameter references or expressions in a CommandLineTool document. This field must not be used in any other context.
`nameroot`	string	False	The basename root such that `nameroot + nameext == basename`, and `nameext` is empty or begins with a period and contains at most one period. For the purposess of path splitting leading periods on the basename are ignored; a basename of `.cshrc` will have a nameroot of `.cshrc`. The implementation must set this field automatically based on the value of `basename` prior to evaluating parameter references or expressions.
`nameext`	string	False	The basename extension such that `nameroot + nameext == basename`, and `nameext` is empty or begins with a period and contains at most one period. Leading periods on the basename are ignored; a basename of `.cshrc` will have an empty `nameext`. The implementation must set this field automatically based on the value of `basename` prior to evaluating parameter references or expressions.
`checksum`	string	False	Optional hash code for validating file integrity. Currently must be in the form "sha1$ + hexadecimal string" using the SHA-1 algorithm.
`size`	long	False	Optional file size
`secondaryFiles`	array<File | Directory>	False	A list of additional files that are associated with the primary file and must be transferred alongside the primary file. Examples include indexes of the primary file, or external references which must be included when loading primary document. A file object listed in `secondaryFiles` may itself include `secondaryFiles` for which the same rules apply.

`format`	string	False	The format of the file: this must be an IRI of a concept node that represents the file format, preferrably defined within an ontology. If no ontology is available, file formats may be tested by exact match. Reasoning about format compatability must be done by checking that an input file format is the same, `owl:equivalentClass` or `rdfs:subClassOf` the format required by the input parameter. `owl:equivalentClass` is transitive with `rdfs:subClassOf`, e.g. if `<B> owl:equivalentClass <C>` and `<B> owl:subclassOf <A>` then infer `<C> owl:subclassOf <A>`. File format ontologies may be provided in the "$schema" metadata at the root of the document. If no ontologies are specified in `$schema`, the runtime may perform exact file format matches.
`contents`	string	False	File contents literal. Maximum of 64 KiB. If neither `location` nor `path` is provided, `contents` must be non-null. The implementation must assign a unique identifier for the `location` field. When the file is staged as input to CommandLineTool, the value of `contents` must be written to a file. If `loadContents` of `inputBinding` or `outputBinding` is true and `location` is valid, the implementation must read up to the first 64 KiB of text from the file and place it in the "contents" field.

5.1.5.1 Directory

Represents a directory to present to a command line tool.

Fields

field	type	required	description
`class`	Directory_class	True	Must be `Directory` to indicate this object describes a Directory.
`location`	string	False	An IRI that identifies the directory resource. This may be a relative reference, in which case it must be resolved using the base IRI of the document. The location may refer to a local or remote resource. If the `listing` field is not set, the implementation must use the location IRI to retrieve directory listing. If an implementation is unable to retrieve the directory listing stored at a remote resource (due to unsupported protocol, access denied, or other issue) it must signal an error. If the `location` field is not provided, the `listing` field must be provided. The implementation must assign a unique identifier for the `location` field. If the `path` field is provided but the `location` field is not, an implementation may assign the value of the `path` field to `location`, then follow the rules above.
`path`	string	False	The local path where the Directory is made available prior to executing a CommandLineTool. This must be set by the implementation. This field must not be used in any other context. The command line tool being executed must be able to to access the directory at `path` using the POSIX `opendir(2)` syscall. If the `path` contains POSIX shell metacharacters (http://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html#tag_18_02) (`|`, `&`, `;`, `<`, `>`, `(`, `)`, `$`, `` ` ``, `\`, `"`, `'`, `<space>`, `<tab>`, and `<newline>`) or characters not allowed (http://www.iana.org/assignments/idna-tables-6.3.0/idna-tables-6.3.0.xhtml) for Internationalized Domain Names for Applications (https://tools.ietf.org/html/rfc6452) then implementations may terminate the process with a `permanentFailure`.
`basename`	string	False	The base name of the directory, that is, the name of the file without any leading directory path. The base name must not contain a slash `/`. If not provided, the implementation must set this field based on the `location` field by taking the final path component after parsing `location` as an IRI. If `basename` is provided, it is not required to match the value from `location`. When this file is made available to a CommandLineTool, it must be named with `basename`, i.e. the final component of the `path` field must match `basename`.
`listing`	array<File | Directory>	False	List of files or subdirectories contained in this directory. The name of each file or subdirectory is determined by the `basename` field of each `File` or `Directory` object. It is an error if a `File` shares a `basename` with any other entry in `listing`. If two or more `Directory` object share the same `basename`, this must be treated as equivalent to a single subdirectory with the listings recursively merged.

