Common Workflow Language, v1.0

DOI:
10.6084/m9.figshare.3115156.v2

Document Version
Final published version

Link to publication record in Manchester Research Explorer

Citation for published version (APA):

Citing this paper
Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights
Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy
If you believe that this document breaches copyright please refer to the University of Manchester's Takedown Procedures [http://man.ac.uk/04Y6Bo] or contact uml.scholarlycommunications@manchester.ac.uk providing relevant details, so we can investigate your claim.
Common Workflow Language (CWL) Workflow 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 jmclifton@gmail.com (mailto:jmclifton@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
One way to define a workflow is: an analysis task represented by a directed graph describing a sequence of operations that transform an input data set to output. This specification defines the Common Workflow Language (CWL) Workflow description, a vendor-neutral standard for representing workflows intended to be portable across a variety of computing platforms.

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
The products of the CWL working group (including this document) are made available under the terms of the Apache License, version 2.0.

Table of contents
Common Workflow Language (CWL) Workflow Description, v1.0
  Abstract
  Status of this document
  1. Introduction
    1.1 Introduction to v1.0
    1.2 Purpose
    1.3 References to other specifications
    1.4 Scope
    1.5 Terminology
  2. Data model
    2.1 Data concepts
    2.2 Syntax
    2.3 Identifiers
    2.4 Document preprocessing
    2.5 Extensions and metadata
  3. Execution model
    3.1 Execution concepts
    3.2 Generic execution process
    3.3 Requirements and hints
    3.4 Parameter references
    3.5 Expressions
    3.6 Executing CWL documents as scripts
    3.7 Discovering CWL documents on a local filesystem
  4. Workflow
    4.1 WorkflowOutputParameter
      4.1.1 Expression
      4.1.2 CommandOutputBinding
      4.1.3 LinkMergeMethod
      4.1.4 CWLType
      4.1.5 File
      4.1.5.1 Directory
      4.1.6 OutputRecordSchema
      4.1.7 OutputRecordField
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, this draft introduces the following changes and additions:

- The inputs and outputs fields have been renamed in and out.
- Syntax simplifications: denoted by the map<> syntax. Example: in contains a list of items, each with an id. Now one can specify a mapping of that identifier to the corresponding InputParameter.

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

can be

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

- The common field description has been renamed to doc.

1.2 Purpose

The Common Workflow Language Command Line Tool Description express workflows for data-intensive science, such as Bioinformatics, Chemistry, Physics, and Astronomy. This specification is intended to define a data and execution model for Workflows that can be implemented on top of a variety of computing platforms, ranging from an individual workstation to cluster, grid, and high performance computing systems.

1.3 References to other specifications

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

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

YAML: http://yaml.org

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


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

1.4 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.

1.5 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 an 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

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 acyclic 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 using 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 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 or 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 $\{\ldots\}$ or $\{\ldots\}$. A code fragment wrapped in the $\{\ldots\}$ 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 $\{\ldots\}$ syntax must be evaluated as a ECMAScript expression (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)
$XDG_DATA_HOME is from the XDG Base Directory Specification (http://standards.freedesktop.org/basedir-spec/basedir-spec-0.6.html)

4. Workflow

A workflow describes a set of steps and the dependencies between those steps. When a step produces output that will be consumed by a second step, the first step is a dependency of the second step.

When there is a dependency, the workflow engine must execute the preceding step and wait for it to successfully produce output before executing the dependent step. If two steps are defined in the workflow graph that are not directly or indirectly dependent, these steps are independent, and may execute in any order or execute concurrently. A workflow is complete when all steps have been executed.
Dependencies between parameters are expressed using the `source` field on workflow step input parameters and workflow output parameters. The `source` field expresses the dependency of one parameter on another such that when a value is associated with the parameter specified by `source`, that value is propagated to the destination parameter. When all data links inbound to a given step are fulfilled, the step is ready to execute.

**Workflow success and failure**

A completed step must result in one of `success`, `temporaryFailure` or `permanentFailure` states. An implementation may choose to retry a step execution which resulted in `temporaryFailure`. An implementation may choose to either continue running other steps of a workflow, or terminate immediately upon `permanentFailure`.

