



**Antmicro**

**Pipeline Manager**

2025-10-22

## CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Pipeline Manager</b>	<b>2</b>
2.1	Prerequisites . . . . .	3
2.2	Building and running . . . . .	3
<b>3</b>	<b>Specification format</b>	<b>7</b>
3.1	Format description . . . . .	7
3.2	Example . . . . .	21
<b>4</b>	<b>Dataflow format</b>	<b>25</b>
4.1	Format description . . . . .	25
4.2	Example dataflow . . . . .	30
<b>5</b>	<b>Front-end features</b>	<b>37</b>
5.1	Graph manipulation . . . . .	38
5.2	Settings . . . . .	40
5.3	Notifications . . . . .	41
5.4	Full screen . . . . .	42
5.5	Editor menu . . . . .	43
5.6	Creating new node types . . . . .	44
5.7	Working with the server . . . . .	44
5.8	URL parameters for the frontend . . . . .	45
5.9	External frontend . . . . .	46
5.10	Testing the front-end features . . . . .	47
<b>6</b>	<b>Communication with an external application</b>	<b>48</b>
6.1	Communication protocol . . . . .	48
6.2	Response messages types - sent by the external application . . . . .	53
6.3	API Specification . . . . .	53
6.4	Implementing a Python-based client for Pipeline Manager . . . . .	66
<b>7</b>	<b>Sample third-party server</b>	<b>69</b>
<b>8</b>	<b>Specification builder</b>	<b>70</b>
8.1	Example usage of the SpecificationBuilder . . . . .	70
8.2	SpecificationBuilder documentation . . . . .	74
<b>9</b>	<b>Dataflow graph builder</b>	<b>87</b>
9.1	Examples of GraphBuilder usage . . . . .	87

9.2	Exception Hierarchy . . . . .	89
9.3	Specification of GraphBuilder . . . . .	90
9.4	Specification of DataflowGraph . . . . .	93
9.5	Specification of Node . . . . .	96
9.6	Specification of Interface . . . . .	99
9.7	Specification of Connection . . . . .	99
9.8	Specification of Property . . . . .	100
<b>10</b>	<b>Dataflow and Specification format changelog</b>	<b>101</b>
10.1	20250623.14 . . . . .	101
10.2	20240723.13 . . . . .	102
10.3	20230523.12 . . . . .	102
10.4	20230830.11 . . . . .	103
10.5	20230824.10 . . . . .	103
10.6	20230824.9 . . . . .	103
10.7	20230818.8 . . . . .	103
10.8	20230817.7 . . . . .	104
10.9	20230809.6 . . . . .	104
10.10	20230619.5 . . . . .	104
10.11	20230619.4 . . . . .	104
10.12	20230619.3 . . . . .	104
10.13	20230615.2 . . . . .	105
10.14	20230615.1 . . . . .	105
<b>11</b>	<b>Demonstration of the frontend</b>	<b>106</b>
11.1	Example of a more complex graph with nodes as interfaces, inheritance and more	106
11.2	Example of a graph with dynamic number of interfaces . . . . .	107
11.3	Example of the graph with include mechanisms . . . . .	107
11.4	Example of the graph with top-level interfaces exposed . . . . .	108
11.5	Example of graph with inheritance . . . . .	108
11.6	Example of Input-Output interfaces . . . . .	109
11.7	Example of interface grouping . . . . .	109
11.8	Example of graphs with connection loopbacks . . . . .	110
11.9	Example of Multiple Input-Output node . . . . .	110
11.10	Example of a graph with graph nodes . . . . .	111
11.11	Communication with Pipeline Manager via postMessage . . . . .	111
<b>Index</b>		<b>112</b>

## INTRODUCTION

Pipeline Manager is a data-based web application for creating, visualizing and managing pipelines for various applications.

This documentation contains the following chapters:

- *Pipeline Manager* - a quickstart guide for the project
- *Specification format* - description of the node specification format
- *Dataflow format* - graph description format overview
- *Front-end features* - description of front-end features and front end testing
- *Communication with an external application* - external application integration overview
- *Sample third-party server* - describes how to configure server for testing frontend features
- *Specification Builder* - describes the Python's SpecificationBuilder module for constructing the specification.
- *Graph Builder* - describes the Python's GraphBuilder class for constructing and manipulating the dataflow graph.
- *Dataflow and specification format changelog* - lists changes and versions in the format of dataflow and specification files
- *Front end demo* - Pipeline Manager's front end demonstration.

## PIPELINE MANAGER

Copyright (c) 2022-2025 [Antmicro](#)

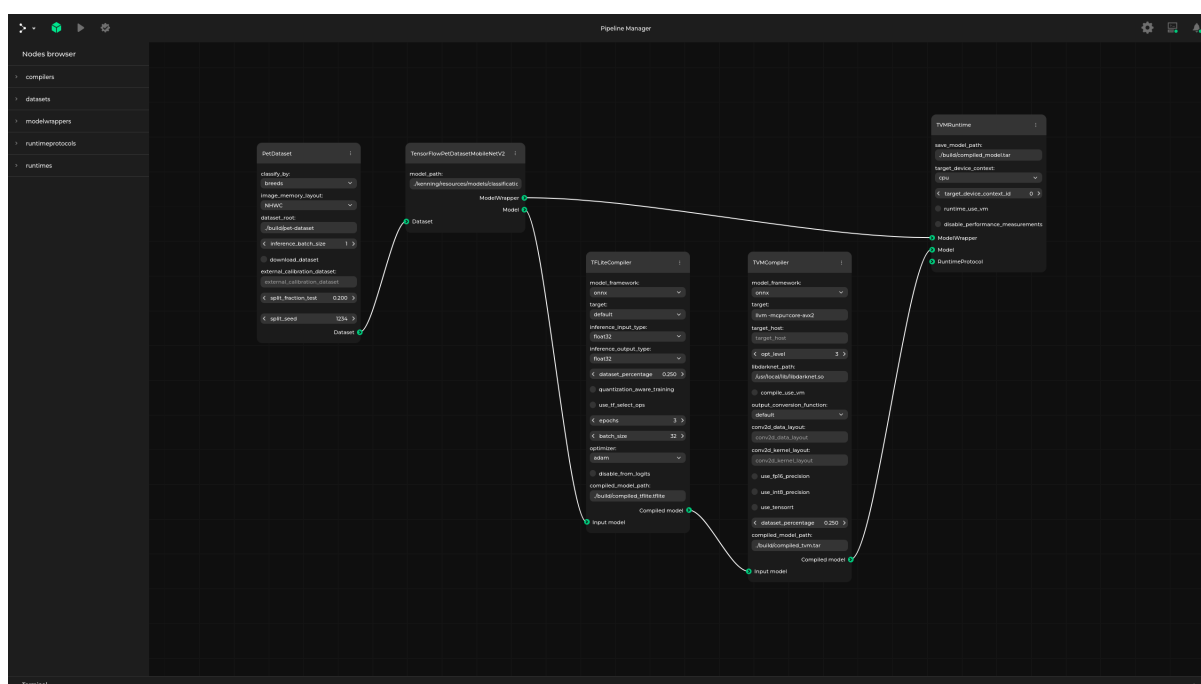
Pipeline Manager is a data-based, application-agnostic web application for creating, visualizing and managing dataflows in various applications. It does not assume any properties of the application it is working with, thanks to which fast integration with various formats is possible.

[Pipeline Manager documentation](#) | [Demo application](#)

It provides functionality for:

- visualizing and editing dataflows,
- saving and loading dataflows,
- communicating with an external application to delegate advanced validation, execution of the defined graph and conversions to and from native formats of the graphs.

Pipeline Manager aims to simplify the process of developing graph-based graphical interfaces for applications that are modular and have a graph-like nature.



## 2.1 Prerequisites

Pipeline Manager requires npm (at least 10.8.2 version is recommended, along with Node.js starting from 20.10.0), python and pip for installing dependencies, building and (optionally) running - use package manager to install those packages.

The backend of the application has a list of requirements in the `pyproject.toml` file. They can be installed using pip:

```
pip install .
```

All of npm modules needed for the frontend of the application are installed automatically during build. They can be found in the `./pipeline_manager/frontend/node_modules` directory after the application is built.

## 2.2 Building and running

Pipeline Manager can be built in two different ways as:

- static HTML application, that can be run in a browser, without any additional backend server,
- regular web application that is designated to communicate and cooperate with an external application (like [Kenning](#)).

### 2.2.1 Static HTML application

To build Pipeline Manager as a static HTML application, in the root directory run:

```
./build static-html
```

For available flags, run:

```
./build static-html -h
```

To run the built application, open `./pipeline_manager/frontend/dist/index.html` in a preferred browser. As an example, if the browser of your choice is firefox you should run:

```
firefox ./pipeline_manager/frontend/dist/index.html
```

After running Pipeline Manager you can use sample specification under `./examples/sample_specification.json` to check the visualization and editing of pipelines. Additionally, `./examples/sample_dataflow.json` can be used to see how dataflows are stored.

The specification can be loaded in the webpage using the Load specification option in the main menu. The dataflow can be loaded in the webpage using the Load graph file option in the main menu.

What is more, the specification and the dataflow can be provided as URL arguments:

- spec - should contain URL to the specification file,
- graph - should contain URL to the dataflow file,
- preview - if true, the graph is displayed in preview mode (read only, no HUD),

- include - to alter the specification from the level of URL, it is possible to provide a URL to additional includes with this field.

For example:

```
firefox "https://antmicro.github.io/kenning-pipeline-manager/_static/pipeline-
↪manager.html?spec=https%3A%2F%2Fraw.githubusercontent.com%2Fantmicro%2Fkenning-
↪pipeline-manager%2Frefs%2Fheads%2Fmain%2Fexamples%2Fsample-specification.json&
↪graph=https%3A%2F%2Fraw.githubusercontent.com%2Fantmicro%2Fkenning-pipeline-
↪manager%2Frefs%2Fheads%2Fmain%2Fexamples%2Fsample-dataflow.json"
```

Will fetch and use specification and dataflow from the GitHub repository for this project. The URLs need to be encoded.

It is possible to add a default specification JSON to the generated HTML. It just needs to be provided as the second argument of the `./build` script:

```
./build static-html <path-to-specification-json>
```

It is also possible to add a default dataflow that will be loaded on the start of the application, e.g.:

```
./build static-html <path-to-specification> <path-to-dataflow>
```

To be able to use some additional assets, like icons for nodes, run:

```
./build --assets-directory <path-to-assets-dir> static-html <path-to-
↪specification> <path-to-dataflow>
```

To change the title of the editor and page, use `--editor-title` flag, e.g.:

```
./build --editor-title 'Graph editor' static-html <path-to-specification> <path-
↪to-dataflow>
```

For details on how to write specification, check:

- [Pipeline Manager documentation](#)
- [Specification format](#)
- [Dataflow format](#)
- [Examples in examples/ directory](#) - in the directory you can find sample specification files (with `-specification.json` suffix), usually paired with supported dataflow files (with `-dataflow.json` suffix)

For example, run:

```
./build static-html ./examples/sample-specification.json ./examples/sample-
↪dataflow.json --output-directory ./pipeline-manager-demo
```

After successful build, run:

```
firefox ./pipeline-manager-demo/index.html
```

You should get a graph view similar to the one in the [documentation's demo](#).

## 2.2.2 Web application

To build Pipeline Manager to work with an external application (like Kenning), in the root directory run:

```
./build server-app
```

For available flags, check:

```
./build server-app -h
```

In this scenario, the backend server is expected to serve the Pipeline Manager content. To do that, in the root directory run:

```
./run
```

By default, the backend server runs on `http://127.0.0.1:5000`. In addition to using the sample specification you can also connect the third-party application (e.g. [Kenning](#)), edit its pipeline, validate it and run it.

## 2.2.3 Miscellaneous

### Development

To run a development server which automatically recompiles the project after detecting any changes, in `./pipeline_manager/frontend` directory run:

```
npm run serve-static
```

in case of a static mode, and run:

```
npm run serve
```

in case of a regular web application mode.

### Validation

Pipeline Manager also includes a validation tool you can use during specification and dataflow development.

To validate an existing specification, run the following in the root directory:

```
./validate <specification-path>
```

To validate an existing specification and one or more dataflows, run the following in the root directory:

```
./validate <specification-path> <dataflow-path> <dataflow-path> ...
```

Replace both `specification-path` and `dataflow-path` with the actual paths to the JSON configuration file you want to validate. When running the validation tool for the first time, make sure to include the `--instal-dependencies` flag.



## Cleanup

To remove the installed npm dependencies and the built application, run the following in the root directory:

```
./cleanup
```

### 2.2.4 Using Pipeline Manager as a Python module

To install Pipeline Manager with pip, run:

```
pip install -U git+https://github.com/antmicro/kenning-pipeline-manager.git
```

To work directly with the repository, install the module with:

```
pip install -e .
```

All Pipeline Manager scripts can then be used from the command-line interface:

```
pipeline_manager  
usage: pipeline_manager {build,run,validate,cleanup}
```

## SPECIFICATION FORMAT

Pipeline Manager requires a JSON specification that defines the working environment. It specifies all nodes, properties, available connections, and metadata for the editor. It can be either provided from a file or fetched from an *external application*.

Such a specification defining the nodes needs to be loaded through the front end in order for you to be able to load any dataflow.

### 3.1 Format description

The specification needs to be provided in a JSON file. The specification consists of:

- metadata - object of type *Metadata* that specifies editor styling and metadata
- nodes - array that specifies valid nodes, where every element is of type *Node*.
- graphs - array of dataflow-like objects defining graph nodes, of type *Graph*.
- include - array of string objects pointing to a remote url to the specifications to include.
- includeGraphs - array of objects of type *Included Graph*, that specifies graph instances to be included in the specification from a remote url.
- entryGraph - ID of the graph that should be loaded with the specification. A graph can be chosen from include or includeGraphs.
- version - string determining version of the specification. Should be set to the newest version described in *Changelogs*. Pipeline Manager uses that value to check the compatibility of the specification with the current implementation, giving warnings about inconsistency in versions.

#### Note

Graph node named New Graph Node is available by default after loading a specification. It can be used to create new, custom graphs. That is why using this name for a graph will result in an error.

#### 3.1.1 Metadata

This object specifies additional editor options and contains the following optional properties:

- interfaces - a dictionary which defines interface and connection styling for particular interface types. The key in the dictionary is the name of the interface type, and the value is of type *Interface style*.

- `allowLoopbacks` - boolean value that determines whether connections with endpoints at the same node are allowed. Default value is `false`.
- `readonly` - boolean value determining whether the editor is in read-only mode. In read-only mode, the user cannot create or remove nodes and connections. Modification of any properties is also disabled. The user is only allowed to load existing dataflows. Default value is `false`.
- `twoColumn` - boolean value determining the layout of the nodes. If set to `true`, then input and output sockets are both rendered in the top part of the node and properties are displayed below. Default value is `false`.
- `connectionStyle` - string value that determines the connection style. Can choose one out of the two options: `curved` or `orthogonal`. Default value is `curved`
- `hideHud` - boolean value determining whether UI elements should be hidden. Components affected by this flag are: popup notifications, navigation bar and terminal window. Default value is `false`
- `layers` - layers specifying groups of interfaces and nodes that can be hidden in the viewer from settings level. The entries are of type *Layer*. By default there are no layers.
- `urls` - list of URL classes present in the specification and flow. It is a dictionary, where key is the name of the URL class, and value is of type *URL class specification*. The given URL classes can be referred to later in *Nodes* with proper link suffix.
- `collapseSidebar` - boolean value determining whether nodes sidebar should be collapsed by default. Default value is `true`
- `movementStep` - Defines offset to which nodes snap in the grid. Default value is `1`.
- `backgroundSize` - Defines size of the background grid. Default value is `100`.
- `layout` - String specifying the name of autolayout algorithm used in placing nodes from dataflow Default value is `NoLayout`
- `icons` - contains definition of icon classes. Icon classes are key-value pairs, where key is the name of the icon class, and value is the URL prefix that is used to compute the actual path.
- `styles` - a dictionary that defines node styles. The key in the dictionary is the name of the style, and the value is of type *Node style*.
- `welcome` - boolean value determining whether a welcome text should appear on empty editor. This feature is enabled by default.
- `navbarItems` - list of buttons that are displayed in the navbar in server-app mode, that allow for calling custom procedures. The entries are of type *Navbar item*.
- `logLevel` - string specifying minimum level of verbosity notification has to have to be displayed. It can be one of `INFO`, `WARNING`, `ERROR`.
- `editableTypes` - Boolean specifying whether nodes can be edited. It may be changed later in the UI (given that `toggleableEditableTypes` is enabled in the specification). The default value is `false`.
- `toggleableEditableTypes` - Boolean value determining whether node editability can be toggled from the frontend. The default value is `true`.

- `newGraphNode` - Boolean value telling whether anonymous graphs should be available for the given specification or not.
- `notifyWhenChanged` - Boolean value specifying whether the frontend should send requests with information about changed values. This option is only effective when Pipeline Manager is run in server-app mode. Notifications include events from the *list of available events* ending with `_on_change`.
- `hideAnchors` - Boolean value determining whether anchors in all connections should be hidden and only visible on hover. The default value is false.
- `showIds` - Boolean value determining whether node IDs and graph IDs should be visible in the editor. The default value is false.
- `showHiddenProperties` - Boolean value determining whether properties marked with `hideOnDefault` should be visible regardless of their value. The default value is false.

An example:

```
"metadata": {
  "interfaces": {
    "Dataset": {
      "interfaceColor": "#FF00FF",
      "interfaceConnectionPattern": "dashed",
      "interfaceConnectionColor": "#FF0000"
    }
  },
  "allowLoopbacks": false,
  "readonly": true,
  "connectionStyle": "orthogonal",
  "hideHud": false,
  "layers": [
    {
      "name": "Example Layer",
      "nodeLayers": ["processing"],
      "nodeInterfaces": ["BinaryImage"]
    }
  ],
  "layout": "NoLayout"
}
```

## Layer

Layer is used to describe a set of types of nodes and interfaces. Layers can be used to hide nodes of given types. They can be also used to hide connections for given interface types.

The layers can be enabled or disabled in editor settings.

Every layer has three properties (at least name and one of `nodeLayers` or `nodeInterfaces` need to be defined):

- `name` - name of the layer displayed in the editor,
- `nodeLayers` (optional) - array of names of node types that belong to the layer,
- `nodeInterfaces` (optional) - array of names of interface types that belong to the layer.

## URL class

URL class provides links with additional information for a given node. They are represented in nodes as icons leading to a URL with additional information on nodes.

The name of the URL class is specified as key in `urls` entry in metadata.

The URL class parameters are following:

- `name` - name of the URL class, appears as hint on hover over the icon,
- `icon` - path to the icon representing the URL class,
- `url` - base URL for URL class. The suffixes for URLs are present in *Node* parameters.

## Interface style

Interface style describes how interfaces of a given type should look like, as well as its input or output connections. It consists of the following properties:

- `interfaceColor` - describes the color of the interface, should be a hexadecimal number representing RGB values.
- `interfaceConnectionPattern` - describes how the connection line should look like. The possible variants are solid, dashed and dotted.
- `interfaceConnectionColor` - describes the color of connection lines, should be a hexadecimal number representing RGB values.

## Node style

Node style defines how nodes of a given style should look like. Currently, the following elements are supported:

- `color` - color of the node unless a node sets its own color,
- `icon` - secondary icon of the node, which is displayed separately from *Node* icon,
- `pill` - pill of the node unless a node sets its own pill.

Field values are defined the same way as in *Node* specification, however, `icon` can obtain *Navbar item* `iconName` values as well.

There exist special styles:

- `__new` - managed automatically and does not persist specification saves and loads,
- `__edited` - added to the node type which was modified.

## Specificity

*Node type* can define multiple styles and styling of type-specific elements (e.g. `icon`). Moreover, *an instance* can do this as well, although it is currently limited to the color option.

The following order of specificity is used when evaluating the actual style of a given element (the first - the most important):

1. Node instance styling
2. Node type styling
3. Styles defined by the node type

## Cascading

Appearance of a given element is determined by the the most specific rule that defines the element.

### Note

Value `null` is a regular style value (i.e. undergoes the same merging procedure). It discards any lower-precedence styling of an element.

## Example

For example, given:

- styles:

```
{
  "style1": {
    "options": {
      "color": "#0000FF",
      "icon": "Cogwheel",
      "pill": {
        "text": "options"
      }
    },
    "miscellaneous": {
      "color": null,
      "pill": {
        "text": "misc"
      }
    }
  }
}
```

- a node type:

```
{
  "name": "Node",
  "style": ["options", "miscellaneous"],
  "pill": null
}
```

- a node instance:

```
{
  "id": "401aff7b-fbb7-44cb-8f77-c78c80deefff",
  "position": {
    "x": 0,
    "y": 0
  },
  "width": 300,
```

(continues on next page)

(continued from previous page)

```

"twoColumn": false,
"interfaces": [],
"properties": [],
"enabledInterfaceGroups": [],
"name": "Node",
"instanceName": "Node",
"color": "#000000"
},

```

would resolve in the following way:

Element	Style options	Style miscellaneous	Node type	Node instance	Resolved value
color	##0000FF	null	undefined	#000000	#000000
icon	Cogwheel	undefined	undefined	–	Cogwheel
pill	{"text": "options"}	{"text": "misc"}	null	–	null

### Navbar item

Describes a list of custom buttons displayed in the navbar. Every element consists of the following properties:

- name - displayed as a tooltip to the user when the button is hovered. Names have to be unique.
- stopName - name displayed in the tooltip when the corresponding procedure is running and is stoppable. If not specified simply 'Stop' is added as a prefix to the name.
- iconName - name of the icon that is used. It can be either a file in the assets directory, or an icon described in /pipeline-manager/pipeline\_manager/frontend/src/icons/index.ts.
- procedureName - name of the procedure (either a built-in *API procedure* or a *custom procedure*) to be called. It is assumed that the procedure accepts a does not need any arguments, or takes one argument which is the currently displayed dataflow (similarly to [dataflow\_run(#external-dataflow-run)]).
- allowToRunInParallelWith - a list of procedure names that can be started in parallel to the currently running job.