5.1.6 CommandInputRecordSchema

Fields

field	type	required	description
`type`	Record_symbol	True	Must be `record`
`fields`	array<CommandInputRecordField>	False	Defines the fields of the record.
`label`	string	False	A short, human-readable label of this object.

5.1.7 CommandInputRecordField

Fields

field	type	required	description
`name`	string	True	The name of the field
`type`	CWLType | CommandInputRecordSchema | CommandInputEnumSchema | CommandInputArraySchema | string | array<CWLType | CommandInputRecordSchema | CommandInputEnumSchema | CommandInputArraySchema | string>	True	The field type
`doc`	string	False	A documentation string for this field
`inputBinding`	CommandLineBinding	False	
`label`	string	False	A short, human-readable label of this process object.

5.1.7.1 CommandInputEnumSchema

Fields

field	type	required	description
`symbols`	array<string>	True	Defines the set of valid symbols.
`type`	Enum_symbol	True	Must be `enum`
`label`	string	False	A short, human-readable label of this object.
`inputBinding`	CommandLineBinding	False	

5.1.7.2 CommandInputArraySchema

Fields

field	type	required	description
`items`	CWLType | CommandInputRecordSchema | CommandInputEnumSchema | CommandInputArraySchema | string | array<CWLType | CommandInputRecordSchema | CommandInputEnumSchema | CommandInputArraySchema | string>	True	Defines the type of the array elements.
`type`	Array_symbol	True	Must be `array`
`label`	string	False	A short, human-readable label of this object.
`inputBinding`	CommandLineBinding	False	

5.2 CommandOutputParameter

An output parameter for a CommandLineTool.

Fields

field	type	required	description
`id`	string	True	The unique identifier for this parameter object.
`label`	string	False	A short, human-readable label of this object.

`secondaryFiles`	string | Expression | array<string | Expression>	False	Only valid when `type: File` or is an array of `items: File`. Describes files that must be included alongside the primary file(s). If the value is an expression, the value of `self` in the expression must be the primary input or output File to which this binding applies. If the value is a string, it specifies that the following pattern should be applied to the primary file: 1. If string begins with one or more caret `^` characters, for each caret, remove the last file extension from the path (the last period `.` and all following characters). If there are no file extensions, the path is unchanged. 2. Append the remainder of the string to the end of the file path.
`format`	string | array<string> | Expression	False	Only valid when `type: File` or is an array of `items: File`. For input parameters, this must be one or more IRIs of concept nodes that represents file formats which are allowed as input to this parameter, preferrably defined within an ontology. If no ontology is available, file formats may be tested by exact match. For output parameters, this is the file format that will be assigned to the output parameter.
`streamable`	boolean	False	Only valid when `type: File` or is an array of `items: File`. A value of `true` indicates that the file is read or written sequentially without seeking. An implementation may use this flag to indicate whether it is valid to stream file contents using a named pipe. Default: `false`.
`doc`	string | array<string>	False	A documentation string for this type, or an array of strings which should be concatenated.
`outputBinding`	CommandOutputBinding	False	Describes how to handle the outputs of a process.
`type`	CWLType | stdout | stderr | CommandOutputRecordSchema | CommandOutputEnumSchema | CommandOutputArraySchema | string | array<CWLType | CommandOutputRecordSchema | CommandOutputEnumSchema | CommandOutputArraySchema | string>	False	Specify valid types of data that may be assigned to this parameter.

5.2.1 stdout

Only valid as a `type` for a `CommandLineTool` output with no `outputBinding` set.

The following

```
outputs:
   an_output_name:
   type: stdout

stdout: a_stdout_file
```

is equivalent to

```
outputs:
  an_output_name:
    type: File
    streamable: true
    outputBinding:
      glob: a_stdout_file

stdout: a_stdout_file
```

If there is no `stdout` name provided, a random filename will be created. For example, the following

```
outputs:
  an_output_name:
    type: stdout
```

is equivalent to

```
outputs:
  an_output_name:
    type: File
    streamable: true
    outputBinding:
      glob: random_stdout_filenameABCDEFG

stdout: random_stdout_filenameABCDEFG
```

Symbols

symbol	description
`stdout`	

5.2.2 stderr

Only valid as a `type` for a `CommandLineTool` output with no `outputBinding` set.