- If any step of a workflow execution results in `permanentFailure`, then the workflow status is `permanentFailure`.
- If one or more steps result in `temporaryFailure` and all other steps complete `success` or are not executed, then the workflow status is `temporaryFailure`.
- If all workflow steps are executed and complete with `success`, then the workflow status is `success`.

**Extensions**

`ScatterFeatureRequirement` and `SubworkflowFeatureRequirement` are available as standard extensions to core workflow semantics.

**Fields**

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputs</td>
<td>array&lt;InputParameter&gt;</td>
<td>map&lt;InputParameter.id, InputParameter.type&gt;</td>
<td>True</td>
</tr>
<tr>
<td>outputs</td>
<td>array&lt;WorkflowOutputParameter&gt;</td>
<td>map&lt;WorkflowOutputParameter.id, WorkflowOutputParameter.type&gt;</td>
<td>True</td>
</tr>
<tr>
<td>class</td>
<td>string</td>
<td>True</td>
<td>The individual steps that make up the workflow. Each step is executed when all of its input data links are fulfilled. An implementation may choose to execute the steps in a different order than listed and/or execute steps concurrently, provided that dependencies between steps are met.</td>
</tr>
<tr>
<td>steps</td>
<td>array&lt;WorkflowStep&gt;</td>
<td>True</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>string</td>
<td>False</td>
<td>The unique identifier for this process object.</td>
</tr>
<tr>
<td>requirements</td>
<td>array&lt;InlineJavascriptRequirement</td>
<td>SchemaDefRequirement</td>
<td>DockerRequirement (CommandLineTool.html#DockerRequirement)</td>
</tr>
<tr>
<td>hints</td>
<td>array&lt;Any&gt;</td>
<td>False</td>
<td>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.</td>
</tr>
<tr>
<td>label</td>
<td>string</td>
<td>False</td>
<td>A short, human-readable label of this process object.</td>
</tr>
<tr>
<td>doc</td>
<td>string</td>
<td>False</td>
<td>A long, human-readable description of this process object.</td>
</tr>
</tbody>
</table>

4.1 WorkflowOutputParameter
Describe an output parameter of a workflow. The parameter must be connected to one or more parameters defined in the workflow that will provide the value of the output parameter.

**Fields**

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>string</td>
<td>True</td>
<td>The unique identifier for this parameter object.</td>
</tr>
<tr>
<td>label</td>
<td>string</td>
<td>False</td>
<td>A short, human-readable label of this object.</td>
</tr>
</tbody>
</table>
| 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, preferably 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.                           |
| outputSource   | string | array<string> | False | Specifies one or more workflow parameters that supply the value of to the output parameter. |
| linkMerge      | LinkMergeMethod                           | False    | The method to use to merge multiple sources into a single array. If not specified, the default method is "merge_nested". |
| type           | CWLType | OutputRecordSchema | OutputEnumSchema | OutputArraySchema | string | array<CWLType> | OutputRecordSchema | OutputEnumSchema | OutputArraySchema | string> | False | Specify valid types of data that may be assigned to this parameter. |

**4.1.1 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**

<table>
<thead>
<tr>
<th>symbol</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExpressionPlaceHolder</td>
<td></td>
</tr>
</tbody>
</table>

**4.1.2 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
### 4.1.3 LinkMergeMethod

The input link merge method, described in `WorkflowStepInput`.

#### Symbols

<table>
<thead>
<tr>
<th>symbol</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge_nested</td>
<td></td>
</tr>
<tr>
<td>merge_flattened</td>
<td></td>
</tr>
</tbody>
</table>

#### 4.1.4 CWLType

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

#### Symbols

<table>
<thead>
<tr>
<th>symbol</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>no value</td>
</tr>
<tr>
<td>boolean</td>
<td>a binary value</td>
</tr>
<tr>
<td>int</td>
<td>32-bit signed integer</td>
</tr>
<tr>
<td>long</td>
<td>64-bit signed integer</td>
</tr>
<tr>
<td>float</td>
<td>single precision (32-bit) IEEE 754 floating-point number</td>
</tr>
<tr>
<td>double</td>
<td>double precision (64-bit) IEEE 754 floating-point number</td>
</tr>
<tr>
<td>string</td>
<td>Unicode character sequence</td>
</tr>
<tr>
<td>File</td>
<td>A File object</td>
</tr>
<tr>
<td>Directory</td>
<td>A Directory object</td>
</tr>
</tbody>
</table>

### 4.1.5 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

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>File_class</td>
<td>True</td>
<td>Must be File to indicate this object describes a file.</td>
</tr>
<tr>
<td>location</td>
<td>string</td>
<td>False</td>
<td>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.</td>
</tr>
</tbody>
</table>
The basename root such that dirname + '/' + basename == path.