Example of a button that is used to run the current graph using a dedicated procedure dataflow\_run looks as follows:

```

{
  "name": "Run",
  "stopName": "Stop",
  "iconName": "Run",
  "procedureName": "dataflow_run",
  "allowToRunInParallelWith": [
    "dataflow_validate",
    "custom_lint_files"
  ]
}

```

(continues on next page)

(continued from previous page)

```
]
}
```

This will create a button called Run, with the Run icon (available built-in icon), which upon clicking will run `dataflow_run` method. During execution of the `dataflow_run` method, we can run in parallel `dataflow_validate` and `custom_lint_files` procedures.

#### Note

By default, only single action can be called from NavBar at a time.

#### Warning

`allowToRunInParallelWith` is a one-way relation. This means that if we have two actions that can be started when the other one is running, then `allowToRunInParallelWith` needs to be defined in both actions.

### 3.1.2 Node

This object specifies a single node.

- `name` - name displayed in the editor.
- `layer` - layer type used for styling and toggling the node visibility in the editor.
- `category` - context menu category displayed in the editor.
- `icon` - name of an SVG icon that is going to be displayed next to the name of the node. The value of `icon` can be:
  - string containing the path to the icon relative to the assets directory, e.g. `filter.svg` for `pipeline_manager/frontend/dist/assets/filter.svg`
  - string containing the URL to the remote image file
  - `{"key": "value"}` pair, where `key` is the name of the icon class created in icons, in *Metadata*, and `value` is a suffix of the URL.

#### Note

The assets directory can be created and filled manually or added by build script with `--assets-directory <path-to-directory-with-icons>`, e.g.:

```
./build --assets-directory examples/sample-assets static-html static-html_
↪examples/sample-specification.json
```

- `style` - node style name described in styles in *Metadata*. The value of `style` can be:
  - string - single style,
  - array - multiple styles, each subsequent style overrides the previous one.



- interfaces - array representing inputs, outputs and bidirectional ports for nodes. The entries are of type *Interface*.
- properties - array with elements of type *Property*,
- interfaceGroups - array with elements of type *Interface Groups*,
- defaultInterfaceGroups - array of objects that specifies which interface groups are enabled by default. Every object should contain a name and direction of an *Interface Groups*.
- urls - a dictionary of *URL class* and URL suffixes pairs. The key should be a URL class key from urls in metadata. The value of the entry is appended to the URL base from the URL class.
- abstract - boolean determining whether the node type is abstract or not. Abstract node types are used only for inheritance purposes, they do not appear in the final list of available nodes. They only have one mandatory field - name. The rest of the fields can be provided to introduce some common properties of classes inheriting from it.
- additionalData - can be any JSON-like object (array, dictionary, number, string, ...), it is only used for storing some additional, node-specific data, such as comments etc.
- description - description of the node in markdown format that is displayed in a sidebar node.
- isCategory - determines whether the node is both a category and a node. If set to true, then the name of the node is inferred from the last text segment in the category parameter. If any node has a node category in its category path then it has to extend from the nearest ancestor category node. Additionally, if a node extends from a category node, then it has to be in its subtree.
- color - a hexadecimal color that will be used as the node background in the editor. If not provided, the default Kenning Pipeline Manager color scheme will be used.
- width - a default width, in pixels, of a node in the editor.
- twoColumn - boolean value. If set to true, the interfaces on opposite sides will be arranged parallel to each other. Otherwise, each interface will be positioned on a separate line.
- pill - a dictionary that defines an additional node tag displayed in the top right corner of the node. It has the following entries:
  - text - content of the pill,
  - color (optional) - background color. The default color is white.

Some or all of the properties above (except for name) can be derived from existing node types using the extends list - check *Node type inheritance*.

Here is an example of a node:

```
{
  "name": "Filter2D",
  "layer": "filters",
  "category": "Filters",
  "icon": "filter.svg",
  "description": "Node that applies a *2D* filter.",
  "properties": [
    {
```

(continues on next page)

(continued from previous page)

```
        "name": "iterations",
        "type": "integer",
        "default": 1
    },
    {
        "name": "border type",
        "type": "select",
        "values": ["constant", "replicate", "wrap", "reflect"],
        "default": "constant"
    }
],
"interfaces": [
    {
        "name": "image",
        "type": "Image",
        "direction": "input"
    },
    {
        "name": "kernel",
        "type": "Image",
        "direction": "input"
    },
    {
        "name": "output",
        "type": "Image",
        "direction": "output"
    }
],
"color": "#4e096f"
}
```

## Interface

An object that specifies a single input, output or inout interface of node. Every interface object has following properties:

- name - name of the input displayed in the editor
- type - type of the input used for styling and validation purposes. Can be either a list of strings or a single string. If two interfaces have at least one matching type, they can be connected. The first type in the list is a “base” type - it is used to color the interface based on *Interface style*. If only one type between two interfaces is matching, the connection style matches the one defined for this particular type. Otherwise, if multiple types are matching, a white solid line for connection is rendered.
- direction (optional) - type of the interface in terms of direction, it can be:
  - input - interface accepts input data,
  - output - interface returns output data.
  - inout - interface can both produce outputs and receive inputs.

The default value is inout.

- `maxConnectionsCount` (optional) - specifies the maximum allowed number of connections for a given port. Value less than 0 means no limit for connections for a given interface. Value equal to 0 means default behavior - one allowed connection to inputs, many allowed connections from outputs. The default value is 0.
- `side` (optional) - specifies the side on which the interface is rendered. Value can be either left or right. Interfaces with direction set to input or inout are by default rendered on the left side of the node. Interfaces with direction set to output are by default rendered on the right side of the node.
- `dynamic` (optional) - determines whether the interface should be dynamic, which means that the number of interfaces created of this definition can be changed in the editor. For each dynamic interface definition, a new property is added to the node that allows the user to change the number of interfaces. The name of the property is determined by the name and direction of the dynamic interface. The format is `{name} {direction} count`, so it is important not to create conflicting properties. The name of the dynamic interfaces is `{name} [{i}]`, where `i` is the index of the interface. `dynamic` property can be either true, then the maximum number of interfaces is not limited, or an array in a form of `[min, max]` that specifies the range of interfaces. All created interfaces share the same `maxConnectionCount` and `type` values.
- `array` (optional) - special keyword to easily define a range of interfaces. Value has to be a list with two integer values that specify the range of interfaces. For example, for an example interface with array: `[0, 2]` two interfaces called `example[0]` and `example[1]` are created.
- `sidePosition` (optional) - specifies the row on which the interface is rendered. Values for interfaces of the same side value have to be unique. If the value is not provided then rows are automatically provided by iterating from the first upper row. This value does not work for array keyword, as it produces more than one interface.
- `externalName` (optional) - used for graph nodes only, to expose an interface of a node within the graph. It specifies the name of the interface of the graph. Both the interface of the graph node and the interface of the node within the graph must have the same id and direction fields. Note that values of `externalName` of the graph node have to be unique.

#### Note

Only interfaces of the same type can be connected together.

## Property

An object that specifies a single property. Every project object has three required base properties:

- `name` - name of the property.
- `type` - type of the property.
- `default` - specifies a default selected value Its type depends on the type chosen.

There are 13 possible values for the type property.

- `text` - property is a string. A text field is displayed to the user.

- multiline - property is a multi-line string. It allows to format the text in Markdown format.
- constant - property is a string. A non-modifiable text field is displayed to the user.
- number - property is a float. A number field is displayed to the user.
- integer - property is an int. A number field that only accepts integers is displayed to the user.
- select - property is a string with a defined range. It requires a values property.
- bool - property is a bool. A bool representing boolean value
- slider - property is a float with a specified range. It requires min and max properties.
- list - property is a list of arguments of the same type, which can be specified using dtype.
- hex - property is a string representing base-16 number which has to match the following regex: `/0x[a-fA-F0-9]`.
- button-url - property is a button which opens a new tab with a given URL.
- button-api - property is a button which triggers button\_click *External App API* request, providing property id and value.
- button-graph - property is a button which opens a subgraph with a given URL.

Additional properties:

- min - specifies the left end of a range or minimal value of the base-16 number.
- max - specifies the right end of a range or maximal value of the base-16 number.
- values - specifies a range of possible values for select.
- dtype - specifies data type of elements in a list. Supported values are string, number, integer, boolean.
- description - description of the property. In some cases, it can be displayed to the user.
- group - object of type *Group*.
- override - boolean value determining whether the property should be overridden.
- readonly - boolean value determining whether the property should be read-only.
- hideOnDefault - boolean value determining whether the property should be kept hidden if it has default or no value assigned.

## Group

Array with elements of type *Property*. It defines properties that are toggled by the property. group can only be used with property of type bool.

Example group of properties:

```
{
  "name": "Group", "type": "bool", "default": true, "group": [
    {"name": "prop-a", "type": "integer", "default": 1},
    {"name": "prop-b", "type": "text", "default": ""}
  ]
}
```

(continues on next page)

(continued from previous page)

```
]
}
```

### Node type inheritance

It is possible to inherit:

- layer
- category
- icon
- interfaces
- properties
- urls

From existing node types using the extends parameter. The parameter accepts a list of node types. The node type is computed by iteratively updating node type definition structures, going through all node types in the extends list (in the specified order), and then applying parameters from the current node type.

Below is a sample specification with used inheritance mechanism:

```
{
  "nodes": [
    {
      "name": "Type A",
      "layer": "class",
      "category": "Classes",
      "properties": [
        {"name": "prop-a", "type": "text", "default": ""}
      ],
      "interfaces": [
        {"name": "output-a", "type": "Interface", "direction": "output"}
      ]
    },
    {
      "name": "Type B",
      "extends": ["Type A"],
      "properties": [
        {"name": "prop-b", "type": "text", "default": ""}
      ],
      "interfaces": [
        {"name": "output-b", "type": "Interface", "direction": "output"}
      ]
    },
    {
      "name": "Type C",
      "layer": "class",
      "category": "Classes",
```

(continues on next page)

(continued from previous page)

```

    "properties": [
      { "name": "prop-c", "type": "text", "default": "" }
    ],
    "interfaces": [
      { "name": "input-c", "type": "Interface", "direction": "input" }
    ]
  },
  {
    "name": "Type D",
    "extends": ["Type B", "Type C"],
    "properties": [
      { "name": "prop-d", "type": "text", "default": "" }
    ],
    "interfaces": [
      { "name": "inout-d", "type": "Interface", "direction": "inout" }
    ]
  }
]
}

```

### Warning

Node types can not be repeated (explicitly in list or implicitly through inheritance) in the extends list.

Moreover, it is possible to override inherited properties and interfaces. This mechanism requires using the same name and "override" attribute set to true. Only attributes used in the child node are overridden, others are inherited without change. For instance:

```

{
  "name": "Type D",
  "extends": ["Type B", "Type C"],
  "properties": [
    { "name": "prop-a", "type": "number", "default": 1.4, "override": ↪
↪ true },
    { "name": "prop-c", "type": "integer", "default": 5, "override": ↪
↪ true },
    { "name": "prop-d", "type": "text", "default": "" }
  ],
  "interfaces": [
    { "name": "inout-d", "type": "Interface", "direction": "inout" },
    { "name": "output-a", "type": "Interface", "direction": "inout" },
    ↪ "override": true }
  ]
}

```

### 3.1.3 Interface Groups

Object similar to a single interface but reserves a range of interfaces. `name`, `type`, `direction`, `maxConnectionsCount` and `side` are the same as in a regular *Interface*. The only difference is that a range of interfaces has to be defined which describes constraints of an interface. For example two interface groups can be defined that consist of common interfaces and thus cannot coexist.

```
"interfaceGroups": [  
  {  
    "name": "1",  
    "type": "test",  
    "direction": "input",  
    "interfaces": [  
      {"name": "1[1]", "direction": "output"},  
      {"name": "1", "array": [3, 15], "direction": "output"},  
      {"name": "1", "array": [35, 48], "direction": "output"}  
    ]  
  }  
]
```

The interface group called 1 consists of three ranges of interfaces: 1[1], interfaces 1[3], 1[4], ..., 1[14] and 1[35], 1[36], ..., 1[47].

### 3.1.4 Included Graphs

Object that specifies a graph instance to be included in the specification. The included graphs are available in the node palette and can be used in the editor.

#### Note

Included graphs cannot consist of more than one graph.

Included graphs can be specified using the `includeGraphs` keyword.

They have the following properties:

- `name` - name of the included graph, if not passed, the name of the graph is inferred from the graph.
- `category` - category in which the included graph is placed in the node palette, uses `/` as a delimiter. By default, the category is set to `default`.
- `url` - URL, where the graph file is located. The file has to be in a valid *dataflow* format.

Graphs included in the node palette can also be described directly in the specification.

To do that, use the `graphs` keyword and describe the graph using the *dataflow* format.

Keep in mind that the node used to represent the graph has to be declared in the specification as well.

The node needs to have `subgraphId` that matched the `id` of the graph.

It is possible to attach one graph to multiple nodes.

```
{
  "name": "Test subgraph node #2",
  "category": "Second Category",
  "color": "#6f091a",
  "subgraphId": "129c6246-d874-48cd-a33c-2927961d42e8"
},
{
  "name": "Test subgraph node #3",
  "category": "Second Category",
  "color": "#4e096f",
  "subgraphId": "129c6246-d874-48cd-a33c-2927961d42e8"
}
```

## 3.2 Example

Below, you can see a sample specification containing a hypothetical definition of nodes for image processing purposes:

```
{
  "nodes": [
    {
      "name": "LoadVideo",
      "layer": "filesystem",
      "category": "Filesystem",
      "properties": [
        {"name": "filename", "type": "text", "default": ""}
      ],
      "interfaces": [{"name": "frames", "type": "Image", "direction":
↪ "output"}]
    },
    {
      "name": "SaveVideo",
      "layer": "filesystem",
      "category": "Filesystem",
      "properties": [
        {"name": "filename", "type": "text", "default": ""}
      ],
      "interfaces": [
        {"name": "color", "type": "Image", "direction": "input"},
        {"name": "binary", "type": "BinaryImage", "direction": "input"}
      ]
    },
    {
      "name": "GaussianKernel",
      "layer": "kernel",
      "category": "Generators",
      "properties": [
        {"name": "size", "type": "integer", "default": 5},
        {"name": "sigma", "type": "number", "default": 1.0}
      ]
    }
  ]
}
```

(continues on next page)



(continued from previous page)

```

    ],
    "interfaces": [{"name": "kernel", "type": "Image", "direction":
↪ "output"}]
  },
  {
    "name": "StructuringElement",
    "layer": "kernel",
    "category": "Generators",
    "properties": [
      {"name": "size", "type": "integer", "default": 5},
      {
        "name": "shape",
        "type": "select",
        "values": ["Rectangle", "Cross", "Ellipse"],
        "default": "Cross"
      }
    ],
    "interfaces": [{"name": "kernel", "type": "BinaryImage", "direction":
↪ "output"}]
  },
  {
    "name": "Filter2D",
    "layer": "processing",
    "category": "Processing",
    "properties": [
      {"name": "iterations", "type": "integer", "default": 1},
      {
        "name": "border type",
        "type": "select",
        "values": ["constant", "replicate", "wrap", "reflect"],
        "default": "constant"
      }
    ],
    "interfaces": [
      {"name": "image", "type": "Image", "direction": "input"},
      {"name": "kernel", "type": "Image", "direction": "input"},
      {"name": "output", "type": "Image", "direction": "output"}
    ]
  },
  {
    "name": "Threshold",
    "layer": "processing",
    "category": "Processing",
    "properties": [
      {"name": "threshold_value", "type": "integer", "default": 1},
      {
        "name": "threshold_type",
        "type": "select",
        "values": ["Binary", "Truncate", "Otsu"],

```

(continues on next page)

(continued from previous page)

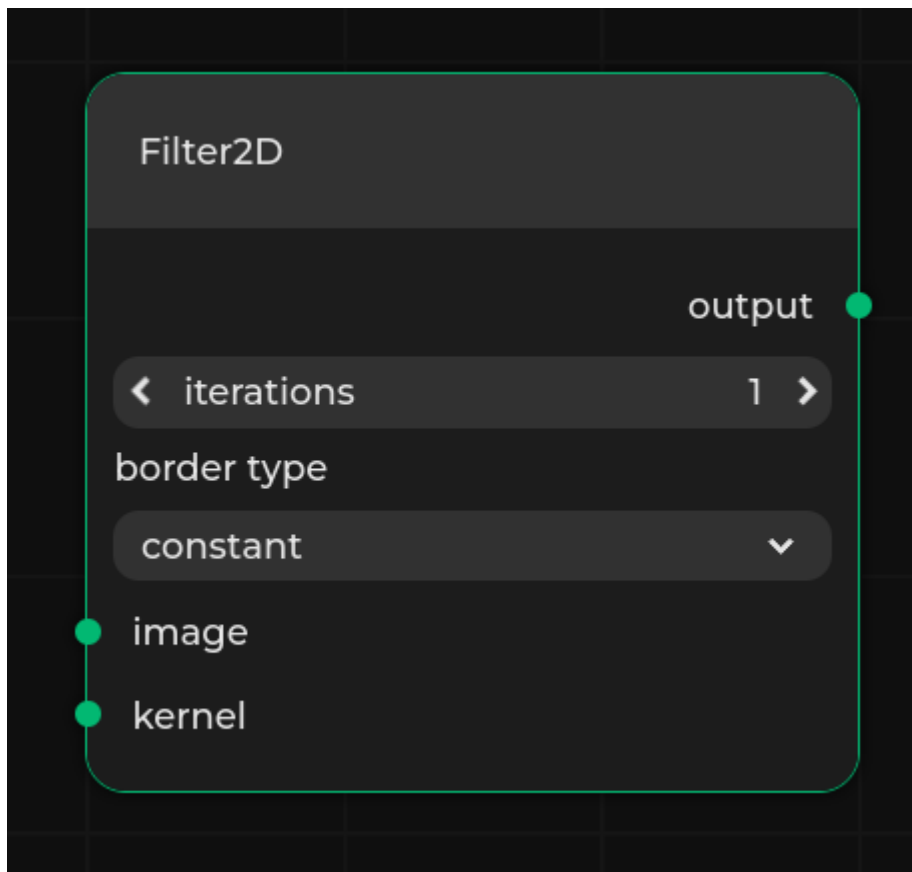
```

        "default": "constant"
    },
    ],
    "interfaces": [
        {"name": "image", "type": "Image", "direction": "input"},
        {"name": "output", "type": "BinaryImage", "direction": "output"}
    ]
},
{
    "name": "Morphological operation",
    "layer": "processing",
    "category": "Processing",
    "properties": [
        {
            "name": "Enable operations", "type": "bool", "default": true,
↪ "group": [
                {
                    "name": "iterations", "type": "integer", "default": 1},
                    {
                        "name": "border type",
                        "type": "select",
                        "values": ["constant", "replicate", "wrap", "reflect
↪"],
                        "default": "constant"
                    },
                    {
                        "name": "operation type",
                        "type": "select",
                        "values": ["dilation", "erosion", "closing", "opening
↪"],
                        "default": "dilation"
                    }
                ]
            }
        ],
        "interfaces": [
            {"name": "image", "type": "BinaryImage", "direction": "input"},
            {"name": "kernel", "type": "BinaryImage", "direction": "input"},
            {"name": "output", "type": "BinaryImage", "direction": "output"}
        ]
    },
    "metadata": {
        "interfaces": {
            "Image": "#00FF00",
            "BinaryImage": "#FF0000"
        }
    },
    "version": "20230830.10"
}

```

Thanks to the flexibility of the specification format, you can use any combination of properties and interfaces to create a custom node. It is also readable and divided into distinct parts, so you can implement a process of automated specification generation into an external application. See the *External App Communication* section to find out more.

A sample node created from the specification above:



## DATAFLOW FORMAT

A rendered graph with all its nodes, their properties, values, and connections is called a dataflow. Its state can be serialized and saved as a JSON file.

### 4.1 Format description

The root of the dataflow format consists of four main attributes.

- `metadata` - structure of type *Metadata*. It is used to override metadata settings from the *Specification format*. In general, values from dataflow's metadata override settings from specification. For simple types, such as strings or integers, values are changed. For arrays and dictionaries, the values are updated (values of existing keys are replaced with new ones, and the new values in arrays are appended to the existing entries).
- `entryGraph` - string that specifies the ID of the graph from graphs list that should be rendered when the dataflow is loaded. If not specified, the first graph in the list is rendered.
- `graphs` - array of objects of type *Graph* that describe available graphs in the dataflow. Each graph is a separate entity that can be rendered in the editor. If `entryGraph` is not specified, the first graph in the list is rendered when the dataflow is loaded.
- `version` - string that identifies the version of the specification and dataflow. It is used to check compatibility between provided dataflow and the current version of the implementation.