The following

```
outputs:
  an_output_name:
  type: stderr

stderr: a_stderr_file
```

is equivalent to

```
outputs:
  an_output_name:
    type: File
    streamable: true
    outputBinding:
      glob: a_stderr_file

stderr: a_stderr_file
```

If there is no `stderr` name provided, a random filename will be created. For example, the following

```
outputs:
  an_output_name:
    type: stderr
```

is equivalent to

```
outputs:
  an_output_name:
    type: File
    streamable: true
    outputBinding:
      glob: random_stderr_filenameABCDEFG

stderr: random_stderr_filenameABCDEFG
```

Symbols

symbol	description
`stderr`	

5.2.3 CommandOutputBinding

Describes how to generate an output parameter based on the files produced by a CommandLineTool.

The output parameter is generated by applying these operations in the following order:

- glob
- loadContents
- outputEval

Fields

field	type	required	description
`glob`	string | Expression | array<string>	False	Find files relative to the output directory, using POSIX glob(3) pathname matching. If an array is provided, find files that match any pattern in the array. If an expression is provided, the expression must return a string or an array of strings, which will then be evaluated as one or more glob patterns. Must only match and return files which actually exist.
`loadContents`	boolean	False	For each file matched in `glob` , read up to the first 64 KiB of text from the file and place it in the `contents` field of the file object for manipulation by `outputEval` .

`outputEval`	string | Expression	False	Evaluate an expression to generate the output value. If `glob` was specified, the value of `self` must be an array containing file objects that were matched. If no files were matched, `self` must be a zero length array; if a single file was matched, the value of `self` is an array of a single element. Additionally, if `loadContents` is `true`, the File objects must include up to the first 64 KiB of file contents in the `contents` field.

5.2.4 CommandOutputRecordSchema

Fields

field	type	required	description
`type`	Record_symbol	True	Must be `record`
`fields`	array<CommandOutputRecordField>	False	Defines the fields of the record.
`label`	string	False	A short, human-readable label of this object.

5.2.5 CommandOutputRecordField

Fields

field	type	required	description
`name`	string	True	The name of the field
`type`	CWLType | CommandOutputRecordSchema | CommandOutputEnumSchema | CommandOutputArraySchema | string | array<CWLType | CommandOutputRecordSchema | CommandOutputEnumSchema | CommandOutputArraySchema | string>	True	The field type
`doc`	string	False	A documentation string for this field
`outputBinding`	CommandOutputBinding	False	

5.2.5.1 CommandOutputEnumSchema

Fields

field	type	required	description
`symbols`	array<string>	True	Defines the set of valid symbols.
`type`	Enum_symbol	True	Must be `enum`
`label`	string	False	A short, human-readable label of this object.
`outputBinding`	CommandOutputBinding	False	

5.2.5.2 CommandOutputArraySchema

Fields

field	type	required	description
`items`	CWLType | CommandOutputRecordSchema | CommandOutputEnumSchema | CommandOutputArraySchema | string | array<CWLType | CommandOutputRecordSchema | CommandOutputEnumSchema | CommandOutputArraySchema | string>	True	Defines the type of the array elements.
`type`	Array_symbol	True	Must be `array`
`label`	string	False	A short, human-readable label of this object.
`outputBinding`	CommandOutputBinding	False	

5.3 InlineJavascriptRequirement

Indicates that the workflow platform must support inline Javascript expressions. If this requirement is not present, the workflow platform must not perform expression interpolatation.

Fields

field	type	required	description
`class`	string	True	Always 'InlineJavascriptRequirement'
`expressionLib`	array<string>	False	Additional code fragments that will also be inserted before executing the expression code. Allows for function definitions that may be called from CWL expressions.

5.4 SchemaDefRequirement

This field consists of an array of type definitions which must be used when interpreting the `inputs` and `outputs` fields. When a `type` field contain a IRI, the implementation must check if the type is defined in `schemaDefs` and use that definition. If the type is not found in `schemaDefs`, it is an error. The entries in `schemaDefs` must be processed in the order listed such that later schema definitions may refer to earlier schema definitions.

Fields

field	type	required	description
`class`	string	True	Always 'SchemaDefRequirement'
`types`	array<InputRecordSchema | InputEnumSchema | InputArraySchema>	True	The list of type definitions.

5.4.1 InputRecordSchema

Fields

field	type	required	description
`type`	Record_symbol	True	Must be `record`
`fields`	array<InputRecordField>	False	Defines the fields of the record.
`label`	string	False	A short, human-readable label of this object.