The implementation must set this field based on the value of basename prior to evaluating parameter references or expressions.

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 purposes 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.

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.

Optional hash code for validating file integrity. Currently must be in the form "sha1$ + hexadecimal string" using the SHA-1 algorithm.

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.

The format of the file: this must be an IRI of a concept node that represents the file format, preferably defined within an ontology. If no ontology is available, file formats may be tested by exact match.

Reasoning about format compatibility 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 <A>` and `<C> owl:subClassOf <A>` then infer `<C> owl:subClassOf <B>`.

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.

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.

### 4.1.5.1 Directory

Represents a directory to present to a command line tool.

**Fields**
### 4.1.6 OutputRecordSchema

**Fields**

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Record_symbol</td>
<td>True</td>
<td>Must be record</td>
</tr>
<tr>
<td>fields</td>
<td>array&lt;OutputRecordField&gt;</td>
<td>False</td>
<td>Defines the fields of the record.</td>
</tr>
<tr>
<td>label</td>
<td>string</td>
<td>False</td>
<td>A short, human-readable label of this object.</td>
</tr>
</tbody>
</table>

### 4.1.7 OutputRecordField

**Fields**

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>True</td>
<td>The name of the field</td>
</tr>
<tr>
<td>type</td>
<td>CWLType</td>
<td>OutputRecordSchema</td>
<td>OutputEnumSchema</td>
</tr>
<tr>
<td>doc</td>
<td>string</td>
<td>False</td>
<td>A documentation string for this field</td>
</tr>
</tbody>
</table>

### 4.1.7.1 OutputEnumSchema

**Fields**
### 4.1.7.2 OutputArraySchema

**Fields**

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>items</td>
<td>CWLType</td>
<td>OutputRecordSchema</td>
<td>OutputEnumSchema</td>
</tr>
<tr>
<td>type</td>
<td>Array_symbol</td>
<td></td>
<td></td>
</tr>
<tr>
<td>label</td>
<td>string</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 4.2 WorkflowStep

A workflow step is an executable element of a workflow. It specifies the underlying process implementation (such as CommandLineTool or another Workflow) in the run field and connects the input and output parameters of the underlying process to workflow parameters.

#### Scatter/gather

To use scatter/gather, ScatterFeatureRequirement must be specified in the workflow or workflow step requirements.

A "scatter" operation specifies that the associated workflow step or subworkflow should execute separately over a list of input elements. Each job making up a scatter operation is independent and may be executed concurrently.

The scatter field specifies one or more input parameters which will be scattered. An input parameter may be listed more than once. The declared type of each input parameter is implicitly wrapped in an array for each time it appears in the scatter field. As a result, upstream parameters which are connected to scattered parameters may be arrays.

All output parameter types are also implicitly wrapped in arrays. Each job in the scatter results in an entry in the output array.

If scatter declares more than one input parameter, scatterMethod describes how to decompose the input into a discrete set of jobs.

- **dotproduct** specifies that each of the input arrays are aligned and one element taken from each array to construct each job. It is an error if all input arrays are not the same length.
- **nested_crossproduct** specifies the Cartesian product of the inputs, producing a job for every combination of the scattered inputs. The output must be nested arrays for each level of scattering, in the order that the input arrays are listed in the scatter field.
- **flat_crossproduct** specifies the Cartesian product of the inputs, producing a job for every combination of the scattered inputs. The output arrays must be flattened to a single level, but otherwise listed in the order that the input arrays are listed in the scatter field.

#### Subworkflows

To specify a nested workflow as part of a workflow step, SubworkflowFeatureRequirement must be specified in the workflow or workflow step requirements.