#### 4.1.1 Graph

Graphs are a way to encapsulate a part of the dataflow into a separate entity. They can be used to simplify the dataflow structure, group nodes, or create reusable components. On the frontend level, graphs are rendered as distinct nodes, which can be interacted with in the same way as standard nodes, but can be entered and expanded to show the content of the subgraph. On top of that, interfaces of such graph nodes may be exposed and accessed outside the graph, allowing to create more complex, multi-layer graph structures.

Each graph can be described with the following properties:

- `id` - unique value that identifies the graph.
- `name` - human-readable name of the graph.
- `additionalData` - any JSON-like data that provides additional information regarding the graph.
- `nodes` - array that specifies all nodes in the dataflow. Every element is of type *Node*.

- connections - array that specifies all connections in the dataflow. Every element is of type *Connection*.
- panning - object of type *Panning* that defines the position of the top-left corner in the rendered editor.
- scaling - floating-point number that defines the zoom level in the editor.

## Node

An object that describes a single node in the editor. Each node has:

- name - node name, as defined in the specification.
- id - unique value assigned to the node.
- instanceName - optional field defining a node's title rendered to the user. If set, instanceName (name) will be displayed, otherwise, just the name will be rendered.
- properties - list describing the node's parameterized values. Every element is of type *Property*.
- interfaces - list describing the node's interfaces. Every element is of type *Interface*.
- width - the node's width in the editor.
- twoColumn - boolean value. If set to true, the interfaces on opposite sides will be arranged parallel to each other. Otherwise, each interface will be positioned on a separate line. This overrides the value set in the specification. If not provided, the value set in the specification will be used instead.
- subgraph - optional field defining the id of the subgraph that this node represents. It refers to one of the *Graphs* entries from graphs with a matching id.
- enabledInterfaceGroups - optional array describing enabled interface groups. Every element is of type *Enabled Interface Groups*.
- color - optional field defining a hexadecimal color that will be used as the node background in the editor. This overrides the node color set in the specification. If not provided, the color set in the specification will be used instead.

## Property

Each property is described by an object with three attributes:

- id - unique value assigned to the property
- name - name of the property
- value - actual value of the property.

Node having two parameters: example\_text of value example\_value and example\_number of value 123 would have the following options value:

```
[
  {
    "id": 1,
    "name": "example_text",
    "value": "example_value"
```

(continues on next page)

(continued from previous page)

```
    },  
    {  
      "id": 2,  
      "name": "example_number",  
      "value": 123  
    }  
  ]  
}
```

## Interface

Each input, output, and inout is described by an object with the following attributes:

- `id` - unique value assigned to the property. It is used to describe connections in the dataflow.
- `name` - name of the interface
- `direction` - value determining the type of the interfaces. Can be either input, output, or inout.
- `side` - tells on which side of the node the interface should be placed.
- `sidePosition` - specifies a row on which the interface is rendered. Values for interfaces of the same side value have to be unique.
- `externalName` - name of the interface displayed in the editor. For every interface in each node, it is possible to make it a graph interface. A graph interface has its own unique external name, which is visible when the graph is nested within another graph. It is used to create multi-layer graphs, with subgraphs that can be connected to others nodes/graphs. Both the interface of the graph node and the interface of the node within the graph have the same `id` and `direction` fields. Note that values of `externalName` within the graph have to be unique.

Example of graph interfaces:

```
{  
  "graphs": [  
    {  
      "id": "9c4d5349-9d3b-401f-86bb-021b7b3e5b81",  
      "nodes": [  
        {  
          "id": "0bfba841-a1e8-429c-aa8a-d98338339960",  
          "position": {  
            "x": 0,  
            "y": 0  
          },  
          "width": 200,  
          "twoColumn": false,  
          "interfaces": [  
            {  
              "name": "Exposed Name",  
              "id": "29cd9a27-2b49-4ae7-a164-a574c07fc684",  
              "direction": "input",  

```

(continues on next page)

(continued from previous page)

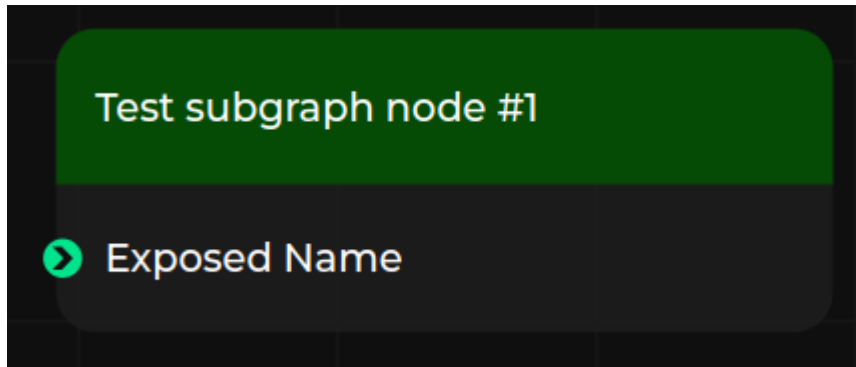
```

        "side": "left",
        "sidePosition": 0
    }
    ],
    "subgraph": "569edd54-0f42-4c24-a809-1509febbe23a",
    "name": "Test subgraph node #1"
}
],
"connections": []
},
{
    "id": "569edd54-0f42-4c24-a809-1509febbe23a",
    "nodes": [
        {
            "id": "56910b7a-9fe9-4db1-9d19-71037711d718",
            "position": {
                "x": 871,
                "y": 316
            },
            "width": 200,
            "twoColumn": false,
            "interfaces": [
                {
                    "name": "Within Input",
                    "externalName": "Exposed Name",
                    "id": "29cd9a27-2b49-4ae7-a164-a574c07fc684",
                    "direction": "input",
                    "side": "left",
                    "sidePosition": 0
                }
            ],
            "name": "Exposed interface node"
        }
    ],
    "connections": []
}
],
"entryGraph": "9c4d5349-9d3b-401f-86bb-021b7b3e5b81",
"version": "20250623.14"
}

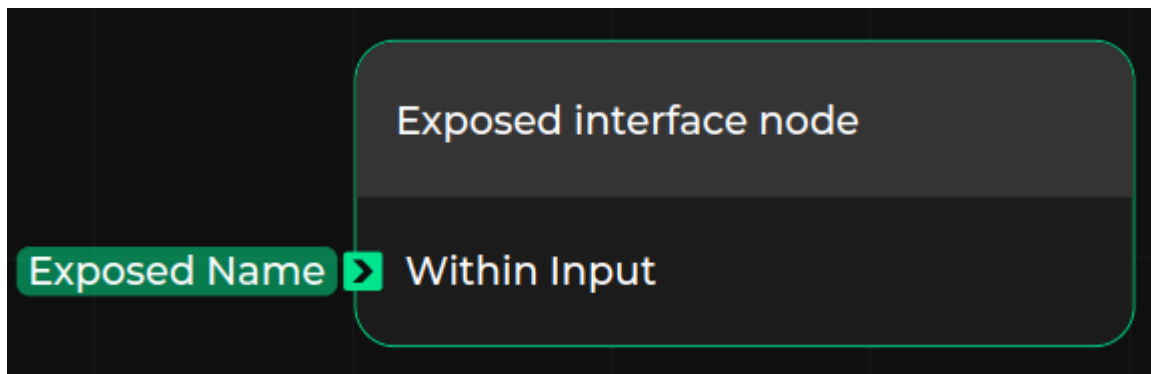
```

The example consists of two graphs. The first graph of id 9c4d5349-9d3b-401f-86bb-021b7b3e5b81 consists of a single graph node with an interface of id 29cd9a27-2b49-4ae7-a164-a574c07fc684. The node represents a graph of id 569edd54-0f42-4c24-a809-1509febbe23a, which is a graph with a single node with an interface of id 29cd9a27-2b49-4ae7-a164-a574c07fc684. The interface of the graph node is exposed as Exposed name thanks to which it can be accessed from the outside of the graph. Because the interface is exposed, both of those interfaces have to have the same id and direction value.

Graph node:



Node within graph:



### Connection

Object that describes a singular connection. It has three attributes:

- id - unique value assigned to the connection
- from - the connection's output interface id
- to - the connection's input interface id.
- anchors - optional list of *Anchor* objects.

### Anchor

This object describes a single anchor that is used to render connections divided into subparts. Every part of the connection is rendered between two adjacent anchors. Two attributes are used:

- x - x coordinate of the anchor
- y - y coordinate of the anchor.

### Panning

This object describes the position of the top-left corner of the rendered editor in the browser. Defines the camera position in the editor space. Two attributes are used:

- x - x coordinate of the corner
- y - y coordinate of the corner.



## Enabled Interface Groups

Each enabled interface group is described by the following properties:

- name - name of the interface group
- direction - value determining the type of the interfaces. Can be either input, output or inout.

### Warning

Make sure that enabled interface groups use disjoint interfaces.

## 4.2 Example dataflow

The example dataflow for a specification defined in *Specification format* is defined as below:

```
{
  "graphs": [
    {
      "id": "2035108300",
      "nodes": [
        {
          "name": "Filter2D",
          "id": "node_168064109167511",
          "position": {
            "x": 544,
            "y": 77
          },
          "width": 200,
          "twoColumn": false,
          "interfaces": [
            {
              "name": "image",
              "id": "ni_168064109167612",
              "direction": "input"
            },
            {
              "name": "kernel",
              "id": "ni_168064109167613",
              "direction": "input"
            },
            {
              "name": "output",
              "id": "ni_168064109167714",
              "direction": "output"
            }
          ],
          "properties": [
            {
              "name": "iterations",
```

(continues on next page)

(continued from previous page)

```

        "id": "8434027854",
        "value": 1
    },
    {
        "name": "border type",
        "id": "7165552813",
        "value": "constant"
    }
],
"instanceName": "Filter"
},
{
    "name": "LoadVideo",
    "id": "node_168064220761015",
    "position": {
        "x": -60,
        "y": -36
    },
    "width": 200,
    "twoColumn": false,
    "interfaces": [
        {
            "name": "frames",
            "id": "ni_168064220761016",
            "direction": "output"
        }
    ],
    "properties": [
        {
            "name": "filename",
            "id": "8887517324",
            "value": "input.mp4"
        }
    ]
},
{
    "name": "GaussianKernel",
    "id": "node_168064222522321",
    "position": {
        "x": -65,
        "y": 295
    },
    "width": 200,
    "twoColumn": false,
    "interfaces": [
        {
            "name": "kernel",
            "id": "ni_168064222522422",
            "direction": "output"
        }
    ]
}

```

(continues on next page)

(continued from previous page)

```
    }
  ],
  "properties": [
    {
      "name": "size",
      "id": "1247863780",
      "value": 5
    },
    {
      "name": "sigma",
      "id": "0187870808",
      "value": 1
    }
  ]
},
{
  "name": "Threshold",
  "id": "node_168064225320530",
  "position": {
    "x": 999,
    "y": 100
  },
  "width": 200,
  "twoColumn": false,
  "interfaces": [
    {
      "name": "image",
      "id": "ni_168064225320531",
      "direction": "input"
    },
    {
      "name": "output",
      "id": "ni_168064225320532",
      "direction": "output"
    }
  ],
  "properties": [
    {
      "name": "threshold_value",
      "id": "8770324282",
      "value": 1
    },
    {
      "name": "threshold_type",
      "id": "8305532648",
      "value": "Otsu"
    }
  ]
},
```

(continues on next page)

(continued from previous page)

```
{
  "name": "StructuringElement",
  "id": "node_168064227787336",
  "position": {
    "x": 1010,
    "y": 409
  },
  "width": 200,
  "twoColumn": false,
  "interfaces": [
    {
      "name": "kernel",
      "id": "ni_168064227787437",
      "direction": "output"
    }
  ],
  "properties": [
    {
      "name": "size",
      "id": "1587558664",
      "value": 5
    },
    {
      "name": "shape",
      "id": "1375086555",
      "value": "Cross"
    }
  ]
},
{
  "name": "Morphological operation",
  "id": "node_168064228786538",
  "position": {
    "x": 1422,
    "y": 54
  },
  "width": 200,
  "twoColumn": false,
  "interfaces": [
    {
      "name": "image",
      "id": "ni_168064228786539",
      "direction": "input"
    },
    {
      "name": "kernel",
      "id": "ni_168064228786540",
      "direction": "input"
    }
  ],
}
```

(continues on next page)

(continued from previous page)

```

        {
            "name": "output",
            "id": "ni_168064228786641",
            "direction": "output"
        }
    ],
    "properties": [
        {
            "name": "iterations",
            "id": "0605526715",
            "value": 1
        },
        {
            "name": "border type",
            "id": "2810748353",
            "value": "constant"
        },
        {
            "name": "operation type",
            "id": "8413506138",
            "value": "dilation"
        }
    ]
},
{
    "name": "SaveVideo",
    "id": "node_168064231007448",
    "position": {
        "x": 1773,
        "y": 76
    },
    "width": 200,
    "twoColumn": false,
    "interfaces": [
        {
            "name": "color",
            "id": "ni_168064231007449",
            "direction": "input"
        },
        {
            "name": "binary",
            "id": "ni_168064231007450",
            "direction": "input"
        }
    ],
    "properties": [
        {
            "name": "filename",
            "id": "3087244218",

```

(continues on next page)

(continued from previous page)

```

        "value": "output.mp4"
      }
    ]
  },
  "connections": [
    {
      "id": "168064222082820",
      "from": "ni_168064220761016",
      "to": "ni_168064109167612"
    },
    {
      "id": "168064222926625",
      "from": "ni_168064222522422",
      "to": "ni_168064109167613"
    },
    {
      "id": "168064225938335",
      "from": "ni_168064109167714",
      "to": "ni_168064225320531"
    },
    {
      "id": "168064230015344",
      "from": "ni_168064225320532",
      "to": "ni_168064228786539"
    },
    {
      "id": "168064230253147",
      "from": "ni_168064227787437",
      "to": "ni_168064228786540"
    },
    {
      "id": "168064231874053",
      "from": "ni_168064228786641",
      "to": "ni_168064231007450"
    }
  ]
  "panning": {
    "x": 0,
    "y": 0
  },
  "scaling": 1
},
]
}

```

The highlighted bits of code represent all code fragments relevant to the Filter2D node. In the nodes list, there is a full specification of the state of the Filter2D node:

- Its unique id

- Its name
- Its parameters (stored in parameters), e.g. `border_type` equal to `constant`
- Its interfaces, with unique id representing each input and output
- Rendering data, such as position or width.

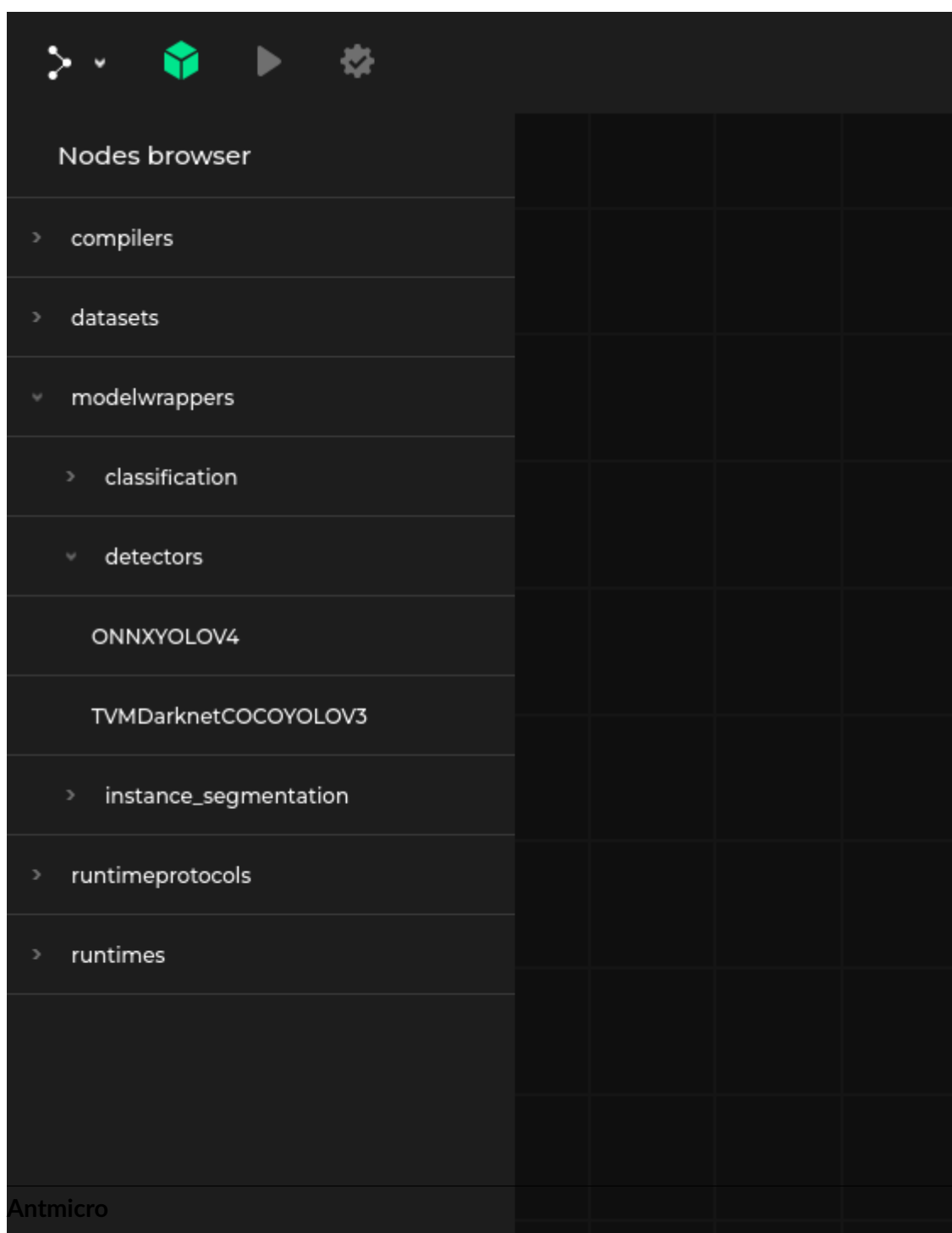
Later, in connections, you can see triples representing to which interfaces the interfaces of `Filter2D` are connected.





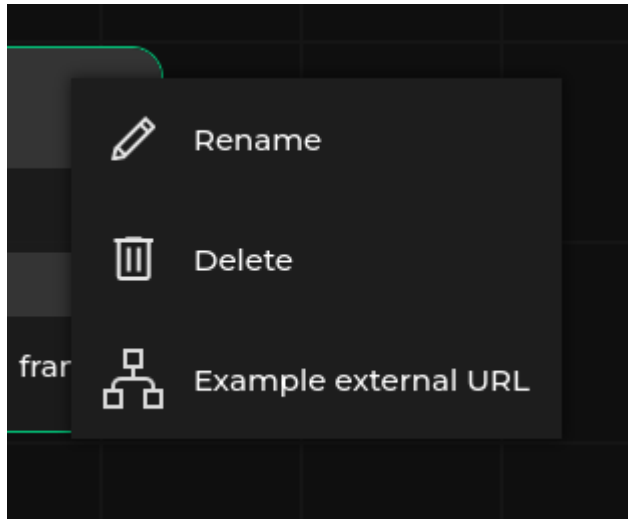
## FRONT-END FEATURES

### 5.1 Graph manipulation



On the left side there is a node palette (which can be turned off or on using the green icon in the upper left corner). To add a new node to the graph, click the node in the palette and drag it to the editor. You can also select existing nodes and copy and paste them using `ctrl-c` and `ctrl-v` keyboard shortcuts.

After loading a specification, the palette will contain all the nodes defined in it and will be divided into hierarchical categories. Additionally, a graph node named `New Graph Node` is always available under a `Graphs` category and can be used to design custom graphs.