5.4.2 InputRecordField

Fields

field	type	required	description
`name`	string	True	The name of the field
`type`	CWLType | InputRecordSchema | InputEnumSchema | InputArraySchema | string | array<CWLType | InputRecordSchema | InputEnumSchema | InputArraySchema | string>	True	The field type
`doc`	string	False	A documentation string for this field
`inputBinding`	CommandLineBinding	False	
`label`	string	False	A short, human-readable label of this process object.

5.4.2.1 InputEnumSchema

Fields

field	type	required	description
`symbols`	array<string>	True	Defines the set of valid symbols.
`type`	Enum_symbol	True	Must be `enum`
`label`	string	False	A short, human-readable label of this object.

`inputBinding`	CommandLineBinding	False

5.4.2.2 InputArraySchema

Fields

field	type	required	description
`items`	CWLType | InputRecordSchema | InputEnumSchema | InputArraySchema | string | array<CWLType | InputRecordSchema | InputEnumSchema | InputArraySchema | string>	True	Defines the type of the array elements.
`type`	Array_symbol	True	Must be `array`
`label`	string	False	A short, human-readable label of this object.
`inputBinding`	CommandLineBinding	False	

5.5 DockerRequirement

Indicates that a workflow component should be run in a Docker (http://docker.com) container, and specifies how to fetch or build the image.

If a CommandLineTool lists `DockerRequirement` under `hints` (or `requirements`), it may (or must) be run in the specified Docker container.

The platform must first acquire or install the correct Docker image as specified by `dockerPull`, `dockerImport`, `dockerLoad` or `dockerFile`.

The platform must execute the tool in the container using `docker run` with the appropriate Docker image and tool command line.

The workflow platform may provide input files and the designated output directory through the use of volume bind mounts. The platform may rewrite file paths in the input object to correspond to the Docker bind mounted locations.

When running a tool contained in Docker, the workflow platform must not assume anything about the contents of the Docker container, such as the presence or absence of specific software, except to assume that the generated command line represents a valid command within the runtime environment of the container.

Interaction with other requirements

If EnvVarRequirement is specified alongside a DockerRequirement, the environment variables must be provided to Docker using `--env` or `--env-file` and interact with the container's preexisting environment as defined by Docker.

Fields

field	type	required	description
`class`	string	True	Always 'DockerRequirement'
`dockerPull`	string	False	Specify a Docker image to retrieve using `docker pull`.
`dockerLoad`	string	False	Specify a HTTP URL from which to download a Docker image using `docker load`.
`dockerFile`	string	False	Supply the contents of a Dockerfile which will be built using `docker build`.
`dockerImport`	string	False	Provide HTTP URL to download and gunzip a Docker images using `docker import.
`dockerImageId`	string	False	The image id that will be used for `docker run`. May be a human-readable image name or the image identifier hash. May be skipped if `dockerPull` is specified, in which case the `dockerPull` image id must be used.
`dockerOutputDirectory`	string	False	Set the designated output directory to a specific location inside the Docker container.

5.6 SoftwareRequirement

A list of software packages that should be configured in the environment of the defined process.

Fields

field	type	required	description
`class`	string	True	Always 'SoftwareRequirement'
`packages`	array<SoftwarePackage> | map<SoftwarePackage.package, SoftwarePackage.specs> | map<SoftwarePackage.package, SoftwarePackage>	True	The list of software to be configured.

5.7 SoftwarePackage

Fields

field	type	required	description
`package`	string	True	The common name of the software to be configured.
`version`	array<string>	False	The (optional) version of the software to configured.
`specs`	array<string>	False	Must be one or more IRIs identifying resources for installing or enabling the software. Implementations may provide resolvers which map well-known software spec IRIs to some configuration action. For example, an IRI `https://packages.debian.org/jessie/bowtie` could be resolved with `apt-get install bowtie` . An IRI `https://anaconda.org/bioconda/bowtie` could be resolved with `conda install -c bioconda bowtie` . Tools may also provide IRIs to index entries such as RRID (http://www.identifiers.org/rrid/), such as `http://identifiers.org/rrid/RRID:SCR_005476`

5.8 InitialWorkDirRequirement

Define a list of files and subdirectories that must be created by the workflow platform in the designated output directory prior to executing the command line tool.

Fields

field	type	required	description
`class`	string	True	InitialWorkDirRequirement
`listing`	array<File | Directory | Dirent | string | Expression> | string | Expression	True	The list of files or subdirectories that must be placed in the designated output directory prior to executing the command line tool. May be an expression. If so, the expression return value must validate as `{type: array, items: [File, Directory]}` .