**Fields**

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>string</td>
<td>True</td>
<td>The unique identifier for this workflow step.</td>
</tr>
<tr>
<td>in</td>
<td>array&lt;WorkflowStepInput&gt;</td>
<td>map&lt;WorkflowStepInput.id, WorkflowStepInput.source&gt;</td>
<td>map&lt;WorkflowStepInput.id, WorkflowStepInput&gt;</td>
</tr>
<tr>
<td>out</td>
<td>array&lt;string</td>
<td>WorkflowStepOutput&gt;</td>
<td>True</td>
</tr>
</tbody>
</table>
run \texttt{true} \text{ string | CommandLineTool (CommandLineTool.html\#CommandLineTool) \mid ExpressionTool | Workflow} \text{ True} \text{ Specifies the process to run.}

\textbf{requirements} \texttt{array<InlineJavascriptRequirement | SchemaDefRequirement | DockerRequirement (CommandLineTool.html\#DockerRequirement) | SoftwareRequirement | InitialWorkDirRequirement | EnvVarRequirement (CommandLineTool.html\#EnvVarRequirement) | ShellCommandRequirement (CommandLineTool.html\#ShellCommandRequirement) | ResourceRequirement (CommandLineTool.html\#ResourceRequirement) \mid SubworkflowFeatureRequirement | ScatterFeatureRequirement | MultipleInputFeatureRequirement | StepInputExpressionRequirement>} \text{ False} \text{ Declares requirements that apply to either the runtime environment or the workflow engine that must be met in order to execute this workflow step. 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.}

\textbf{hints} \texttt{array<Any>} \text{ False} \text{ Declares hints applying to either the runtime environment or the workflow engine that may be helpful in executing this workflow step. It is not an error if an implementation cannot satisfy all hints, however the implementation may report a warning.}

\textbf{label} \texttt{string} \text{ False} \text{ A short, human-readable label of this process object.}

\textbf{doc} \texttt{string} \text{ False} \text{ A long, human-readable description of this process object.}

\textbf{scatter} \texttt{string | array<string>} \text{ False} \text{ scatterMethod ScatterMethod} \text{ False} \text{ Required if scatter is an array of more than one element.}

### 4.2.1 WorkflowStepInput

The input of a workflow step connects an upstream parameter (from the workflow inputs, or the outputs of other workflows steps) with the input parameters of the underlying step.

**Input object**

A WorkflowStepInput object must contain an \texttt{id} field in the form \texttt{#fieldname} or \texttt{#stepname.fieldname}. When the \texttt{id} field contains a period, the field name consists of the characters following the final period. This defines a field of the workflow step input object with the value of the \texttt{source} parameter(s).

**Merging**

To merge multiple inbound data links, MultipleInputFeatureRequirement must be specified in the workflow or workflow step requirements.

If the sink parameter is an array, or named in a workflow scatter operation, there may be multiple inbound data links listed in the \texttt{source} field. The values from the input links are merged depending on the method specified in the \texttt{linkMerge} field. If not specified, the default method is “merge_nested”.

- \textbf{merge\textunderscore nested}
  
  The input must be an array consisting of exactly one entry for each input link. If “merge\textunderscore nested” is specified with a single link, the value from the link must be wrapped in a single-item list.

- \textbf{merge\textunderscore flattened}
  
  1. The source and sink parameters must be compatible types, or the source type must be compatible with single element from the “items” type of the destination array parameter.
  
  2. Source parameters which are arrays are concatenated. Source parameters which are single element types are appended as single elements.

**Fields**

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>string</td>
<td>\texttt{true}</td>
<td>A unique identifier for this workflow input parameter.</td>
</tr>
<tr>
<td>source</td>
<td>string</td>
<td>array&lt;string&gt;</td>
<td>\texttt{false}</td>
</tr>
<tr>
<td>linkMerge</td>
<td>LinkMergeMethod</td>
<td>\texttt{false}</td>
<td>The method to use to merge multiple inbound links into a single array. If not specified, the default method is &quot;merge\textunderscore nested&quot;.</td>
</tr>
<tr>
<td>default</td>
<td>Any</td>
<td>\texttt{false}</td>
<td>The default value for this parameter if there is no source field.</td>
</tr>
</tbody>
</table>
valueFrom | string | Expression | False

To use valueFrom, StepInputExpressionRequirement must be specified in the workflow or workflow step requirements.