In the upper right corner of each node there is a context menu toggle. It contains the following options:

- `Rename` - Changes `instanceName` of the node. More details regarding node naming can be found in the *'instanceName' field of dataflow format*
- `Delete` - Deletes the node from the dataflow. Another way of deleting nodes is to select and press the `Delete` key.
- Additional, user defined URLs. More details in the *URL specification*

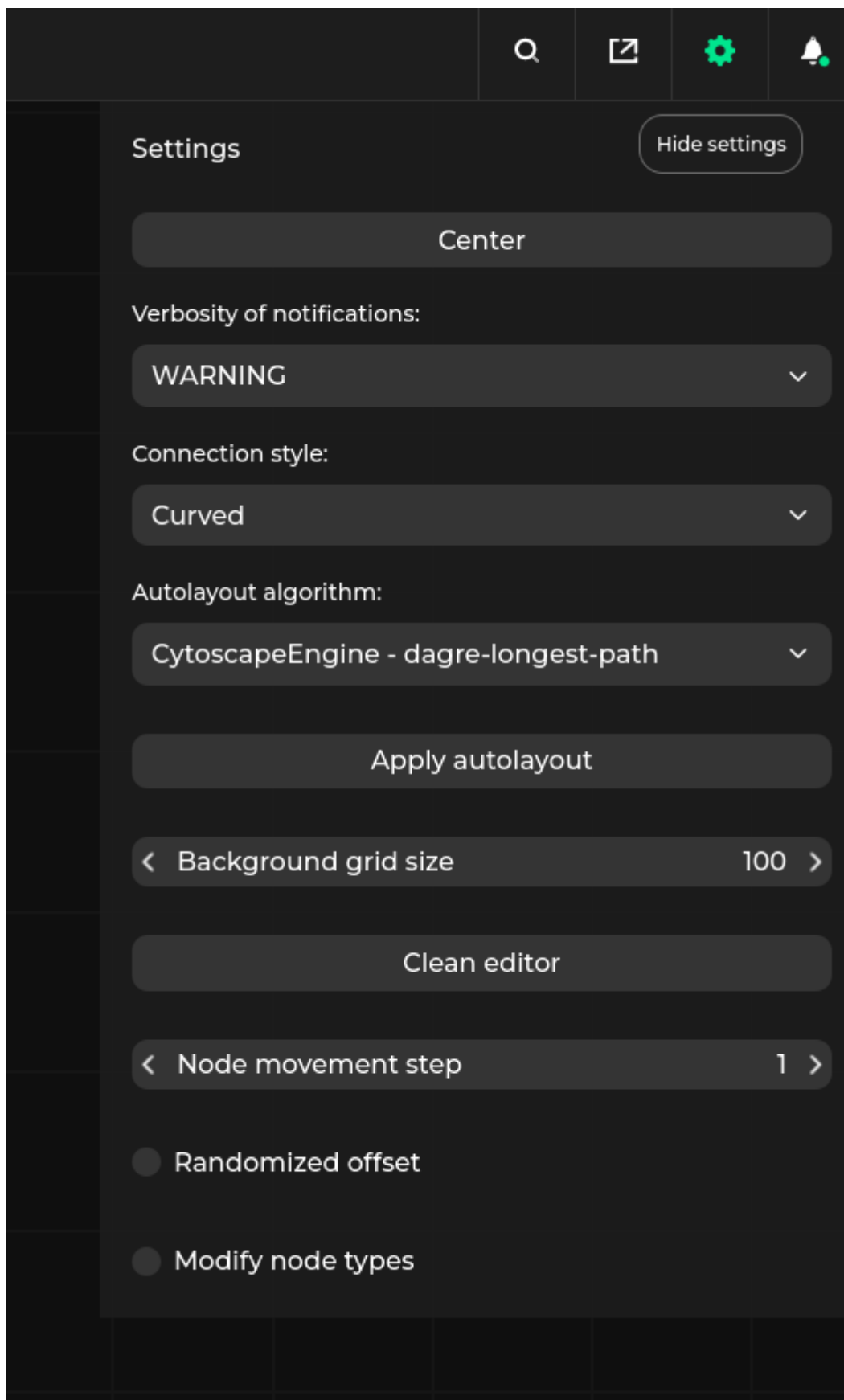
Scroll lets you to zoom in and out. Left-clicking and dragging the editor background allows you to move around the editor area.

Left-clicking on a node allows you to select and drag the node within the editor area. Pressing the `ctrl` key while moving a node allows you to align the node with another one along some axis.

To create a connection, left-click a node's connector and connect it to a connector (of a matching type, see *Specification format*) on another node. Double left-click on an existing connection removes it.

Clicking on a connection while pressing the `ctrl` key creates an anchor, which breaks up the connection into two parts, allowing to manually configure the layout. Right-clicking on an anchor removes it. Note that anchors can only be created when `Connection` style is set to `orthogonal` and the editor is not in read-only mode.

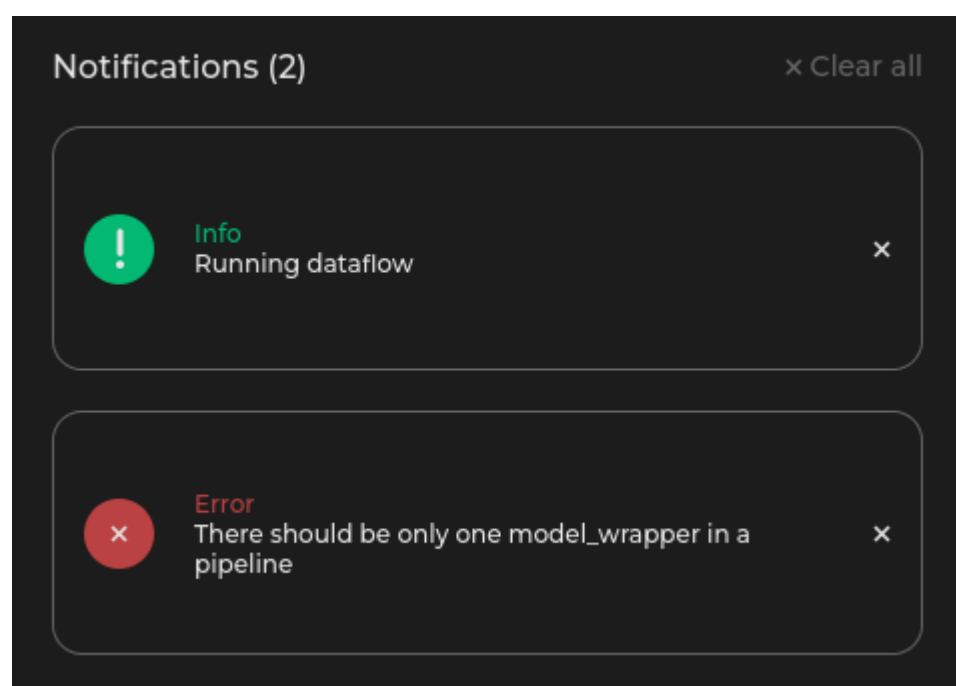
## 5.2 Settings



On the upper right corner of Pipeline Manager window there is a gear icon toggling settings tab. In the tab the following options can be found:

- Connection style - Switch the style of the connection between orthogonal and curved
- AutoLayout algorithm - Choose the algorithm for automatic node placement. Autolayout algorithm is triggered in the following situations:
  - When loading the dataflow autolayout is used to place nodes without the position parameter set. Dataflow load can be triggered either via external application or choosing the Load graph file option. In this case autolayout is applied to both the main graph and the nodes in the subgraphs.
  - By pressing the Apply autolayout button in the settings tab. In this case autolayout is applied to the entire graph.
- Background grid size - Sets the size of single grid cell visible in the background
- Node movement step - Sets the minimum step size which can be taken along each axis when moving node.
- Center - Pressing this buttons moves the viewport to the center of graph and sets the zoom level so that whole dataflow is visible
- Hide layers - *Metadata* allows to specify layers for a certain set of interface types and connections. Toggling this checkbox allows to hide connections belonging to said layer
- Modify node types - Toggling this checkbox allows editing existing nodes and adding new node types to the specification
- Hide anchors - Hides anchors in all connections. The anchors are still visible (and can be edited) when hovering over the connection

## 5.3 Notifications

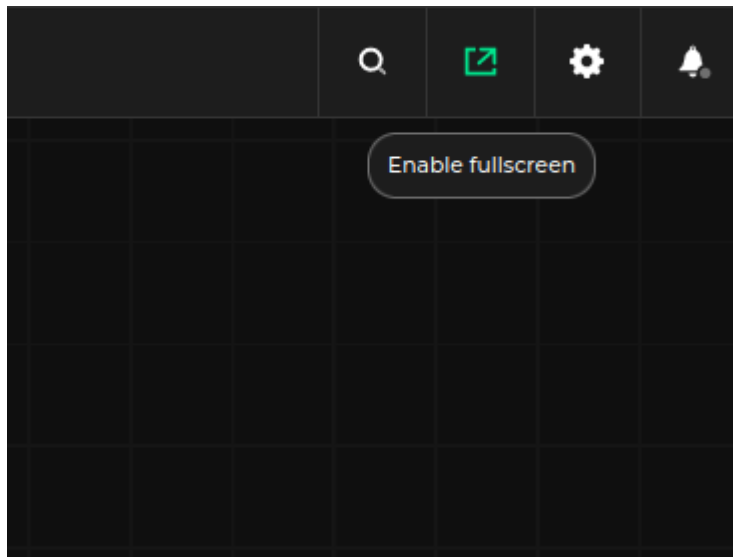


Pipeline Manager provides notifications describing errors occurring in:

- The front end, such as invalid input specification, or invalid dataflow
- The back end (see *Communication with an external application*)

During pipeline development or execution, notifications can display various messages to the user.

## 5.4 Full screen

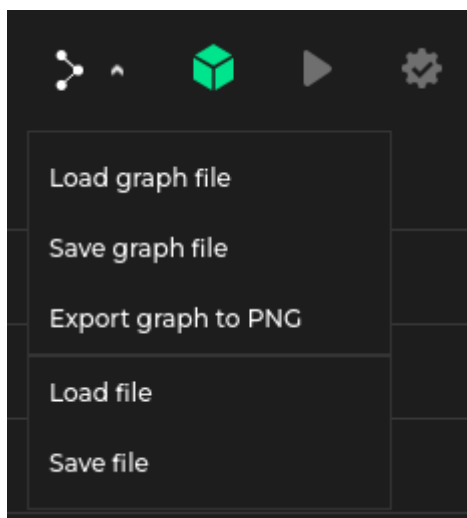


Near the settings tab, there is a full screen icon. Clicking it will expand the Pipeline Manager window to full screen. Note that this feature may be disabled during embedding Pipeline Manager in an external application, make sure to allow full screen mode.

For example, in the case of embedding Pipeline Manager in an iframe, the `allow="fullscreen"` attribute must be set:

```
<iframe src="https://url" allow="fullscreen">
  <p>Your browser does not support iframes.</p>
</iframe>
```

## 5.5 Editor menu



Depending on the application's mode (static-html or server-app), the following options will be available in the Pipeline Manager menu:

- Load specification - lets the user load a specification file describing the nodes that can appear in the graph (see *Specification format*)

### **Note**

It appears only in the static-html build mode, where a specification is not delivered by a third-party app.

- Load graph file - lets the user load a graph specification in Pipeline Manager's internal format (see *Dataflow specification*).
- Save graph file - Saves the graph currently stored in the editor into JSON using Pipeline Manager's internal format.
- Export graph to PNG - Saves the graph currently stored in the editor as PNG image
- Load file - lets the user load a file describing a graph in the native format supported by the third-party application using Pipeline Manager for visualization.

### **Note**

It appears only in the server-app build mode, since the third-party app performs conversion from its native format to the *Dataflow format*

- Save file - saves the current graph in a native format supported by the third-party application using Pipeline Manager for visualization.

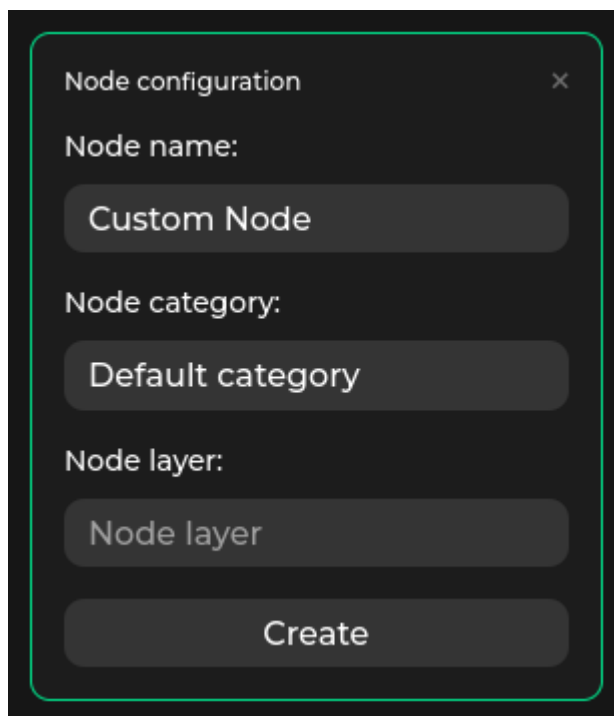
### **Note**

It appears only in the server-app build mode, since the third-party app performs conversion from the *Dataflow format* received from the editor to its native format.

## 5.6 Creating new node types

After toggling the `Modify nodes types` checkbox in settings, the special `New Node Type` becomes available in the node palette.

Drag and drop it to the editor to display the `Node configuration` menu.

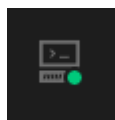


After the node is created, it will be added to the editor and available in the node palette.

The node type can be configured further from the editor, e.g. by adding properties and interfaces.

## 5.7 Working with the server

When Pipeline Manager works in the `server-app` mode, it is connected to an external application, making it possible to manipulate or execute a graph as well as save and load files in the application's native format.



Status of the connection is displayed in the upper right corner of the editor. The color of the icon indicates the status of the connection with the server:

- Red - No connection with the server.
- Green - Connection with the server is established.

External application may deliver additional features by providing additional buttons in the editor menu. More details can be found in the *Specification format* section.

## 5.8 URL parameters for the frontend

The frontend of Pipeline Manager provides a set of URL parameters that can be used to change the specification, graph or default behavior of the application. Those arguments need to be encoded - we need to escape all URL-specific characters. This can be achieved either with `urllib.parse.urlencode` in Python:

```
import urllib.parse

urllib.parse.urlencode({
    "spec": "https://github.com/antmicro/kenning-pipeline-manager/blob/main/
↪examples/sample-specification.json",
    "graph": "https://github.com/antmicro/kenning-pipeline-manager/blob/main/
↪examples/sample-dataflow.json"
})

# RESULT: 'spec=https%3A%2F%2Fgithub.com%2Fantmicro%2Fkenning-pipeline-manager
↪%2Fblob%2Fmain%2Fexamples%2Fsample-specification.json&graph=https%3A%2F
↪%2Fgithub.com%2Fantmicro%2Fkenning-pipeline-manager%2Fblob%2Fmain%2Fexamples
↪%2Fsample-dataflow.json'
```

or with `encodeURIComponent` in Javascript:

```
encodeURIComponent("https://github.com/antmicro/kenning-pipeline-manager/blob/
↪main/examples/sample-specification.json")

// RESULT: 'https%3A%2F%2Fgithub.com%2Fantmicro%2Fkenning-pipeline-manager%2Fblob
↪%2Fmain%2Fexamples%2Fsample-specification.json'
```

Available parameters are as follows:

- spec - URL to the specification, by default it can be a HTTP/HTTPS URL,
- graph - URL to the graph, by default it can be a HTTP/HTTPS URL,
- preview - starts Pipeline Manager in read only mode, without HUD,
- include - allows to provide includes for the specification,
- backend - connects Pipeline Manager to the specified backend regardless of the application mode.

When it comes to spec and graph, by default we can use following URI schemes:

- http://, https://
- relative:// - picks a path relative to the Pipeline Manager URL
- data:application, or data:application;base64, - JSON embedded as **Data URL**

To add more URI schemes, we need to define `VUE_APP_JSON_URL_SUBSTITUTES` variable holding a dictionary mapping scheme name to appropriate template/prefix.

The JSON for URL substitutes can look as follows:



```
{
  "examples": "https://github.com/antmicro/kenning-pipeline-manager/examples/{}"
}
```

And be later be referred to as `examples://sample-specification.json` in URL.

#### Note

The URI schemes can be passed to `pipeline_manager` build via `--json_url_specification` argument.

## 5.9 External frontend

These features are available for websites that embed Pipeline Manager using `iframe`. **POST requests** are used as a communication channel.

The usage example can be found in `pipeline_manager/frontend_tester/tester_api/index.html` (see **README** for build instructions). You can check a deployed instance in **Frontend-to-Frontend demo**.

### 5.9.1 Making API requests

There is a possibility to call Pipeline Manager **API** from the frontend level.

```
let iframe;

window.addEventListener('load', () => {
  iframe = document.getElementById(pipelineManagerIframe);
});

window.addEventListener('message', (event) => {
  // handling responses from Pipeline Manager
  const response = JSON.stringify(event.data, null, 4);
  console.log(response);
});

// sending specification_change request to Pipeline Manager's iframe
iframe.contentWindow.postMessage({ method: 'specification_change', params: {
  ↪specification } });
```

This piece of code opens Pipeline Manager in an `iframe` and provides it with a JSON object containing the request body corresponding to a `specification_change` **frontend endpoint**. Note that this method also supports **backend endpoints** and **external app endpoints** if external application is attached to Pipeline Manager.

Check **External Frontend Communication** for more details.

## 5.9.2 Serving External Application

A wrapping website may implement *External Application*, e.g.:

```
let iframe;

window.addEventListener('load', () => {
  iframe = document.getElementById(pipelineManagerIframe);
  iframe = iframe.contentWindow.postMessage({ method: 'register_external_
↪frontend' });
});

window.addEventListener('message', (event) => {
  const { data } = event;
  if (data.method !== undefined) {
    const result = handleExternalAppRequest(data);
    iframe.contentWindow.postMessage({ id: data.id, result });
  } else {
    handleResponse(data);
  }
});
```

Here, the code registers the website as an external application, and then sets up the listener for processing corresponding requests and responses.

Check *External Frontend Application* for more details.

## 5.10 Testing the front-end features

The best way to test the front-end features is to use the `pipeline_manager.frontend_tester.tester_client`, *Third-party server example*.

## COMMUNICATION WITH AN EXTERNAL APPLICATION

The communication with an external application is based on **JSON-RPC** messages.

Pipeline Manager supports two ways of defining external application:

- *External Backend Application*
- *External Frontend Application*

### 6.1 Communication protocol

The application layer protocol specifies two blocks:

- size - four first bytes form an unsigned integer and tell the size of the content of the message in bytes.
- content - the rest of the message holding additional data in JSON-RPC format described in *API Specification*.

The size and content are stored in big-endian.

#### 6.1.1 External Backend Application

This is a core concept of server-app Pipeline Manager mode.

Pipeline Manager implements a TCP server that is listening on a specified port and waiting for the client to connect. Pipeline Manager frontend sends JSON-RPC requests to this server through SocketIO, which redirects messages to connected client for specific actions described in this chapter. External application can also request actions from Pipeline Manager in similar manner.

#### Frontend / Backend / External application

By default, Pipeline Manager will wait for an external application to connect and then request the specification. If the connection is established successfully, Pipeline Manager frontend will check if external application is still connected every 0.5 second. Apart from that, both Pipeline Manager frontend and external application can send requests which pass through Pipeline Manager backend.

```
sequenceDiagram
    Note over Frontend,Backend: Pipeline Manager (SocketIO)
    Note over External App: Pipeline Manager Backend Communication
    Frontend->>Backend: status_get
    Backend->>Frontend: status
    Frontend->>+Backend: external_app_connect
    Note over Backend,External App: BSD socket
    External App->>Backend: connect_socket
    Backend->>-Frontend: MessageType.OK
    Frontend->>+Backend: specification_get
    Backend->>External App: specification_get
    External App->>Backend: specification
    Backend->>-Frontend: MessageType.OK + specification
    loop Every 0.5s
        Frontend->>Backend: status_get
        Backend->>Frontend: status
    end
    par External App API
        Frontend->>+Backend: request
        Backend->>+External App: redirected request
        External App->>-Backend: response
        Backend->>-Frontend: redirected response
    and Frontend API
        External App->>+Backend: request
        Backend->>+Frontend: redirected request
        Frontend->>-Backend: response
        Backend->>-External App: redirected response
    end
end
```

As Pipeline Manager part of communication is done with SocketIO, it is based on events, which are precisely defined for both sides and can trigger different actions.

Frontend listens to:

- **api** – received messages are JSON-RPC requests, they are validated with *specification*, executed and generated responds are resend,
- **api-response** – received messages are JSON-RPC responses, they are also validated and returned as result of previous request.

Backend implements the following events:

- **backend-api** – receives all JSON-RPC requests, runs methods and responds,
- **external-api** – redirects messages to external application through BSD socket.

On the other hand, communication between backend and external application is done through BSD socket. To manage this, both sides run socket listener as separate coroutine task, which waits for messages and responds or redirects them.