5.8.1 Dirent

Define a file or subdirectory that must be placed in the designated output directory prior to executing the command line tool. May be the result of executing an expression, such as building a configuration file from a template.

Fields

field	type	required	description
`entry`	string | Expression	True	If the value is a string literal or an expression which evaluates to a string, a new file must be created with the string as the file contents. If the value is an expression that evaluates to a `File` object, this indicates the referenced file should be added to the designated output directory prior to executing the tool. If the value is an expression that evaluates to a `Dirent` object, this indicates that the File or Directory in `entry` should be added to the designated output directory with the name in `entryname` . If `writable` is false, the file may be made available using a bind mount or file system link to avoid unnecessary copying of the input file.
`entryname`	string | Expression	False	The name of the file or subdirectory to create in the output directory. If `entry` is a File or Directory, this overrides `basename` . Optional.
`writable`	boolean	False	If true, the file or directory must be writable by the tool. Changes to the file or directory must be isolated and not visible by any other CommandLineTool process. This may be implemented by making a copy of the original file or directory. Default false (files and directories read-only by default).

5.9 EnvVarRequirement

Define a list of environment variables which will be set in the execution environment of the tool. See `EnvironmentDef` for details.

Fields

field	type	required	description
`class`	string	True	Always 'EnvVarRequirement'
`envDef`	array<EnvironmentDef> | map<EnvironmentDef.envName, EnvironmentDef.envValue> | map<EnvironmentDef.envName, EnvironmentDef>	True	The list of environment variables.

5.10 EnvironmentDef

Define an environment variable that will be set in the runtime environment by the workflow platform when executing the command line tool. May be the result of executing an expression, such as getting a parameter from input.

Fields

field	type	required	description
`envName`	string	True	The environment variable name
`envValue`	string | Expression	True	The environment variable value

5.11 ShellCommandRequirement

Modify the behavior of CommandLineTool to generate a single string containing a shell command line. Each item in the argument list must be joined into a string separated by single spaces and quoted to prevent intepretation by the shell, unless `CommandLineBinding` for that argument contains `shellQuote: false`. If `shellQuote: false` is specified, the argument is joined into the command string without quoting, which allows the use of shell metacharacters such as `|` for pipes.

Fields

field	type	required	description
`class`	string	True	Always 'ShellCommandRequirement'

5.12 ResourceRequirement

Specify basic hardware resource requirements.

"min" is the minimum amount of a resource that must be reserved to schedule a job. If "min" cannot be satisfied, the job should not be run.

"max" is the maximum amount of a resource that the job shall be permitted to use. If a node has sufficient resources, multiple jobs may be scheduled on a single node provided each job's "max" resource requirements are met. If a job attempts to exceed its "max" resource allocation, an implementation may deny additional resources, which may result in job failure.

If "min" is specified but "max" is not, then "max" == "min" If "max" is specified by "min" is not, then "min" == "max".

It is an error if max < min.

It is an error if the value of any of these fields is negative.

If neither "min" nor "max" is specified for a resource, an implementation may provide a default.

Fields

field	type	required	description
`class`	string	True	Always 'ResourceRequirement'
`coresMin`	long | string | Expression	False	Minimum reserved number of CPU cores
`coresMax`	int | string | Expression	False	Maximum reserved number of CPU cores
`ramMin`	long | string | Expression	False	Minimum reserved RAM in mebibytes (2**20)
`ramMax`	long | string | Expression	False	Maximum reserved RAM in mebibytes (2**20)
`tmpdirMin`	long | string | Expression	False	Minimum reserved filesystem based storage for the designated temporary directory, in mebibytes (2**20)
`tmpdirMax`	long | string | Expression	False	Maximum reserved filesystem based storage for the designated temporary directory, in mebibytes (2**20)
`outdirMin`	long | string | Expression	False	Minimum reserved filesystem based storage for the designated output directory, in mebibytes (2**20)
`outdirMax`	long | string | Expression	False	Maximum reserved filesystem based storage for the designated output directory, in mebibytes (2**20)

5.13 CWLVersion

Version symbols for published CWL document versions.

Symbols

symbol	description
`draft-2`	
`draft-3.dev1`	
`draft-3.dev2`	
`draft-3.dev3`	
`draft-3.dev4`	

draft-3.dev5
draft-3
draft-4.dev1
draft-4.dev2
draft-4.dev3
v1.0.dev4
v1.0