If `valueFrom` is a constant string value, use this as the value for this input parameter.

If `valueFrom` is a parameter reference or expression, it must be evaluated to yield the actual value to be assigned to the input field.

The `self` value of in the parameter reference or expression must be the value of the parameter(s) specified in the `source` field, or null if there is no `source` field.

The value of `inputs` in the parameter reference or expression must be the input object to the workflow step after assigning the `source` values and then scattering. The order of evaluating `valueFrom` among step input parameters is undefined and the result of evaluating `valueFrom` on a parameter must not be visible to evaluation of `valueFrom` on other parameters.

4.2.1.1 Any

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

Symbols

<table>
<thead>
<tr>
<th>symbol</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any</td>
<td></td>
</tr>
</tbody>
</table>

4.2.2 WorkflowStepOutput

Associate an output parameter of the underlying process with a workflow parameter. The workflow parameter (given in the `id` field) be may be used as a `source` to connect with input parameters of other workflow steps, or with an output parameter of the process.

Fields

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>string</td>
<td>True</td>
<td>A unique identifier for this workflow output parameter. This is the identifier to use in the <code>source</code> field of <code>WorkflowStepInput</code> to connect the output value to downstream parameters.</td>
</tr>
</tbody>
</table>

4.2.3 ScatterMethod

The scatter method, as described in workflow step scatter.

Symbols

<table>
<thead>
<tr>
<th>symbol</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dotproduct</td>
<td></td>
</tr>
<tr>
<td>nested_crossproduct</td>
<td></td>
</tr>
<tr>
<td>flat_crossproduct</td>
<td></td>
</tr>
</tbody>
</table>

4.2.4 InlineJavascriptRequirement

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

Fields

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>string</td>
<td>True</td>
<td>Always 'InlineJavascriptRequirement'</td>
</tr>
<tr>
<td>expressionLib</td>
<td>array&lt;string&gt;</td>
<td>False</td>
<td>Additional code fragments that will also be inserted before executing the expression code. Allows for function definitions that may be called from CWL expressions.</td>
</tr>
</tbody>
</table>

4.2.5 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

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>string</td>
<td>True</td>
<td>Always 'SchemaDefRequirement'</td>
</tr>
</tbody>
</table>
4.2.5.1 InputRecordSchema

Fields

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Record_symbol</td>
<td>True</td>
<td>Must be record</td>
</tr>
<tr>
<td>fields</td>
<td>array&lt;InputRecordField&gt;</td>
<td>False</td>
<td>Defines the fields of the record.</td>
</tr>
<tr>
<td>label</td>
<td>string</td>
<td>False</td>
<td>A short, human-readable label of this object.</td>
</tr>
</tbody>
</table>

4.2.5.2 InputRecordField

Fields

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>True</td>
<td>The name of the field</td>
</tr>
<tr>
<td>type</td>
<td>CWLType</td>
<td>InputRecordSchema</td>
<td>InputEnumSchema</td>
</tr>
<tr>
<td>doc</td>
<td>string</td>
<td>False</td>
<td>A documentation string for this field</td>
</tr>
<tr>
<td>inputBinding</td>
<td>CommandLineBinding</td>
<td>False</td>
<td></td>
</tr>
<tr>
<td>label</td>
<td>string</td>
<td>False</td>
<td>A short, human-readable label of this process object.</td>
</tr>
</tbody>
</table>

4.2.5.2.1 InputEnumSchema

Fields

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>symbols</td>
<td>array&lt;string&gt;</td>
<td>True</td>
<td>Defines the set of valid symbols.</td>
</tr>
<tr>
<td>type</td>
<td>Enum_symbol</td>
<td>True</td>
<td>Must be enum</td>
</tr>
<tr>
<td>label</td>
<td>string</td>
<td>False</td>
<td>A short, human-readable label of this object.</td>
</tr>
<tr>
<td>inputBinding</td>
<td>CommandLineBinding</td>
<td>False</td>
<td></td>
</tr>
</tbody>
</table>