## External Frontend / Pipeline Manager

If the Pipeline Manager is wrapped with another frontend (see *Frontend Features* for details), the communication can be established with **POST requests**.

```
sequenceDiagram
    Note over External Frontend,Frontend: postMessage
    par Frontend API
        External Frontend->>Frontend: request
        Frontend->>External Frontend: response
    and Backend/External API
        External Frontend->>+Frontend: request
        Note over Frontend,Backend & External App: Pipeline Manager (SocketIO)
        Frontend->>+Backend & External App: redirected request
        Backend & External App->>-Frontend: response
        Frontend->>-External Frontend: redirected response
    end
```

## Request structure

Following communication structure diagram below, we have:

- blue lines describing *Backend API* request from frontend,
- red lines describing *External App API* request from frontend,
- purple lines describing *Frontend API* request from external application.

```

C4Deployment
  Deployment_Node(cf, "External Frontend", "") {
    Container(external-post-message, "postMessage")
  }
  Deployment_Node(pm, "Pipeline Manager", "") {
    Deployment_Node(front, "Frontend", "") {
      Container(front-post-message, "postMessage")
      Deployment_Node(socketio, "SocketIO", "") {
        Container(front-socket, "SocketIO")
        Deployment_Node(front-event, "Events", "") {
          Container(front-api, "api")
          Container(front-response-api, "api-response")
        }
      }
    }
  }
  Deployment_Node(back, "Backend", "") {
    Deployment_Node(flask, "Python-SocketIO events", "") {
      Container(flask-backend-api, "backend-api")
      Container(flask-external-api, "external-api")
    }
    Deployment_Node(flask-task, "Coroutine task", "") {
      Container(back-socket, "BSD socket listener")
    }
  }
}
Deployment_Node(pmbc, "Pipeline Manager Backend Communication", "") {
  Deployment_Node(pmbc-socket, "Socket", "") {
    Container(pmbc-socket, "BSD socket")
    Deployment_Node(pmbc-task, "Coroutine task", "") {
      Container(pmbc-listener, "BSD socket listener")
    }
  }
}
%% frontend to frontend
BiRel(external-post-message, front-post-message, "JSON-RPC requests and
frontend responses")
Rel(front-post-message, front-socket, "Redirected requests")
Rel(front-response-api, external-post-message, "Redirected responses")
UpdateRelStyle(external-post-message, front-post-message, $offsetX="-55",
$offsetY="-30")
UpdateRelStyle(front-post-message, front-socket, front-post-message,
$offsetX="-55", $offsetY="-30")
%% frontend to backend request
Rel(front-socket, flask-backend-api, "JSON-RPC request")
Rel(flask-backend-api, front-response-api, "JSON-RPC response")
UpdateRelStyle(front-socket, flask-backend-api,
$lineColor="var(--md-code-hl-keyword-color)",
$textColor="var(--md-code-hl-keyword-color)", $offsetY="-15")
UpdateRelStyle(flask-backend-api, front-response-api,
$lineColor="var(--md-code-hl-keyword-color)",

```

(continues on next page)

(continued from previous page)

```
$textColor="var(--md-code-hl-keyword-color)", $offsetX="-50", $offsetY="55")
  %% frontend to external app request
  Rel(front-socket, flask-external-api, "JSON-RPC request")
  Rel(flask-external-api, pmhc-listener, "Redirected requests and responses")
  Rel(pmhc-listener, back-socket, "JSON-RPC response")
  Rel(back-socket, front-response-api, "Redirected response")
  UpdateRelStyle(front-socket, flask-external-api,
$lineColor="var(--md-code-hl-number-color)",
$textColor="var(--md-code-hl-number-color)")
  UpdateRelStyle(flask-external-api, pmhc-listener,
$lineColor="var(--md-code-hl-name-color)",
$textColor="var(--md-code-hl-name-color)", $offsetX="-10", $offsetY="-10")
  UpdateRelStyle(pmhc-listener, back-socket,
$lineColor="var(--md-code-hl-number-color)",
$textColor="var(--md-code-hl-number-color)", $offsetX="25", $offsetY="-25")
  UpdateRelStyle(back-socket, front-response-api,
$lineColor="var(--md-code-hl-number-color)",
$textColor="var(--md-code-hl-number-color)", $offsetX="-10", $offsetY="20")
  %% external app to frontend request
  Rel(pmhc-socket, back-socket, "JSON-RPC request")
  Rel(back-socket, front-api, "Redirected request")
  Rel(front-api, flask-external-api, "JSON-RPC response")
  Rel(pmhc-listener, pmhc-socket, "Received response")
  UpdateRelStyle(pmhc-socket, back-socket,
$textColor="var(--md-code-hl-function-color)",
$lineColor="var(--md-code-hl-function-color)", $offsetX="-10", $offsetY="15")
  UpdateRelStyle(back-socket, front-api,
$textColor="var(--md-code-hl-function-color)",
$lineColor="var(--md-code-hl-function-color)", $offsetX="45")
  UpdateRelStyle(front-api, flask-external-api,
$textColor="var(--md-code-hl-function-color)",
$lineColor="var(--md-code-hl-function-color)", $offsetX="-65", $offsetY="-30")
  UpdateRelStyle(pmhc-listener, pmhc-socket,
$textColor="var(--md-code-hl-function-color)",
$lineColor="var(--md-code-hl-function-color)", $offsetX="5", $offsetY="-30")
  UpdateLayoutConfig($c4ShapeInRow="1", $c4BoundaryInRow="3")
```

### 6.1.2 External Frontend Application

If the Pipeline Manager is used as an `iframe`, a website that wraps it can implement an *External Application*.

Similarly to *External Frontend Communication*, data transfer relies on **POST requests**: both requests and responses use *MessageEvent* (see *Frontend Features* for details).

This application type can be enabled dynamically via dedicated `register_external_frontend` *Frontend API* request. This is possible for both static-html and server-app modes. Currently, in the latter case, external backend application will be overridden with the frontend one.

## Request structure

```
sequenceDiagram
    Note over External Frontend,Pipeline Manager Frontend: postMessage
    par Frontend API
        External Frontend->>Pipeline Manager Frontend: request
        Pipeline Manager Frontend->>External Frontend: response
    and External API
        Pipeline Manager Frontend->>External Frontend: request
        External Frontend->>Pipeline Manager Frontend: response
    end
```

## 6.2 Response messages types - sent by the external application

### 6.2.1 OK

Message of type OK (0) is used to indicate a success and optionally includes an answer to a previous request. Its content may vary depending on the request type.

### 6.2.2 ERROR

Message of type ERROR (1) is used to indicate a failure and optionally includes an answer to a previous request. Its content may vary depending on the answered request.

### 6.2.3 PROGRESS

Message of optional type PROGRESS (2) is used to inform Pipeline Manager about the status of a running dataflow. The PROGRESS message type can only be used once a message of type RUN is received and can be sent multiple times before sending a final response message of type either ERROR or OK that indicates the end of the run. The progress information is conveyed in content using a number ranging 0 - 100 encoded in UTF-8 that signals the percentage of completion of the run. See [RUN](#) for more information.

### 6.2.4 WARNING

Message of type WARNING (3) is used to indicate a success but also alerts of a condition that might cause a problem in the future. It optionally includes an answer to a previous request. Its content may vary depending on the answered request.

## 6.3 API Specification

### 6.3.1 Frontend API

#### frontend\_status\_get

Returns status of the frontend

- **params:** Refer to [null\\_or\\_empty](#).
- **result** (*object*): Description of frontend status. Cannot contain additional properties.
  - **status** (*string, required*): Status.



### frontend\_specification\_get

Returns specification used by frontend

- **params:** Refer to [null\\_or\\_empty](#).
- **result:** Refer to [specification\\_object](#).

### graph\_get

Returns dataflow used by frontend

- **params:** Refer to [null\\_or\\_empty](#).
- **result:** Refer to [dataflow\\_object](#).

### node\_get

Procedure to read node state

- **params**
  - **All of**
    - \* : Refer to [node\\_object](#).
- **result** (*object*): Cannot contain additional properties.
  - **node** (*object, required*): State of the node.

### properties\_get

Procedure to alter values of a node

- **params** (*object*): Cannot contain unevaluated properties.
  - **All of**
    - \* : Refer to [node\\_object](#).
  - **properties** (*[array, null]*): Properties which values are going to be read specified by either a name of id. If undefined then all properties are returned.
    - \* **Items** (*object*)
      - **One of**
        - *object*: Cannot contain additional properties.
        - **id** (*string, required*): Id of the property to read.
        - *object*: Cannot contain additional properties.
        - **name** (*string, required*): Name of the property to read.
- **result** (*array*)
  - **Items** (*object*): Values of the searched properties. Cannot contain additional properties.
    - \* **id** (*string, required*): Id of the property.
    - \* **name** (*string, required*): Name of the property.
    - \* **value**: Value of the property. Its type depends on the property.

### properties\_change

Procedure to alter properties of a node

- **params:** Refer to `properties_diff`.
- **result:** Refer to `null_or_empty`.

### position\_change

Procedure to alter position of a node

- **params:** Refer to `position_diff`.
- **result:** Refer to `null_or_empty`.

### nodes\_change

Procedure to add and delete nodes

- **params:** Refer to `nodes_diff`.
- **result:** Refer to `null_or_empty`.

### connections\_change

Procedure to add and delete connections

- **params:** Refer to `connections_diff`.
- **result:** Refer to `null_or_empty`.

### graph\_change

Procedure to add and delete nodes

- **params** (*object*): Cannot contain additional properties.
  - **dataflow:** Refer to `dataflow_or_url`.
  - **loadingScreen:** Refer to `loadingScreen`.
- **result:** Refer to `null_or_empty`.

### progress\_change

Notification with progress of job ran by external application

- **params** (*object*): Parameters for notification. Cannot contain additional properties.
  - **method** (*string, required*): Name of the method used to run the job.
  - **progress** (*number, required*): Progress of job ran by external application. If between 0 and 100 progress will be set and if -1, animation will run.

### metadata\_change

Updates the editor's metadata

- **params:** Refer to `metadata`.

## viewport\_center

Center the editor

- **params:** Refer to [null\\_or\\_empty](#).

## terminal\_add

Create new terminal instance

- **params** (*object*): Parameters for request.
  - **name** (*string, required*): Unique name for the new terminal instance. This name is used for any communication.
  - **readonly** (*boolean*): Specifies whether the terminal should be read-only, or editable by user. Default: true.
- **result:** Refer to [null\\_or\\_empty](#).

## terminal\_remove

Create new terminal instance

- **params** (*object*): Parameters for request.
  - **name** (*string, required*): Name of the terminal to remove.
- **result:** Refer to [null\\_or\\_empty](#).

## terminal\_write

Writes a message line to a terminal instance. Allows to send both text and hterm's control sequences. For more details on available control sequences check [hterm Control Sequences](#).

### Warning

Since data is sent in JSON format, the hexadecimal values need to be escaped with `\u`, not `\x`, e.g. `\u001b[38:2:238:178:17mexample`.

- **params** (*object*): Parameters for request.
  - **name** (*string, required*): Name of the terminal to which the message is written.
  - **message** (*string, required*): Message to be written to the terminal.
- **result:** Refer to [null\\_or\\_empty](#).

## terminal\_clear

Clears a terminal with a given name

- **params** (*object*)
  - **name** (*string*): Name of the terminal to clear. If the name is not provided, the main terminal is cleared.
- **result:** Refer to [null\\_or\\_empty](#).

### terminal\_hide

Hides a terminal panel

- **params:** Refer to `null_or_empty`.
- **result:** Refer to `null_or_empty`.

### terminal\_show

- **params** (*object*): Parameters for request.
  - **name** (*string*): Name of the terminal to show. If the name is not provided, the main terminal is chosen.
- **result:** Refer to `null_or_empty`.

### terminal\_view

Configures the layout of the terminal panel

- **params** (*object*): Parameters for request.
  - **names** (*array*): Names of the terminal to show. If the parameter is empty, then the terminal view is unmodified.
    - \* **Items** (*string*): Name of the terminal.
  - **clearButton** (*boolean*): Specifies whether to 'Clear terminal' button should be displayed. If not specified, the state of the clear button is not modified.
- **result:** Refer to `null_or_empty`.

### terminal\_get\_instances

Configures the layout of the terminal panel

- **params:** Refer to `null_or_empty`.
- **result** (*array*): Existing terminals.
  - **Items** (*string*): Name of the terminal.

### notification\_send

Sends a notification to the frontend. It will display the message both in the notifications and in the default terminal

- **params** (*object*): Parameters for notification.
  - **type:** Type of the notification. Must be one of: ["error", "warning", "info"].
  - **title** (*string, required*): Title of the notification, appearing both in the terminal and notification.
  - **details** (*string, required*): Details of the notification, displayed only in the terminal.
- **result:** Refer to `null_or_empty`.

### specification\_change

Procedure to update specification

- **params** (*object*)
  - **specification**: Refer to [specification\\_or\\_url](#).
  - **urloverrides**: Refer to [urloverrides](#).
  - **loadingScreen**: Refer to [loadingScreen](#).
  - **tryMinify**: Refer to [tryMinify](#).
- **result**: Refer to [null\\_or\\_empty](#).

### specification\_graph\_change

Procedure to update specification and dataflow

- **params** (*object*): Cannot contain additional properties.
  - **specification**: Refer to [specification\\_or\\_url](#).
  - **dataflow**: Refer to [dataflow\\_or\\_url](#).
  - **urloverrides**: Refer to [urloverrides](#).
  - **loadingScreen**: Refer to [loadingScreen](#).
  - **tryMinify**: Refer to [tryMinify](#).
- **result**: Refer to [null\\_or\\_empty](#).

### specification\_preprocess

Handles preprocessor specification directives

- **params** (*object*)
  - **specification**: Refer to [specification](#).
  - **unmarkNewNodes** (*boolean*): Decides whether to remove built-in style of new nodes from specification.
  - **urloverrides**: Refer to [urloverrides](#).
  - **tryMinify**: Refer to [tryMinify](#).
- **result**
  - Any of
    - \* : Refer to [specification\\_object](#).
    - \* : Refer to [null\\_or\\_empty](#).

### nodes\_highlight

Procedure to highlight specified nodes in the editor

- **params**: Refer to [highlight\\_diff](#).
- **result**: Refer to [null\\_or\\_empty](#).

### register\_external\_frontend

Register external frontend in the Pipeline Manager frontend, available only through 'postMessage' interface

- **params:** Refer to `null_or_empty`.
- **result:** Refer to `null_or_empty`.

### 6.3.2 Backend API

#### status\_get

Returns status of connection with external application

- **params:** Refer to `null_or_empty`.
- **result** (*object*): Description of backend status. Cannot contain additional properties.
  - **status** (*object, required*): Status. Cannot contain additional properties.
    - \* **connected** (*boolean, required*): Connection with external application.

#### external\_app\_connect

Request to wait till connection with external application is established

- **params:** Refer to `null_or_empty`.
- **result:** Refer to `empty`.

#### connected\_frontends\_get

Request the number of connected frontends

- **params:** Refer to `null_or_empty`.
- **result** (*object*): Cannot contain additional properties.
  - **connections** (*number, required*): Number of connections.

### 6.3.3 External App API

#### specification\_get

Request specification used by external application

- **params:** Refer to `null_or_empty`.
- **result:** Refer to `external_endpoint_return_type`.

#### app\_capabilities\_get

Request external application capabilities

- **params:** Refer to `null_or_empty`.
- **result:** Refer to `navbar_items`.

## dataflow\_import

Request to convert dataflow in external app to Pipeline Manager format and import it

- **params** (*object*): Parameters for request. Cannot contain additional properties.
  - **external\_application\_dataflow** (*string, required*): Dataflow in external application format. If loaded file cannot be represented as text, it will be sent as base64 string. To make sure received data are in readable format `convert_message_to_string` from `pipeline_manager_backend_communication.utils` can be used. Optionally, for conversion to bytes `convert_message_to_bytes` function is available.
  - **mime** (*string, required*): MIME type.
  - **base64** (*boolean, required*): Specifies whether `external_application_dataflow` is in base64 format.
- **result**: Refer to `external_endpoint_return_type`.

## dataflow\_validate

Request external application to validate dataflow

- **params**: Refer to `dataflow_object`.
- **result**: Refer to `external_endpoint_return_type`.

## dataflow\_run

Request external application to run dataflow

- **params**: Refer to `dataflow_object`.
- **result**: Refer to `external_endpoint_return_type`.

## dataflow\_stop

Request external application to run dataflow

- **params** (*object*): Parameters for stopping dataflow. Cannot contain additional properties.
  - **method** (*string, required*): Name of the method used to start run.
- **result**: Refer to `external_endpoint_return_type`.

## dataflow\_export

Request external application to export dataflow

- **params**: Refer to `dataflow_object`.
- **result** (*object*): Common type returned by external app. Cannot contain additional properties.
  - **type** (*number, required*): `MessageType` specifying success or error.
  - **content** (*[object, string]*): Exported dataflow, which is then saved by the frontend user. Should be either a json object, or a base64 encoded string. If any error occurred then it should contain a proper message.
  - **filename** (*string*): Suggested filename used to save the file.

### frontend\_on\_connect

Request sent when Pipeline Manager frontend connects to backend and notifyWhenChanged option is enabled in the metadata section of specification.

- **params:** Refer to `null_or_empty`.
- **result:** Refer to `null_or_empty`.

### properties\_on\_change

Request sent when properties of any node change and notifyWhenChanged option is enabled in the metadata section of specification.

- **params:** Refer to `properties_diff`.
- **result:** Refer to `null_or_empty`.

### position\_on\_change

Request sent when position of any node changes and notifyWhenChanged option is enabled in the metadata section of specification.

- **params:** Refer to `position_diff`.
- **result:** Refer to `null_or_empty`.

### nodes\_on\_change

Request sent when node was added or deleted and notifyWhenChanged option is enabled in the metadata section of specification.

- **params:** Refer to `nodes_diff`.
- **result:** Refer to `null_or_empty`.

### connections\_on\_change

Request sent when connection was added or deleted and notifyWhenChanged option is enabled in the metadata section of specification.

- **params:** Refer to `connections_diff`.
- **result:** Refer to `null_or_empty`.

### graph\_on\_change

Request sent when whole dataflow changed, e.g. when dataflow is loaded, and notifyWhenChanged option is enabled in the metadata section of specification.

- **params:** Refer to `dataflow_object`.
- **result:** Refer to `null_or_empty`.

### specification\_on\_change

Request sent when whole specification changed, e.g. when node type is modified, and notifyWhenChanged option is enabled in the metadata section of specification.

- **params:** Refer to `specification_object`.



- **result:** Refer to [null\\_or\\_empty](#).

### [metadata\\_on\\_change](#)

Request sent when metadata was changed, and notifyWhenChanged option is enabled in the metadata section of specification.

- **params:** Refer to [metadata](#).
- **result:** Refer to [null\\_or\\_empty](#).

### [viewport\\_on\\_center](#)

Request sent when editor was centered and notifyWhenChanged option is enabled in the metadata section of specification.

- **params:** Refer to [null\\_or\\_empty](#).
- **result:** Refer to [null\\_or\\_empty](#).

### [terminal\\_read](#)

Request sent by the frontend when terminal received an input

- **params** (*object*): Parameters for request.
  - **name** (*string, required*): Name of the terminal to which the message was written. Terminal has to be not read-only.
  - **message** (*string, required*): Terminal input.
- **result:** Refer to [null\\_or\\_empty](#).

### [nodes\\_on\\_highlight](#)

Request sent when node was either selected or unselected and notifyWhenChanged option is enabled in the metadata section of specification.

- **params:** Refer to [highlight\\_diff](#).
- **result:** Refer to [null\\_or\\_empty](#).

### [button\\_click](#)

Request send when button node property was pressed

- **params** (*object*): Parameters for request.
  - **id** (*string, required*): ID of the property.
  - **value** (*[object, null], required*): Value of the property.
- **result:** Refer to [null\\_or\\_empty](#).

## 6.3.4 Common Types

### [empty](#)

- *object*: Empty object definition. Cannot contain additional properties.

### null\_or\_empty

- *[object, null]*: Empty or missing object definition. Cannot contain additional properties.

### node\_object

- *object*: Schema that identifies a node in a graph.
  - **graph\_id** (*string, required*): Id of the graph.
  - **node\_id** (*string, required*): Id of the node.

### urloverrides

- *object*: Mapping for URL replacement.
  - **^.\*\$** (*string*)

### loadingScreen

- *boolean*: Show loading screen. Default: false.

### tryMinify

- *boolean*: Try to minify specification if a dataflow or entry graph is provided. Default: false.

### dataflow\_or\_url

- *[object, string]*: JSON or link to the remote JSON with graph definition.

### dataflow\_object

- *object*: Definition containing dataflow object. Cannot contain additional properties.
  - **dataflow** (*object*): JSON with graph definition in PM format.

### specification

- *object*: JSON with specification definition in PM format.

### specification\_or\_url

- *[object, string]*: JSON or link to the remote JSON with specification.

### specification\_object

- *object*: Definition containing specification object. Cannot contain additional properties.
  - **specification**: Refer to [specification](#).

### properties\_diff

- *object*: Schema that represents differences in properties of the node. Cannot contain unevaluated properties.
  - **All of**
    - \* : Refer to [node\\_object](#).

- **properties** (*array, required*): Properties to change specified by either a name or id.

- \* **Items** (*object*)

- **One of**

- *object*: Cannot contain additional properties.
      - **id** (*string, required*): Id of the property to alter.
      - **new\_value**: New value of the property.
      - *object*: Cannot contain additional properties.
      - **name** (*string, required*): Name of the property to alter.
      - **new\_value**: New value of the property.

### position\_diff

- *object*: Schema that represent differences in position of the node. Cannot contain unevaluated properties.

- **All of**

- \* : Refer to **node\_object**.

- **position** (*object, required*): Position to change specified by either a name or id.

- \* **x** (*number*): X coordinate.

- \* **y** (*number*): Y coordinate.

### nodes\_diff

- *object*: Schema that represents nodes' differences in a graph.

- **graph\_id** (*string, required*): Id of the graph.

- **nodes** (*object, required*): .

- \* **added** (*array*): List with created nodes.

- **Items** (*object*): JSON with node definition.

- \* **deleted** (*array*): List with removed nodes.

- **Items** (*string*): ID of node to delete.

- **remove\_with\_connections** (*boolean*): Should node be removed with connections.

### connections\_diff

- *object*: Schema that represents connections' differences in a graph.

- **graph\_id** (*string, required*): Id of the graph.

- **connections** (*object, required*): .

- \* **added** (*array*): List with created connections.

- **Items** (*object*): JSON with connection definition.

- \* **deleted** (*array*): List with removed connections.

- **Items** (*object*): Connection defined with its beginning and end.
  - **from** (*string*): ID of output interface.
  - **to** (*string*): ID of input interface.

### highlight\_diff

- *object*: Schema that represents the difference in node selection in the editor.
  - **graph\_id** (*string, required*): Id of the graph.
  - **nodes** (*object, required*): .
    - \* **selected** (*array*): List with selected nodes.
      - **Items** (*string*): ID of selected node.
    - \* **unselected** (*array*): List with unselected nodes.
      - **Items** (*string*): ID of unselected node.

### metadata

- *object*: Type with PM metadata. Cannot contain additional properties.
  - **metadata** (*object, required*): JSON with metadata description in PM format.

### external\_endpoint\_return\_type

- *object*: Common type returned by external app. Cannot contain additional properties.
  - **type** (*number, required*): *MessageType* specifying success, error or progress.
  - **content** (*[object, string]*): Additional information, either message or dataflow.

### navbar\_items

- *array*: List of buttons displayed in the application navbar. Default: [].
  - **Items** (*object*): Description of a single button. Cannot contain additional properties.
    - \* **name** (*string, required*): Name displayed in the tooltip.
    - \* **stopName** (*string*): Name displayed in the tooltip when the corresponding procedure is running and is stoppable. If not specified simply Stop is added as a prefix to the name.
    - \* **iconName** (*string, required*): Name of the icon displayed in the navbar. Name should correspond to a file in the asset directory or to an icon in frontend/src/icons.
    - \* **procedureName** (*string, required*): Name of the procedure used when button is pressed. The currently displayed graph is used as an argument.
    - \* **allowToRunInParallelWith** (*array*): List of procedures which can be run when this navbarItem is still running. Default: [].
      - **Items** (*string*): procedureName of navbarItem.
    - \* **requireResponse** (*boolean*): Flag determining whether response from external application should be awaited when running the procedure. Default: true.

### 6.3.5 Custom procedures

External application can define new remote procedures, which will be called by custom *Navbar button*. To use it, procedure's name has to start with `custom_` prefix, e.g. `custom_simulate_design`.

Such remote procedures can be called from the frontend using custom Navbar buttons defined in the *metadata's navbarItems field*, e.g.:

```
{
  "name": "Simulate design",
  "stopName": "Stop simulation",
  "iconName": "Run",
  "procedureName": "custom_simulate_design"
}
```

Custom procedure has the same parameters and return type as `dataflow_run` method.

## 6.4 Implementing a Python-based client for Pipeline Manager

The communication described above is necessary to integrate an application with Pipeline Manager. The client needs to be able to read requests coming from Pipeline Manager and send proper responses.

For applications written in Python, you can use the `pipeline-manager-backend-communication` library. It implements an easy-to-use interface that is able to communicate with Pipeline Manager along with helper structures and enumerations.

The main structures provided by the `pipeline-manager-backend-communication` library are:

- `CommunicationBackend` - class that implements the functionality for receiving and sending messages.
- `MessageType` - enum used to easily distinguish message types.
- `Status` - enum that describes the current state of the client.

The following code is an example of how to receive requests and send responses to Pipeline Manager: As Pipeline Manager communication is based on JSON-RPC, application should implement method that can be requested. They are described in *Backend API*.

```
# Class containing all implemented methods
class RPCMethods:
    def specification_get(self) -> Dict:
        # ...
        return {'type': MessageType.OK.value, 'content': specification}

    # ...
```

Defined methods have to have appropriate (matching with specification) name, input and output.

```
# Function name matches with the dataflow_import endpoint from External App_
↪API
def dataflow_import(self, external_application_dataflow: str, mime: str, ...
```

(continues on next page)

(continued from previous page)

```
→base64: bool) -> Dict:
    # Function will receive one parameter, it's name has to be the same
    # as the one from API specification `params`.
    # Optional, but you can convert the received file to string format
    # with `convert_message_to_string`
    # from pipeline_manager_backend_communication.utils import (
    #     convert_message_to_string
    # )
    data_as_string = convert_message_to_string(
        external_application_dataflow,
        base64,
        mime
    )
    # ...
    # pipeline_manager_dataflow here is the converted input file to the
    # Pipeline Manager's graph representation
    return {
        'type': MessageType.OK.value,
        'content': pipeline_manager_dataflow
    }

def dataflow_validate(self, dataflow: Dict) -> Dict:
    # ...
    # Returned object has to match API specification `returns`
    return {'type': MessageType.OK.value}

def dataflow_run(self, **kwargs: Dict) -> Dict:
    # All params can also be retrieved as one dictionary
    print(kwargs['dataflow'])
    # ...
    return {'type': MessageType.OK.value}

# Custom procedure example
def custom_build(self, dataflow: Dict) -> Dict:
    # ...
    return {'type': MessageType.OK.value}
```

Moreover, every uncaught exception will be classified as error.

```
def dataflow_export(self, dataflow: Dict) -> Dict:
    # ...
    raise Exception('Something went very, very bad...')
```

RPC methods can also be asynchronous. It is automatically detected by server and awaited.

```
async def dataflow_stop(self) -> Dict:
    # ...
    return {'type': MessageType.OK.value}
```

Therefore, the following JSON-RPC error message will be returned to frontend application.

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "error": {
    "code": -3,
    "message": "Something went very, very bad...",
    "data": {}
  }
}
```

TCP client, that connects to Pipeline Manager using host and port parameters provided has to be created. It has to be initialized with object containing JSON-RPC methods.

```
host = '127.0.0.1'
port = 5000

# Creating a client instance with host and port specified
client = CommunicationBackend(host, port)
# Registering implemented methods and
# connecting to Pipeline Manager
await client.initialize_client(RPCMethods())
```

Once the connection is established, the application can start listening for the incoming requests.

```
await client.start_json_rpc_client()
```

These methods can be wrapped into the async function and run with `asyncio.run` function.

### 6.4.1 Sending JSON-RPC requests to Pipeline Manager

Sending requests is defined as coroutine which has to be awaited.

```
response = await client.request('graph_get')
```

This method sends *graph-get* request to frontend application and receive following response:

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "dataflow": {
      // ...
    }
  }
}
```

## SAMPLE THIRD-PARTY SERVER

Some front-end features can be tested with the `frontend_tester` application that acts as a third-party application in the `server-app` scenario. It allows the user to create graph representations of basic front end use cases.

To be able to test the front end, start with building the front end in the `server-app` mode. In the root of the project, run:

```
python -m pipeline_manager build server-app
```

### Note

If the specification requires additional assets (e.g. icons), provide them with `--assets-directory <path-to-assets>` flag, for example:

```
python -m pipeline_manager build server-app --assets-directory examples/sample-  
↪assets
```

Now, start the Pipeline Manager server application:

```
python -m pipeline_manager run --verbosity INFO
```

Once the server has started, it waits for the third-party application to connect to. Let's run the front end tester with:

```
python -m pipeline_manager.frontend_tester.test_client
```

After this, the entire testing setup should be up and running at `http://127.0.0.1:5000`. From there, it is possible to add mini scenarios mocking the work of the server to check various front-end features.

The implementation of the `tester_client` is documented and quite straightforward. You can also easily extend it to test other front-end (or back-end) features.



## SPECIFICATION BUILDER

Creating a specification, especially in external applications written in Python, can be significantly simplified with the `SpecificationBuilder` class. This tool:

- Provides methods to update the contents of the specification
- Provides an API that allows modifying the specification without worrying about the changes in the format
- Provides sanity checks of URLs, prevents duplicates and validates the specification with the frontend.

Note that this chapter only presents the initial steps of constructing a specification in Pipeline Manager. Check the *specification format* for some more comprehensive details.

### 8.1 Example usage of the `SpecificationBuilder`

#### 8.1.1 Creating the builder

Import the package, specify the specification version to use and create an instance:

```
from pathlib import Path
from pipeline_manager.specification_builder import SpecificationBuilder

SPECIFICATION_VERSION = '20250623.14'
ASSETS_DIRECTORY = Path("./assets")

specification_builder = SpecificationBuilder(
    spec_version=SPECIFICATION_VERSION,
    assets_dir=ASSETS_DIRECTORY,
    check_urls=True
)
```

`assets_dir` is a path to additional assets (icons, visualizations) - it is used during validation to inform the user that e.g. paths provided in the specification do not have corresponding files. `check_urls` determines whether the `SpecificationBuilder` should check URLs provided in the specification for availability and raise errors when pages are not found.

### 8.1.2 Creating node types

Node types can be created with:

```
specification_builder.add_node_type(  
    name='Digital camera',  
    category='Video provider'  
)
```

This method requires a unique name of the node type. We can also specify category, parent classes, layer and abstract fields. Follow the documentation of the class for more details.

### 8.1.3 Add node type “as category”

To create a node type that can act as a category grouping the child node types, we can use `add_node_type_as_category`:

```
specification_builder.add_node_type_as_category(  
    categoryname='Filter',  
)
```

To add a node that extends this node type as category, just provide its name in the `extends` field in the child node:

```
specification_builder.add_node_type(  
    name='GaussianFilter',  
    extends='Filter'  
)
```

### 8.1.4 Adding interfaces to node types

The following snippets show how to add interfaces to node types. Note that one or more values can be passed to the `interfacetype` argument, signifying which types of other interfaces can be connected to it.

```
specification_builder.add_node_type_interface(  
    name='Digital camera',  
    interfacename='Video output',  
    interfacetype='video',  
    direction='output'  
)  
  
specification_builder.add_node_type_interface(  
    name='Filter',  
    interfacename='Video',  
    interfacetype=['video', 'filtered_output'],  
    direction='input'  
)  
  
specification_builder.add_node_type_interface(  
    name='Filter',  
    interfacename='Output',
```

(continues on next page)

(continued from previous page)

```
)  
interfacetype='filtered_output',  
direction='output',  
)
```

### 8.1.5 Adding properties to node types

To create a property in Pipeline Manager, we need to pass three required arguments:

- name - the name of the property
- type - the type of the property, see *specification format* for possible values
- default - specifies a default value, its type depends on the type of the property

```
specification_builder.add_node_type_property(  
    name='Digital camera',  
    proptype='Focal length',  
    proptype='slider',  
    min=14,  
    max=135,  
    default=50,  
    description='Value signifying a camera\'s angle of view'  
)  
  
specification_builder.add_node_type_property(  
    name='Filter',  
    proptype='Kernel size',  
    proptype='select',  
    values=['3x3', '5x5', '7x7'],  
    default='3x3',  
    description='Small matrix used for convolution'  
)
```

There are some additional arguments, available in the *specification format*.

### 8.1.6 Adding node type descriptions

Adding descriptions is just a matter of providing a node name and a description string.

```
specification_builder.add_node_description(  
    name='Digital camera',  
    description='Camera providing digital video output'  
)
```

### 8.1.7 Creating interface groups

All interfaces belonging to a group must exist. Members of an interface group may be:

- interface range(s)
- single interface(s)

For example, to create an interface range pins with indices 0–99:

```
specification_builder.add_node_type_interface(  
    name="Digital camera",  
    interface_name="pins",  
    array=[0, 100],  
    direction=Direction.INOUT.value,  
)
```

#### Note

Direction enum may be imported with `pipeline_manager.dataflow_builder.data_structures import Direction`.

Adding an interface group is a two-step process:

1. Create an empty interface group.

```
specification_builder.add_node_interface_group(  
    node_type_name="Digital camera",  
    interface_group_name=f"lower_half",  
    direction=Direction.INOUT,  
)
```

2. Add interfaces and/or interface ranges to the newly created interface group.

```
specification_builder.add_interface_to_group_interface(  
    node_type_name="Digital camera",  
    interface_group_name=f"lower_half",  
    actual_interface_name="pins",  
    direction=Direction.INOUT,  
    index_or_indices=(0, 50),  
)
```

### 8.1.8 Adding metadata

Metadata specifies additional editor options, like `connectionStyle` or `movementStep`. All parameters are defined in the *specification format*.

```
specification_builder.metadata_add_param(  
    paramname='connectionStyle',  
    paramvalue='orthogonal'  
)
```

### 8.1.9 Adding URLs to node types

To add a URL, first specify the URL group, which is part of the metadata. The icon argument can be a path to an asset file or a link.

```
specification_builder.metadata_add_url(  
    groupname='wikipedia',  
    displayname='Wikipedia',
```

(continues on next page)

(continued from previous page)

```
icon='https://en.wikipedia.org/static/favicon/wikipedia.ico',  
url='https://en.wikipedia.org/'  
)
```

Then, add the node URL by defining the suffix that will be appended to the group URL.

```
specification_builder.add_node_type_url(  
    name='Digital camera',  
    urlgroup='wikipedia',  
    suffix='wiki/Digital_camera'  
)
```

### 8.1.10 Constructing and validating the specification

This function will generate and check the validity of the specification:

```
specification = specification_builder.create_and_validate_spec()
```

The possible arguments are:

- `workspacedir` - The path to the workspace directory for Pipeline Manager, the same that has been provided for the build script
- `fail_on_warnings` - Determines whether the validation should fail on warnings
- `dump_spec` - A path to where the specification should be dumped as a file before validation. Useful for debugging purposes.
- `sort_spec` - Return specification that has introduced ordering to dictionaries and lists. In lists containing dictionaries the unique field of each dictionary is used to sort entries.

The created specification upon successful run should contain a full specification based on `SpecificationBuilder`. It is a regular Python dictionary that can be saved to a JSON file using the `json.dump` method.

#### Note

To make sure the entries in the specification are sorted (mostly dictionaries) after using `sort_spec`, in case of Python's `json.dump` and `json.dumps` methods use `sort_keys=True`, e.g.:

```
json.dumps(spec, sort_keys=True, indent=4)
```

## 8.2 SpecificationBuilder documentation

```
class pipeline_manager.specification_builder.SpecificationBuilder(spec_version: str,  
    assets_dir: Path |  
    None = None,  
    check_urls: bool =  
    False)
```

Creates a specification file and checks validity of files.

This class allows to:

- Create and modify specification entries using simple API
- Merge existing specifications with some consistency checks
- Check URLs to remote resources in terms of availability
- Checks correctness of paths to assets from the given directory

This class also performs very strict checking, e.g. it does not allow creating duplicates. In case of any errors spotted it raises a `SpecificationBuilderException`.

**add\_include**(*include*: *str*, *style*: *str* | *list[str]* | *None* = *None*)

Adds include defined by url to the specification.

#### Parameters

**include** : *str*

URL to the specification to include

**style** : *Optional[Style]*

Style of included nodes

#### Raises

**SpecificationBuilderException** – Raised when include already exists.

**add\_include\_subgraph**(*dataflow*: *dict[tuple[str, str], str]*)

Adds dataflow to the include subgraph defined by url.

#### Parameters

**dataflow** : *Dict[Tuple[str, str], str]*

Dataflow to include. Keys are tuples of node names and node urls

#### Raises

**SpecificationBuilderException** – Raised when included subgraph already exists

**add\_interface\_to\_group\_interface**(*node\_type\_name*: *str*, *interface\_group\_name*: *str*, *actual\_interface\_name*: *str*, *index\_or\_indices*: *int* | *tuple[int]*, *direction*: *Direction* = *Direction.INOUT*, *max\_connection\_count*: *int* | *None* = *None*, *side*: *Side* | *None* = *None*)

Add an interface or a range of interfaces to an existing interface group.

#### Parameters

**node\_type\_name** : *str*

Name of the node type.

**interface\_group\_name** : *str*

Name of the interface group.

**actual\_interface\_name** : *str*

Name of the actual interface to be split within groups.

**index\_or\_indices** : *Union[int, Tuple[int]]*

Either a single index of a new interface or [lower, upper) range of interface to be added to the group interface.

**direction : Direction, optional**

Direction of the interface. By default, Direction.INOUT.

**max\_connection\_count : Optional[int], optional**

Maximal number of connection of the interface or the range of interface. By default, None.

**side : Optional[Side], optional**

Side, where the interface or the group of interface should be located. By default, None.

#### Raises

- **TypeError** – Raised if: - an unsupported type of *index\_or\_indices* was provided, - a provided range of indices has size different than two.
- **ValueError** – Raised if: - there is no interface group present in the specification builder, - a group with *interface\_group\_name* does not exist.

**add\_node\_description(name: str, description: str)**

Sets description for the node if it is not defined.

#### Parameters

**name : str**

Name of the node type

**description : str**

Description

#### Raises

**SpecificationBuilderException** – Raised if description is already defined

**add\_node\_interface\_group(node\_type\_name: str, interface\_group\_name: str, direction: Direction = Direction.INOUT, interface\_type: str | None = None, interfaces: list[dict] | None = None)**

Add an empty interface group to a node type.

#### Parameters

**node\_type\_name : str**

Name of a node type.

**interface\_group\_name : str**

Name of a newly created interface group.

**direction : Direction**

Direction of the interface group, by default Direction.INOUT.

**interface\_type : Optional[str]**

Type of an interface group, by default None.

**interfaces : Optional[List[Dict]]**

A list of interfaces to be added to the interface group. It is safer to use *add\_interface\_to\_group\_interface* method to add interface to an interface group - it performs more detailed data validation. Each interface is a dictionary with at least two keys: *name* and *direction*.

### Raises

- **MissingInterfaceAttribute** – Raised if either *name* or *direction* is missing in *interfaces*.
- **ValueError** – Raised if *direction* has an illegal value.

**add\_node\_type**(*name*: str, *category*: str | None = None, *layer*: str | None = None, *extends*: str | list[str] | None = None, *abstract*: bool | None = False)

Adds a node type to the specification.

### Parameters

**name** : str

Name of the node type.

**category** : Optional[str]

Category of the node.

**layer** : Optional[str]

Name of the layer metatype.

**extends** : Optional[Union[str, List[str]]]

Base classes for the node type.

**abstract** : Optional[bool]

Tells if the type is abstract or not. Abstract types do not need to be complete, they are also not added to the final specification. They are templates for other classes.

### Raises

**SpecificationBuilderException** – Raised if the node type is redefined.

**add\_node\_type\_additional\_data**(*name*: str, *additionaldata*: Any)

Adds additional data to the node, in JSON-like format.

### Parameters

**name** : str

Name of the node type

**additionaldata** : Any

Any JSON-like construct

### Raises

**SpecificationBuilderException** – Raised if *additionalData* already exists.

**add\_node\_type\_as\_category**(*categoryname*: str, *categoryparent*: str = '', *layer*: str | None = None, *extends*: str | list[str] | None = None)

Adds a node type “as category” to the specification.

### Parameters

**categoryname** : str

Name of the newly added category

**categoryparent** : str

Path to the new category



**layer : Optional[str]**

Name of the layer metatype

**extends : Optional[Union[str, List[str]]]**

Base classes for the node type

**Raises**

**SpecificationBuilderException** – Raised when node type is redefined.

**add\_node\_type\_category(name: str, category: str)**

Adds a category to the given node. Raises an error if different category is already added.

**Parameters**

**name : str**

name of the node type

**category : str**

category for the node type

**Raises**

**SpecificationBuilderException** – Raised if there is already a node type category.

**add\_node\_type\_color(name: str, color: str)**

Sets color for the node.

**Parameters**

**name : str**

Name of the node

**color : str**

Color of the node

**add\_node\_type\_from\_spec(node)**

Adds single node type defined in JSON-like format.

**add\_node\_type\_icon(name: str, iconpath: dict | str) → None**

Adds icon for the node type.

**Parameters**

**name : str**

Name of the node type

**iconpath : Union[dict, str]**

Icon path

**add\_node\_type\_interface(name: str, interfacename: str, interfacetype: str | list[str] | None = None, direction: str = 'inout', dynamic: bool | list[int] = False, side: str | None = None, maxcount: int | None = None, override: bool | None = None, array: list[int] | None = None)**

Adds interface to the node type.

**Parameters**

**name : str**  
Name of the node type

**interfacename : str**  
Name of the interface

**interfacetype : Optional[Union[str, List[str]]]**  
List of matching types for interfaces

**direction : str**  
Direction of the connection, by default “inout”.

**dynamic : Union[bool, List[int]]**  
Determine whether a number of interfaces may be dynamically adjusted. By default, False.

**side : Optional[str]**  
On which side the interface should be placed by default

**maxcount : Optional[int]**  
The maximum connections to the given interface

**override : Optional[bool]**  
Determines whether interface should be overridden

**array : Optional[List[int]]**  
Creates an array of interfaces with given name. Accepts two integers - minimal and maximal value

#### Raises

**SpecificationBuilderException** – Raised when interface already exists.

**add\_node\_type\_parent**(**name**: str, **parent\_names**: str | list[str])

Adds a parent class to the node type. Raises an exception if the base name does not exist.

#### Parameters

**name : str**  
name of the node type

**parent\_names : Union[str, List[str]]**  
name of the base class, or list of base classes' names.

#### Raises

**SpecificationBuilderException** – Raised when base/parent class is not found.

**add\_node\_type\_property**(**name**: str, **propname**: str, **proptype**: str, **default**: Any, **description**: str | None = None, **min**: Any = None, **max**: Any = None, **values**: list[Any] | None = None, **dtype**: str | None = None, **override**: bool | None = None)

Adds property to the node.

#### Parameters

**name : str**  
Name of the node type

**propname : str**  
Name of the property

**proptype : str**  
Type of the property

**default : Any**  
Default value of the property

**description : Optional[str]**  
Optional description for the property

**min : Any**  
Minimal value

**max : Any**  
Maximal value

**values : Optional[List[Any]]**  
List of allowed values

**dtype : Optional[str]**  
Type of elements in property type is list

**override : Optional[bool]**  
Determines whether property should be overridden

#### Raises

**SpecificationBuilderException** – Raised when the property exists already.

**add\_node\_type\_property\_group**(**name**: str, **propgroupname**: str, **propname**: str, **proptype**: str, **default**: Any, **description**: str | None = None, **min**: Any = None, **max**: Any = None, **values**: list[Any] | None = None, **dtype**: str | None = None, **override**: bool | None = None)

Adds a property to a property group.

#### Parameters

**name : str**  
Name of the node type

**propgroupname : str**  
Name of the group of property

**propname : str**  
Name of the property

**proptype : str**  
Type of the property

**default : Any**  
Default value of the property

**description : Optional[str]**  
Optional description for the property

**min : Any**  
Minimal value

**max : Any**

Maximal value

**values : Optional[List[Any]]**

List of allowed values

**dtype : Optional[str]**

Type of elements in property type is list

**override : Optional[bool]**

Determines whether property should be overridden

#### **Raises**

**SpecificationBuilderException** – Raised when no properties are found that could be bound to the group.

**add\_node\_type\_style**(**name**: str, **style**: str | list[str])

Sets style for the node.