4.2.5.2.2 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.
**loadContents**

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>False</td>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>

**position**

| field     | int        | False | The sorting key. Default position is 0. |

**prefix**

| field     | string     | False | Command line prefix to add before the value. |

**separate**

| field     | 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**

| field     | string     | False | Join the array elements into a single string with the elements separated by by itemSeparator. |

**valueFrom**

| field     | 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 command 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**

| field     | 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. |

### 4.2.5.2.3 InputArraySchema

**Fields**

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>items</td>
<td>CWLType</td>
<td>True</td>
<td>Defines the type of the array elements.</td>
</tr>
<tr>
<td>type</td>
<td>Array_symbol</td>
<td>True</td>
<td>Must be array.</td>
</tr>
<tr>
<td>label</td>
<td>string</td>
<td>False</td>
<td>A short, human-readable label of this object.</td>
</tr>
</tbody>
</table>

**inputBinding**

| field     | CommandLineBinding | False | |

### 4.2.6 SoftwareRequirement

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

**Fields**

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>string</td>
<td>True</td>
<td>Always “SoftwareRequirement”</td>
</tr>
<tr>
<td>packages</td>
<td>array&lt;SoftwarePackage&gt;</td>
<td>map&lt;SoftwarePackage.package, SoftwarePackage.specs&gt;</td>
<td>map&lt;SoftwarePackage.package, SoftwarePackage&gt;</td>
</tr>
</tbody>
</table>

### 4.2.7 SoftwarePackage

**Fields**

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>package</td>
<td>string</td>
<td>True</td>
<td>The common name of the software to be configured.</td>
</tr>
<tr>
<td>version</td>
<td>array&lt;string&gt;</td>
<td>False</td>
<td>The (optional) version of the software to configured.</td>
</tr>
<tr>
<td>specs</td>
<td>array&lt;string&gt;</td>
<td>False</td>
<td>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 <a href="https://packages.debian.org/jessie/bowtie">https://packages.debian.org/jessie/bowtie</a> could be resolved with apt-get install bowtie. An IRI <a href="https://anaconda.org/bioconda/bowtie">https://anaconda.org/bioconda/bowtie</a> could be resolved with conda install -c bioconda bowtie. Tools may also provide IRIs to index entries such as RRID (<a href="http://www.identifiers.org/rrid/">http://www.identifiers.org/rrid/</a>), such as <a href="http://identifiers.org/rrid:SCR_005476">http://identifiers.org/rrid:SCR_005476</a></td>
</tr>
</tbody>
</table>


4.2.8 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

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>string</td>
<td>True</td>
<td>InitialWorkDirRequirement</td>
</tr>
</tbody>
</table>
| 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]}`.

4.2.8.1 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

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
<td>string</td>
<td>Expression</td>
<td>True</td>
</tr>
<tr>
<td>entryname</td>
<td>string</td>
<td>Expression</td>
<td>False</td>
</tr>
<tr>
<td>writable</td>
<td>boolean</td>
<td></td>
<td>False</td>
</tr>
</tbody>
</table>

4.2.9 SubworkflowFeatureRequirement

Indicates that the workflow platform must support nested workflows in the run field of WorkflowStep.

Fields

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>string</td>
<td>True</td>
<td>Always ‘SubworkflowFeatureRequirement’</td>
</tr>
</tbody>
</table>

4.2.10 ScatterFeatureRequirement

Indicates that the workflow platform must support the scatter and scatterMethod fields of WorkflowStep.

Fields

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>string</td>
<td>True</td>
<td>Always ‘ScatterFeatureRequirement’</td>
</tr>
</tbody>
</table>

4.2.11 MultipleInputFeatureRequirement

Indicates that the workflow platform must support multiple inbound data links listed in the source field of WorkflowStepInput.

Fields

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>string</td>
<td>True</td>
<td>Always ‘MultipleInputFeatureRequirement’</td>
</tr>
</tbody>
</table>

4.2.12 StepInputExpressionRequirement
Indicate that the workflow platform must support the `valueFrom` field of WorkflowStepInput.