#### **Parameters**

**name : str**

Name of the node

**style : Style**

Style of the node

#### **Raises**

**SpecificationBuilderException** – Raised when the given style is not present egistered in metadata

**add\_node\_type\_subgraph\_id**(**name**: str, **subgraph\_id**: str)

Sets subgraph ID for the node.

#### **Parameters**

**name : str**

Name of the node

**subgraph\_id : str**

Subgraph ID of the node

**add\_node\_type\_url**(**name**: str, **urlgroup**: str, **suffix**: str)

Adds URL for the node type.

#### **Parameters**

**name : str**

Name of the node type

**urlgroup : str**

Name of the URL group the entry is defined for. URL group defines prefix, and icon for the URL.

**suffix : str**

Appended suffix to the URL group

#### **Raises**

**SpecificationBuilderException** – Raised if provided urlgroup does not exist.

**add\_subgraph\_from\_spec(subgraph)**

Adds subgraph defined in JSON-like format.

**create\_and\_validate\_spec**(workspacedir: Path | None = None, resolved\_specification: Path | None = None, fail\_on\_warnings: bool = True, sort\_spec: bool = False, dump\_spec: Path | None = None, skip\_validation: bool = False) → dict

Creates a specification and validates it using schema.

#### Parameters

**workspacedir : Optional[Path]**

Path to the workspace directory for Pipeline Manager

**resolved\_specification : Optional[Path]**

Path to specification that has have resolved inheritance, i.e. resolved 'extends' attributes. If none then the specification is not resolved.

**fail\_on\_warnings : bool**

Tells if the specification creation should fail on warnings

**sort\_spec : bool**

True if the entries in the specification should be sorted.

**dump\_spec : Optional[Path]**

Tells where the specification should be dumped to file before validation and resolving for debugging purposes.

**skip\_validation : bool**

Whether validation of the specification should be omitted.

#### Returns

Built specification, if successful

#### Return type

Dict

#### Raises

**SpecificationBuilderException** – Raised when specification is not valid or when warnings appeared.

**create\_property**(propname: str, proptype: str, default: Any, description: str | None = None, min: Any = None, max: Any = None, values: list[Any] | None = None, dtype: str | None = None, override: bool | None = None, stringify: bool = False) → dict | str

Creates and returns a property.

#### Parameters

**propname : str**

Name of the property

**proptype : str**

Type of the property

**default : Any**

Default value of the property

**description : Optional[str]**

Optional description for the property

**min : Any**

Minimal value

**max : Any**

Maximal value

**values : Optional[List[Any]]**

List of allowed values

**dtype : Optional[str]**

Type of elements in property type is list

**override : Optional[bool]**

Determines whether property should be overridden

**stringify : bool**

Determines whether property should be returned as a string

#### Returns

Creates a single property for the node type

#### Return type

Union[Dict, str]

**enable\_interface\_group\_by\_default**(*node\_name*: str, *interface\_group*: str, *direction*:  
Direction)

Add an interface group to default interface groups of a node type.

The default interface group contains a list of interface groups enabled by default. Adding an interface group there effectively enables the interface group from the specification.

#### Parameters

**node\_name : str**

Name of the node type.

**interface\_group : str**

Name of the interface group.

**direction : Direction**

Direction of the interface group.

#### Raises

**ValueError** – Raised if: - the node with the provided *node\_name* does not have any interface groups, - more than one interface group matches provided *interface\_group* and *direction*, - none of interface groups matches provided *interface\_group* and *direction*.

**get\_node\_description**(*name*: str) → str

Gets description for the node type.

#### Parameters

**name : str**

Name of the node type

### Returns

Description of the node type

### Return type

str

```
metadata_add_interface_styling(interfacename: str, interfacecolor: str | None = None,
                              interfaceconnpattern: str | None = None,
                              interfaceconncolor: str | None = None)
```

Adds interface styling to metadata.

**interfacename: str**

Name of the interface type

**interfacecolor: Optional[str]**

Color of the interface

**interfaceconnpattern: Optional[str]**

Interface connection line pattern

**interfaceconncolor: Optional[str]**

Color of the interface connection line

```
metadata_add_layer(name: str, nodelayers: str | list[str] | None = None, nodeinterfaces:
                  str | list[str] | None = None)
```

Adds nodes' layer to metadata.

### Parameters

**name : str**

Name of the layer

**nodelayers : Optional[Union[str, List[str]]]**

List of node layers in the layer

**nodeinterfaces : Optional[Union[str, List[str]]]**

List of interface types in the layer

### Raises

**SpecificationBuilderException** – Raised when provided layer already exists.

```
metadata_add_node_style(stylename: str, styleicon: str | dict | None = None, stylecolor:
                       str | None = None)
```

Adds node styling to metadata.

### Parameters

**stylename : str**

Name of the style

**styleicon : Optional[Union[str, Dict]]**

Icon of the style

**stylecolor : Optional[str]**

Color of the style

### Raises

**SpecificationBuilderException** – Raised when the style already exists

**metadata\_add\_param**(**paramname**: *str*, **paramvalue**: *Any*, **metadata**: *dict* | *None* = **None**)

Sets parameter in metadata. Modifies the metadata dictionary in place.

The following rules apply: \* If the parameter is a list and the value is a list, the parameter is extended by the value. \* If the parameter is a dictionary and the value is a dictionary, the parameter is recursively updated by the value. \* Otherwise, the parameter is set to the value.

#### Parameters

**paramname** : *str*

Name of the metadata parameter

**paramvalue** : *Any*

Value of the parameter

**metadata** : **Optional**[*Dict*]

Metadata dictionary to modify. if None, the class metadata is used

**metadata\_add\_url**(**groupname**: *str*, **displayname**: *str*, **icon**: *str*, **url**: *str*)

Adds URL group to metadata.

#### Parameters

**groupname** : *str*

Name of the URL group

**displayname** : *str*

Displayable name of the URL group

**icon** : *str*

Path in assets directory or URL to the icon

**url** : *str*

URL prefix for the URL group

#### Raises

**SpecificationBuilderException** – Raised when URL group already exists and differs in format with the existing entry.

**register\_category**(**categorypath**: *str*)

Adds a category to the set of available categories.

#### Parameters

**categorypath** : *str*

Category. Subcategories are added via path-like string.

**remove\_node\_type\_interface**(**name**: *str*, **interfacename**: *str*) → *None*

Removes interface from a selected node type.

#### Parameters

**name** : *str*

Name of the node type

**interfacename** : *str*

Name of the interface



**Raises**

**SpecificationBuilderException** – Raised when node type or interface does not exist.

**reset()**

Resets all fields for the specification.

**set\_node\_description(name: str, description: str)**

Sets description for the node type.

**Parameters**

**name : str**

Name of the node type

**description : str**

Description for the node

**update\_spec\_from\_other(otherspec: Any)**

Updates the specification from other specification. It can be also used to merge partial specifications.

**Parameters**

**otherspec : Any**

JSON-like structure with other specification

**Raises**

**SpecificationBuilderException** – Raised when “extends” in any node has non-existent classes

## DATAFLOW GRAPH BUILDER

Class `GraphBuilder` (available in `pipeline_manager.dataflow_builder.dataflow_builder`) provides a convenient Python API for creating and editing a dataflow graph.

### 9.1 Examples of `GraphBuilder` usage

#### 9.1.1 Creating `GraphBuilder`

In order to create an instance of `GraphBuilder`:

```
from pipeline_manager.dataflow_builder.dataflow_builder import GraphBuilder

builder = GraphBuilder(
    specification="examples/sample-specification.json",
    specification_version="20250623.14",
)
```

The code creates an instance of `GraphBuilder` with the `sample-specification.json` loaded.

#### 9.1.2 Obtaining a dataflow graph

A dataflow graph may be either loaded from a dataflow file or created.

To create a new empty dataflow graph:

```
from pipeline_manager.dataflow_builder.dataflow_graph import DataflowGraph

graph = builder.create_graph()
```

To load an existing specification from a file:

```
builder.load_graphs(dataflow_path='examples/sample-dataflow.json')
graph = builder.graphs[0]
```

In both cases, the dataflow graph:

- has to be compliant with the provided specification.
- is retrieved and may be edited with its public methods.

### 9.1.3 Creating a node

Having obtained the `DataflowGraph` object (in the previous examples, stored in the `graph` variable), a new node may be added to the graph in the following way:

```
from pipeline_manager.dataflow_builder.entities import Node

node = graph.create_node(
    name="LoadVideo", # Required.
    position=Vector2(0, 0), # Optional.
)
```

Only keyword parameters are allowed. The available parameters of the `create_node` method are attributes of the `Node` dataclass (which is located under `pipeline_manager.dataflow_builder.entities.Node`).

### 9.1.4 Getting a node

A dataflow graphs stores nodes. Thus, nodes are retrieved from the dataflow graph in the following way:

```
from pipeline_manager.dataflow_builder.dataflow_graph import AttributeType

nodes = graph.get(AttributeType.NODE, name="LoadVideo")
if len(nodes) > 0:
    node = nodes[0]
```

Only nodes with name equal to `LoadVideo` will be retrieved.

The `DataflowGraph.get` method retrieves not only nodes but other objects present in the `AttributeType` enum: connections, interfaces, and nodes. In this case, nodes are retrieved. Parameters other than type are filters, which specify what values should the obtained objects (here: nodes) have. Take a look at the next example to see such usage.

### 9.1.5 Getting a node matching multiple criteria

To get a node, which satisfies all the matching criteria, use the `get` method once again:

```
from pipeline_manager.dataflow_builder.entities import Vector2

[node] = graph.get(AttributeType.NODE, position=Vector2(0, 0), name="LoadVideo")
```

The code above puts in the `node` variable a node with name `LoadVideo` and position with coordinates `[0, 0]`.

### 9.1.6 Manipulating a node

After having a node retrieved from a graph, it is time to modify its attributes. For example, to change an instance name of the node, run:

```
node.instance_name = 'Instance name of a node'
```

To change the property of the node, `set_property` can be used:

```
try:
    # Use of `set_property` is recommended to set values of properties of a node.
    node.set_property('compression_rate', 1)
except KeyError:
    print('Cannot set compression_rate to 1 as the property does not exists.')
```

To move the node from its current position, use move:

```
# Move a node to [1000, 1000].
node.move(Vector2(1000, 1000))

# Move the node by 500 pixels to the right, relative to its previous position.
node.move(new_position=Vector2(500, 0), relative=True)
```

## 9.2 Exception Hierarchy

The dataflow builder module has custom exceptions ordered in a hierarchy of inheritance. The highest in the hierarchy is DataflowBuilderError.

The complete hierarchy of custom exceptions:

```
DataflowBuilderError
├── InterfaceError
│   ├── InterfaceCountError
│   ├── MissingInterfaceAttributeError
│   ├── MultipleInterfacesSelectedError
│   ├── NotDynamicInterfaceError
│   ├── NoInterfaceMatchedError
│   ├── InvalidInterfaceTypeError
│   ├── MissingInterfaceError
│   └── OutOfSpecificationInterfaceError
├── PropertyError
│   ├── InvalidPropertyValueError
│   ├── MissingPropertyError
│   └── InvalidPropertyTypeError
├── NodeError
│   ├── MissingNodeAttributeError
│   ├── ExtraNodeAttributeError
│   ├── OutOfSpecificationNodeError
│   ├── NodeLacksInterfacesError
│   └── MissingNodeError
├── GraphError
│   ├── DuplicateExternalNamesError
│   ├── InvalidPanningError
│   ├── ScalingOutOfBoundsError
│   ├── InvalidDataflowError
│   ├── NoGraphsError
│   └── GraphRenamingError
├── ConnectionError
│   └── InvalidDirectionError
```

(continues on next page)

(continued from previous page)

```

├── ConnectionExistsError
├── MismatchingInterfaceTypesError
├── GraphBuilderError
│   ├── NotSpecificationError
│   ├── NoGraphSelectedError
│   ├── ExternalValidatorError
│   └── SpecificationWithoutNodesError

```

## 9.3 Specification of GraphBuilder

```

class pipeline_manager.dataflow_builder.dataflow_builder.GraphBuilder(specification:
    Path | str |
    dict | SpecificationBuilder,
    specification_version:
    str,
    workspace_directory:
    Path | None =
    None)

```

Class for building dataflow graphs.

Each instance of the GraphBuilder must be associated with a single specification to ensure proper validation.

```

create_graph(based_on: Path | str | DataflowGraph | dict | None = None, identifier: str
    | None = None) → DataflowGraph

```

Create a dataflow graph and return its instance.

Create an instance of a dataflow graph, add it to the internal list and return it. The dataflow graph is based on the graph provided in *based\_on* parameter.

### Parameters

**based\_on** : Union[Path, str, DataflowGraph, Dict, None], optional  
 Dataflow graph, on which the new graph should be based on. When *Path* or *str*, it should be a path to dataflow file in a JSON format. When *Dict*, it should be a dictionary representation of the dataflow. When *DataflowGraph*, it should be a valid representation (as its deep copy will be added). When *None*, the new dataflow graph will not be based on anything, by default *None*.

**identifier** : Optional[str]  
 Either *id* or *name* of a dataflow graph. That identifier should uniquely identify a graph from the dataflow file provided in *based\_on* argument. If *based\_on* is either *None* or of type *DataflowGraph*, the value of *identifier* is not used. Defaults to the graph defined as *entryGraph* in the dataflow file.

### Returns

Instance a of a dataflow graph, preserved in the GraphBuilder.

**Return type***DataflowGraph***get\_graph\_by\_id**(*id*: str) → *DataflowGraph*

Get a graph by its ID.

**Parameters****id** : str

ID of the graph to retrieve.

**Returns**

Dataflow graph with the provided ID.

**Return type***DataflowGraph***Raises****NoGraphSelectedError** – Raised if the provided ID does not match any graph or matches more than one graph.**get\_graph\_by\_property**(*property\_name*: str, *property\_value*: Any) → *DataflowGraph*Get the first graph with *property\_name* equal to *property\_value*.**Parameters****property\_name** : strName of the property, e.g.: *id* or *name*.**property\_value** : Any

Exact value to be matched.

**Returns**

Dataflow graph satisfying the provided criteria.

**Return type***DataflowGraph***Raises**

- **MissingPropertyError** – Raised if any graph lacks the property.
- **NoGraphSelectedError** – Raised if the provided property name and values does not match any graph.

**get\_subgraph\_by\_name**(*name*: str) → *DataflowGraph* | list[*DataflowGraph*]

Get a subgraph by the name.

Get subgraph by either the name of a subgraph or name of the node containing the subgraph.

**Parameters****name** : str

Either name of the subgraph or name of the node containing the subgraph.

**Returns**

An single instance if one graph is matched, a list of instances if multiple graphs are matched.

**Return type**

Union[*DataflowGraph*, list[*DataflowGraph*]]

**Raises**

**NoGraphSelectedError** – Raised if name matches to more than one or none of graphs.

**get\_subgraphs()** → list[*DataflowGraph*]

Get a list of subgraphs.

Get a list of graphs, which are contained in some node, what makes them subgraphs.

**Returns**

A list of subgraphs.

**Return type**

List[*DataflowGraph*]

**load\_graphs**(*dataflow\_source*: Path | str | dict, *skip\_validation*: bool = **False**)

Load all dataflow graphs from a file.

**Parameters**

**dataflow\_source** : Union[Path, str, Dict]

Path to a dataflow graph or dataflow graph dictionary

**skip\_validation** : bool

Whether validation of a dataflow graph should be omitted.

**Raises**

- **InvalidDataflowError** – Raised if a dataflow graph could not be loaded.
- **NoGraphsError** – Raised if there are no graph in the dataflow file.

**load\_specification**(*specification*: Path | str | dict | *SpecificationBuilder*,  
*purge\_old\_graphs*: bool = **True**)

Load a specification from given file or dictionary to use in GraphBuilder.

The default behaviour is to remove loaded dataflow graphs when loading a new specification. That is due to the fact that a dataflow graph is closely linked with its specification. Notice that loading a specification overrides the old one.

**Parameters**

**specification** : Union[Path, str, Dict, *SpecificationBuilder*]

Path to specification file, dictionary or the builder directly

**purge\_old\_graphs** : bool, optional

Determine if dataflow graphs loaded to memory should be purged. It makes sense as after changing a specification dataflow graphs may no longer be valid. By default True.

**Raises**

**NotSpecificationError** – Raised if specification cannot be loaded or its file is invalid.

**save**(*json\_file*: Path, *skip\_validation*: bool = False)

Save graphs to a JSON file.

Saving also validates all the stored graphs.

#### Parameters

**json\_file** : Path

Path, where an output JSON file will be created.

**skip\_validation** : bool

Whether the validation of the script should be skipped or not.

**validate**()

Validate the entire dataflow file including all the included graphs.

#### Raises

**ExternalValidatorError** – Raised if an external validator failed to validate either a dataflow or specification file. An error message is provided by the external validator.

## 9.4 Specification of DataflowGraph

```
class pipeline_manager.dataflow_builder.dataflow_graph.DataflowGraph(builder_with_spec:
    Specification-
    Builder,
    builder_with_dataflow:
    Any, dataflow:
    dict[str, Any] |
    None = None)
```

Representation of a dataflow graph.

**create\_connection**(*from\_interface*: Interface | str, *to\_interface*: Interface | str, *connection\_id*: str | None = None) → InterfaceConnection

Create a connection between two existing interfaces.

The function performs numerous checks to verify validity of the desired connection, including if the connection already exists, interfaces' types are matching and so on. The connection is added, given it has passed these checks.

#### Parameters

**from\_interface** : Union[Interface, str]

Source interface, where data will flow from.

**to\_interface** : Union[Interface, str]

Destination interface, where data will flow to.

**connection\_id** : Optional[str]

Identifier of a connection. If not supplied, one will be generated.

#### Returns

Created connection added to the graph.

#### Return type

InterfaceConnection



### Raises

- **MissingInterfaceError** – Raised if either a source or destination interface does not belong to the graph.
- **InvalidDirectionError** – Raised if either a source interface direction has *input* direction or a destination interface direction is *output*.
- **MismatchingInterfaceTypesError** – Raised if a mismatch between source and destination interfaces' types occurs.

**create\_node**(*name*: str, *\*\*kwargs*) → *Node*

Create the node initialized with the supplied arguments.

Use this method instead of manually adding a node. Default values are taken from the specification, based on the provided *name* parameter. The default values may be overridden by the values supplied in *kwargs*. *id* is already initialized.

### Parameters

**name : str**

Name of a node, based on which default values will be derived.

***\*\*kwargs***

Keyword arguments to initialise a newly created node. Check attributes of *Node* dataclass, to find all available keys.

### Returns

The initialized node that belongs to the dataflow graph.

### Return type

*Node*

### Raises

- **InvalidMethodUsedError** – Raised if *subgraph* attribute was present in *kwargs*.
- **ExtraNodeAttributeError** – Raised if out-of-specification key is present in the *kwargs* directory.
- **OutOfSpecificationNodeError** – Raised if the provided name of the node does not exist in the specification.

**create\_subgraph\_node**(*name*: str, *subgraph\_id*: str, *\*\*kwargs*) → *Node*

Create a node with a subgraph.

A graph has to exist inside *GraphBuilder* instance before creating a node containing it as a subgraph.

### Parameters

**name : str**

Name of the node.

**subgraph\_id : str**

ID of the subgraph that contained by the node.

### Returns

Newly created node with a subgraph.

### Return type

*Node*

**get**(*type*: *AttributeType*, *\*\*kwargs*) → list[*Node*] | list[*InterfaceConnection*] | list[*Interface*]

Get items of a given type, which satisfy all the desired criteria.

Items are understood as either nodes or connections or interfaces. The function finds objects by eliminating these, which do not match the criteria. Thus, between all the criteria is *AND* logical operator.

### Parameters

**type** : *AttributeType*

Type of the output objects.

**\*\*kwargs**

Contains search criteria. Available: - Keys: the attributes of the object of the chosen type. - Values: values to be matched.

### Returns

List of items satisfying the criteria.

### Return type

Union[List[*Node*], List[*InterfaceConnection*], List[*Interface*]]

**get\_by\_id**(*type*: *AttributeType*, *id*: *str*) → *InterfaceConnection* | *Node* | *Interface* | None

Fast getter, which finds an item of a supplied type and with the provided id.

It has complexity of  $O(1)$  (except for interfaces) in juxtaposition to the *get* method, which iterates over all items.

### Parameters

**type** : *AttributeType*

Type of the output objects.

**id** : *str*

ID of the sought object.

### Returns

Either a single instance of *InterfaceConnection* or *Node* or *Interface* depending on the provided *type*. If does not exist, *None* is returned.

### Return type

Optional[Union[*InterfaceConnection*, *Node*, *Interface*]]

**property id** : *str*

Getter to obtain the ID of the graph.

**property panning** : *Vector2*

Getter to obtain the panning of the graph.

**property scaling** : *float*

Getter to obtain the scaling of the graph.

**to\_json**(*as\_str*: *bool* = *True*) → *str* | *dict*

Convert a dataflow graph to the JSON format.

### Parameters

**as\_str : bool, optional**

Determine return type. By default, True.

### Returns

Representation of the dataflow graph in JSON. If *as\_str* is *True*, JSON in str is returned. Otherwise, a Python dictionary is returned.