### Fields

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>string</td>
<td>True</td>
<td>Always 'StepInputExpressionRequirement'</td>
</tr>
</tbody>
</table>

### 4.2.13 ExpressionTool

Execute an expression as a Workflow step.

#### Fields

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputs</td>
<td>array&lt;InputParameter&gt;</td>
<td>map&lt;InputParameter.id, InputParameter.type&gt;</td>
<td>True</td>
</tr>
<tr>
<td>outputs</td>
<td>array&lt;ExpressionToolOutputParameter&gt;</td>
<td>map&lt;ExpressionToolOutputParameter.id, ExpressionToolOutputParameter.type&gt;</td>
<td>True</td>
</tr>
<tr>
<td>class</td>
<td>string</td>
<td>True</td>
<td>The expression to execute. The expression must return a JSON object which matches the output parameters of the ExpressionTool.</td>
</tr>
<tr>
<td>expression</td>
<td>string</td>
<td>Expression</td>
<td>True</td>
</tr>
<tr>
<td>id</td>
<td>string</td>
<td>False</td>
<td>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.</td>
</tr>
<tr>
<td>requirements</td>
<td>array&lt;InlineJavascriptRequirement</td>
<td>DockerRequirement (CommandLineTool.html#DockerRequirement)</td>
<td>False</td>
</tr>
<tr>
<td>hints</td>
<td>array&lt;Any&gt;</td>
<td>False</td>
<td>A short, human-readable label of this process object.</td>
</tr>
<tr>
<td>label</td>
<td>string</td>
<td>False</td>
<td>A long, human-readable description of this process object.</td>
</tr>
<tr>
<td>cwlVersion</td>
<td>CWLVersion</td>
<td>False</td>
<td></td>
</tr>
</tbody>
</table>

### 4.2.13.1 InputParameter

#### Fields

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>string</td>
<td>True</td>
<td>The unique identifier for this parameter object.</td>
</tr>
<tr>
<td>label</td>
<td>string</td>
<td>False</td>
<td>A short, human-readable label of this object.</td>
</tr>
</tbody>
</table>
secondaryFiles | string | Expression | array<string | Expression> | False
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
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, preferably 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
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 | InputRecordSchema | InputEnumSchema | InputArraySchema | string | array<CWLType> | InputRecordSchema | InputEnumSchema | InputArraySchema | string | False
Specify valid types of data that may be assigned to this parameter.

4.2.13.2 ExpressionToolOutputParameter

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>required</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>string</td>
<td>True</td>
<td>The unique identifier for this parameter object.</td>
</tr>
<tr>
<td>label</td>
<td>string</td>
<td>False</td>
<td>A short, human-readable label of this object.</td>
</tr>
</tbody>
</table>
| secondaryFiles | string | Expression | array<string | Expression> | False
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, preferably 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  

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>  

A documentation string for this type, or an array of strings which should be concatenated.

outputBinding: CommandOutputBinding  

Describes how to handle the outputs of a process.

type: CWLType | OutputRecordSchema | OutputEnumSchema | OutputArraySchema | string | array<CWLType | OutputRecordSchema | OutputEnumSchema | OutputArraySchema | string>  

Specify valid types of data that may be assigned to this parameter.

---

### 4.2.13.3 CWLVersion

Version symbols for published CWL document versions.

#### Symbols

<table>
<thead>
<tr>
<th>symbol</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>draft-2</td>
<td></td>
</tr>
<tr>
<td>draft-3.dev1</td>
<td></td>
</tr>
<tr>
<td>draft-3.dev2</td>
<td></td>
</tr>
<tr>
<td>draft-3.dev3</td>
<td></td>
</tr>
<tr>
<td>draft-3.dev4</td>
<td></td>
</tr>
<tr>
<td>draft-3.dev5</td>
<td></td>
</tr>
<tr>
<td>draft-3</td>
<td></td>
</tr>
<tr>
<td>draft-4.dev1</td>
<td></td>
</tr>
<tr>
<td>draft-4.dev2</td>
<td></td>
</tr>
<tr>
<td>draft-4.dev3</td>
<td></td>
</tr>
<tr>
<td>v1.0.dev4</td>
<td></td>
</tr>
<tr>
<td>v1.0</td>
<td></td>
</tr>
</tbody>
</table>