### Return type

Union[Dict, str]

## 9.5 Specification of Node

```
class pipeline_manager.dataflow_builder.entities.Node(specification_builder:
    SpecificationBuilder,
    for_subgraph_node: bool =
    False, **kwargs)
```

Representation of a node in a dataflow graph.

```
get(type: NodeAttributeType, **kwargs) → list[Property] | list[Interface]
```

Get either properties or interfaces matching criteria.

### Parameters

**type : NodeAttributeType**

Type of an item to retrieve.

**kwargs : Any**

Criteria, which items have to satisfy.

### Returns

List of either Property or Interface instances.

### Return type

Union[List[Property], List[Interface]]

```
get_interfaces_by_regex(pattern: str) → list[Interface]
```

Find interfaces with names matching *pattern*.

### Parameters

**pattern : str**

Regular expression used to match the names of the interfaces.

### Returns

List of interfaces matching *pattern* with their names.

### Return type

List[Interface]

```
get_number_of_dynamic_interfaces(interface_name: str) → int
```

Get a number of dynamic interfaces with a given name.

### Parameters

**interface\_name : str**

Name of the dynamic interface without any indices.

**Returns**

Number of dynamic interfaces.

**Return type**

int

**Raises**

- **NoInterfaceMatchedError** – Raised if multiple properties representing the same number of dynamic interfaces exist.
- **NotDynamicInterfaceError** – Raised if the provided interface is non-dynamic.

**get\_property**(*property\_name*: str) → Any

Convenient getter to get value of a property of a node.

**Parameters**

**property\_name** : str  
Name of the property.

**Returns**

Value of the property.

**Return type**

Any

**Raises**

**MissingPropertyError** – Raised if the supplied name is not associated with any property.

**static init\_subgraph\_node**(*specification\_builder*: SpecificationBuilder, *name*: str, *subgraph*: Any, \*\*kwargs) → Node

An alternative constructor of Node object.

Not for external use. Use *GraphBuilder.create\_subgraph\_node(...)* instead.

**Parameters**

**specification\_builder** : SpecificationBuilder  
Specification builder with loaded specification.

**name** : str  
Name of the node.

**subgraph** : DataflowGraph  
Dataflow graph that this node should contain.

**Returns**

Constructed Node object containing a subgraph.

**Return type**

Node

**Raises**

**RuntimeError** – Raised if *graph\_id* of a non-existent graph is provided.

**move**(*new\_position*: Vector2, *relative*: bool = False)

Change a position of the node.

Change a position of the node either in a relative manner or an absolute manner. Values are clamped to stay in the range of values.

#### Parameters

**new\_position** : Vector2

If *relative* is False, then this a new position. Otherwise, it is a displacement (movement) vector.

**relative** : bool, optional

Whether position should be calculated based on the previous one (True) or not (False), by default False.

**property name** : str

Getter to retrieve name of the node.

**set\_number\_of\_dynamic\_interfaces**(*interface\_name*: str, *new\_count*: int)

Set a number of dynamic interfaces to *new\_count*.

#### Parameters

**interface\_name** : str

Name of the dynamic interface.

**new\_count** : int

Number of the dynamic interfaces that should exist in the end.

#### Raises

- **ValueError** – Raised if: - new interface count is equal to or lower than 0, - interface name is associated with none of the interfaces,
- **MultipleInterfacesSelectedError** – Raised if the interface name is associated with more than one interface.
- **MissingInterfaceAttributeError** – Raised if the interface is not dynamic due to missing *dynamic* attribute.
- **NotDynamicInterfaceError** – Raised if the interface is not dynamic due to explicit definition in the specification.
- **InterfaceCountError** – Raised if either maximum interface count from the specification was exceeded or it has gone below a minimum interface count from the specification.

**set\_property**(*property\_name*: str, *property\_value*: Any) → None

Convenient setter to change value of a property of the node.

#### Parameters

**property\_name** : str

Name of the property.

**property\_value** : Any

New value of a property.

#### Return type

None

#### Raises

**MissingPropertyError** – Raised if property with the supplied *property\_named* was not found.

**to\_json**(*as\_str*=True) → str | dict

Convert a dataflow graph to the JSON format.

#### Parameters

**as\_str** : bool, optional

Determine return type. By default, True.

#### Returns

Representation of the dataflow graph in JSON. If *as\_str* is *True*, JSON in str is returned. Otherwise, a Python dictionary is returned.

#### Return type

Union[Dict, str]

## 9.6 Specification of Interface

```
class pipeline_manager.dataflow_builder.entities.Interface(name: str, direction:
    ~pipeline_manager.dataflow_builder.data_
    side:
    ~pipeline_manager.dataflow_builder.data_
    | None = None,
    side_position: int | None
    = None, external_name:
    str | None = None, id: str
    = <factory>, type:
    ~typing.List[str] =
    <factory>)
```

Representation of a node's interface.

**to\_json**(*as\_str*: bool = True) → str | dict

Convert a dataflow graph to the JSON format.

#### Parameters

**as\_str** : bool, optional

Determine return type. By default, True.

#### Returns

Representation of the dataflow graph in JSON. If *as\_str* is *True*, JSON in str is returned. Otherwise, a Python dictionary is returned.

#### Return type

Union[Dict, str]

## 9.7 Specification of Connection

Notice that name of the class representing connection is *InterfaceConnection*, not *Connection*.

```
class pipeline_manager.dataflow_builder.entities.InterfaceConnection(from_interface:
    ~pipeline_manager.dataflow_b
to_interface:
    ~pipeline_manager.dataflow_b
anchors: ~typ-
ing.List[~pipeline_manager.da
| None =
None, id: str =
<factory>)
```

Representation of a connection between two interfaces in a dataflow graph.

**to\_json**(*as\_str*: bool = **True**) → str | dict

Convert a dataflow graph to the JSON format.

#### Parameters

**as\_str** : bool, optional

Determine return type. By default, True.

#### Returns

Representation of the dataflow graph in JSON. If *as\_str* is *True*, JSON in str is returned. Otherwise, a Python dictionary is returned.

#### Return type

Union[Dict, str]

## 9.8 Specification of Property

```
class pipeline_manager.dataflow_builder.entities.Property(name: str, type: str | None
    = None, default: Any | None
    = None, value: Any | None
    = None, values: list[Any] |
None = None, min: str |
None = None, max: str |
None = None, dtype: str |
None = None, group:
dict[str, Any] | None =
None, id: str =
'78596bfb-65b1-423e-bade-ddcca6bebe05')
```

A property of a node.

**to\_json**(*as\_str*: bool = **True**) → dict | str

Convert a dataflow graph to the JSON format.

#### Parameters

**as\_str** : bool, optional

Determine return type. By default, True.

#### Returns

Representation of the dataflow graph in JSON. If *as\_str* is *True*, JSON in str is returned. Otherwise, a Python dictionary is returned.

#### Return type

Union[Dict, str]

## DATAFLOW AND SPECIFICATION FORMAT CHANGELOG

Formats of dataflows and specification is often updated with new features, that makes former formats incompatible with the new ones. This section lists all versions, related features and breaking changes.

### 10.1 20250623.14

Commit SHA - 8938cb326abc59ee8a7d506f9285954a8c6a027c

- Nodes:
  - direction is now optional, the default value is inout
  - Added color for both node specification and node dataflow instance allowing to customize the background color of the node title
  - Nodes no longer have to have category field - node types are created at the root level by default
  - Added dynamic field allowing to modify number of interfaces with dedicated node properties during runtime
- Properties:
  - Added multiline node property, which renders Markdown inside of node
  - Added readonly property field allowing to forbid value modification from UI
- Subgraphs:
  - Introduced new simplified implementation
  - Subgraphs do not have to be defined for graph nodes
  - Graph nodes can have properties
- Metadata:
  - Added urloverrides allowing to modify metadata URLs of included specifications
  - Added editableTypes field to metadata allowing to:
    - \* Add new node types from UI
    - \* Modify node types from UI with:
      - forms and buttons
      - YAML-based editor



- Added node styling:
  - styles in metadata defines icon and color for each named style:
    - \* icon is secondary, displayed on the right side of the node title
    - \* color is the same as in node specification, however, it has lower priority
  - style in the node type specification allows to use and combine styles defined in metadata
  - include supports new value - a pair {"url": "...", "style": "..."}

## 10.2 20240723.13

### Warning

This update changes the existing format of specification and dataflow.

#### Commit SHA - TODO

- Added includeGraphs field allowing to include graphs in the specification as subgraphs
- Added include field allowing to include partial specification to the specification
- Replaced graph field with graphs list
- The default graph now can be:
  - the first entry in the graphs list
  - the graph from graphs, where id of the graph is equal to entryGraph
- Replaced subgraphs in specification with graphs that have the exact same format as regular graphs
- Introduced externalName for graph nodes
- Fixed list type for properties - now they can be defined as lists in the graph, and display/modified as space-separated entries
- Improved Markdown-based rendering of description

## 10.3 20230523.12

### Warning

This update changes the existing format of specification and dataflow.

#### Commit SHA - 006998910f871da00c49ef00625e5d8bccf3a9c3

- Introduced changes in dataflow and specification formats:
  - Removed interfaces property from graphs, which were used to define interface nodes for subgraphs.

- Instead of “interface nodes”, a new property was added to regular interfaces of nodes, called `externalName`.
- `externalName`, when not empty, indicates that the interface is an outer interface of the subgraph, which should be exposed with the name from this field.

## 10.4 20230830.11

### Warning

This update changes the existing format of specification and dataflow.

Commit SHA - 0bee99e152a408650590bb6ea4210e96502b33ef

- Introduced changes in dataflow and specification formats:
  - `type` -> `name` for dataflows for consistency with specification naming
  - `name` -> `instanceName` for dataflows to make it more explicit and avoid conflicts
  - `type` -> `layer` for nodes in specification. This property is now also optional.
  - `nodeTypes` -> `nodeLayers` in `layers` keywords in metadata
  - `checkbox` -> `bool` for property type
  - `graphTemplateInstances` -> `subgraphs` for dataflow and specification.
- Added converter from version 20230824.10 to 20230523.12
- Added abstract parameter for node types to define abstract types.
- Added `isCategory` parameter that allows defining category nodes which serve both as a node and category.

## 10.5 20230824.10

Commit SHA - 6ce3bf106353bc58fb8e7217b9bc8aa7db9ebd20

- Choosing an entry graph that is rendered to the user when loading a subgraph dataflow is now possible using the `entryGraph` property.

## 10.6 20230824.9

Commit SHA - de31d01d9b6591993559e3b10851815f3320e545

- Introduced `description` keyword for nodes that allows displaying Markdown-based description in a sidebar.

## 10.7 20230818.8

Commit SHA - d4abcc80bce3e280120078a47d784eff92821a8a

- Introduced `group` keyword for `checkbox` property which allows defining groups of properties that can be disabled.

## 10.8 20230817.7

Commit SHA - 5946db06d8f42a33934a07fba95634aa8a70c78e

- Format of subgraph dataflows and specifications is redesigned. Details can be found in *Dataflow format* and *Specification format*.
- Dataflow
  - SubgraphIO was renamed to interfaces and its properties were changed.
  - Connections are now defined using connections property instead of nodeInterfaceId keyword.
- Specification
  - id property for nodes and connections in subgraphs is no longer required.
  - Connections are now defined using connections property instead of nodeInterface keyword.

## 10.9 20230809.6

Commit SHA - 59b04f8bc9cd0ce1cb757dcc4027750e1275d935

- Introduced anchors for connections, which is a list of coordinates that allows rendering custom connections shapes.

## 10.10 20230619.5

Commit SHA - 84fe865ca44b3a80b87a2be418eedc1d1c025ee4

- Introduced defaultInterfaceGroups groups of interfaces that are enabled by default.
- Implemented more verbose error logging both for interfaces and interface groups.

## 10.11 20230619.4

Commit SHA - bdc4b2fb998ed9d6318a676d4ac77f92c1603d39

- Introduced interfaceGroups and enabledInterfaceGroups keywords that allow defining groups of interfaces.
- Introduced arrays of interfaces that can be easily created using array keyword.
- Simplified dataflows as interfaces that have no connections are no longer saved into output .json files.

## 10.12 20230619.3

Commit SHA - 84559ae8327d7aa214c388200c5b48d112021679

- Introduced two new keywords to the specification's metadata:
  - backgroundSize - defines size of the background grid. It has strictly visual effects.
  - movementStep - minimal node movement step.

- Added `additionalData` for storing some node-related data, not relevant to Pipeline Manager
- Changed `connectionSide` to `side` in dataflow and specification
- type of interface now can be either a single string or a list of strings.
- Introduced optional metadata keyword `layout` specifying algorithm used for automatic node position calculation.
- Introduced `randomizedOffset` keyword that adds a random offset to connections so that a layout from complex graphs can be created easier. This value propagates into dataflows.

## 10.13 20230615.2

Commit SHA - 711ea7224e30d342924319c3964f1cb076939a29

- Introduced inheritance mechanism in specification - for each node type it is possible to specify `extends` list that provides information which are the base classes for a given type.

## 10.14 20230615.1

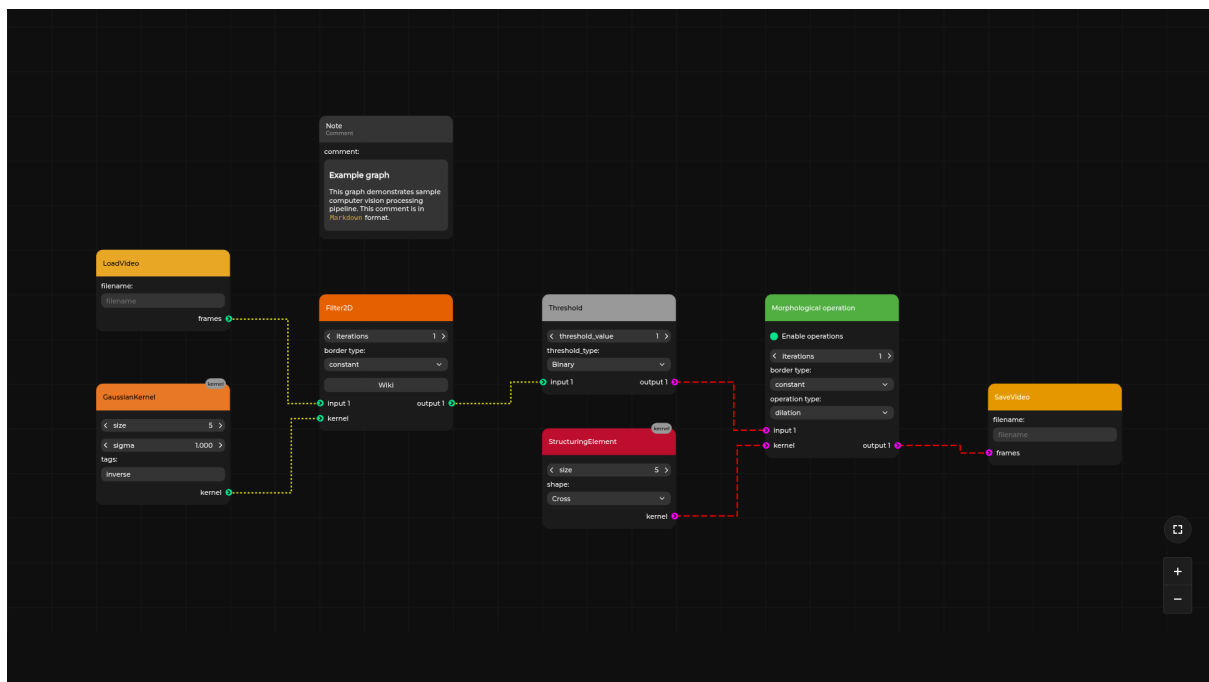
Commit SHA - 4e0cf99ccaa5bc3513804e8184b907ed0230985f

- Introduced versioning for dataflow and specification format, starting with version 20230615.1. Pipeline Manager saves the current version in a dataflow save so that when loading a dataflow an appropriate message is displayed.
- Implemented a dataflow format converter that can apply a range of patches so that an obsolete dataflow can be updated to the current format.

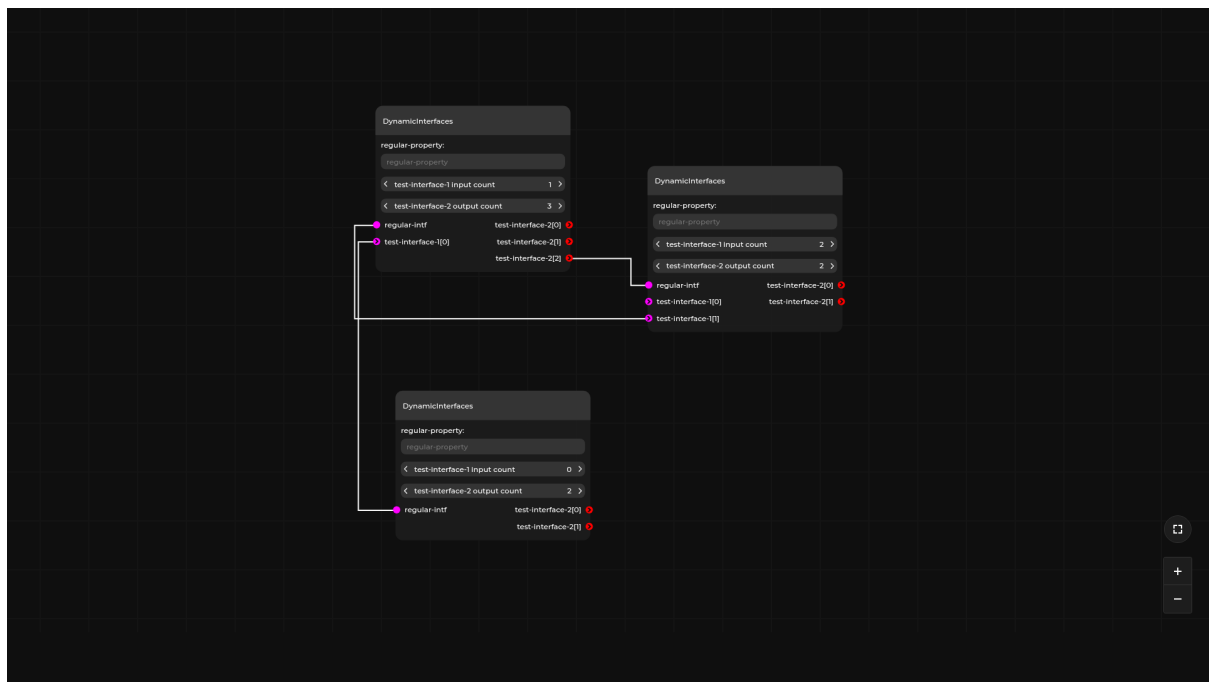
## DEMONSTRATION OF THE FRONTEND

To see the work of the frontend check one of the below examples:

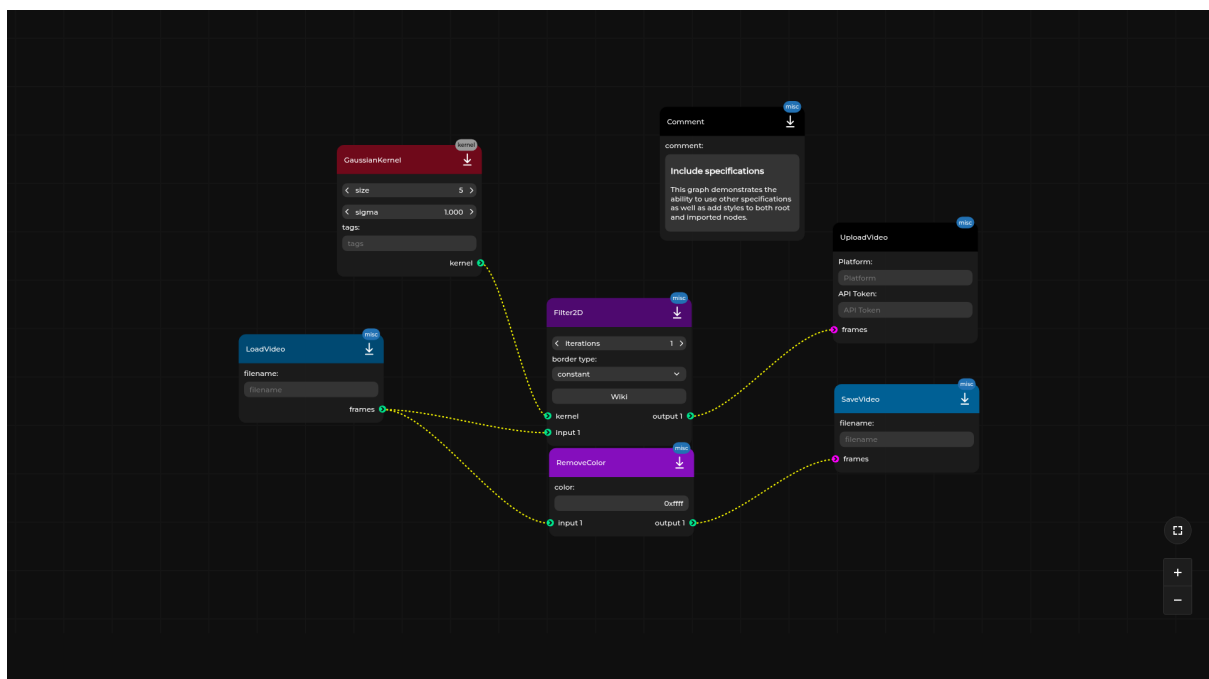
### 11.1 Example of a more complex graph with nodes as interfaces, inheritance and more



## 11.2 Example of a graph with dynamic number of interfaces



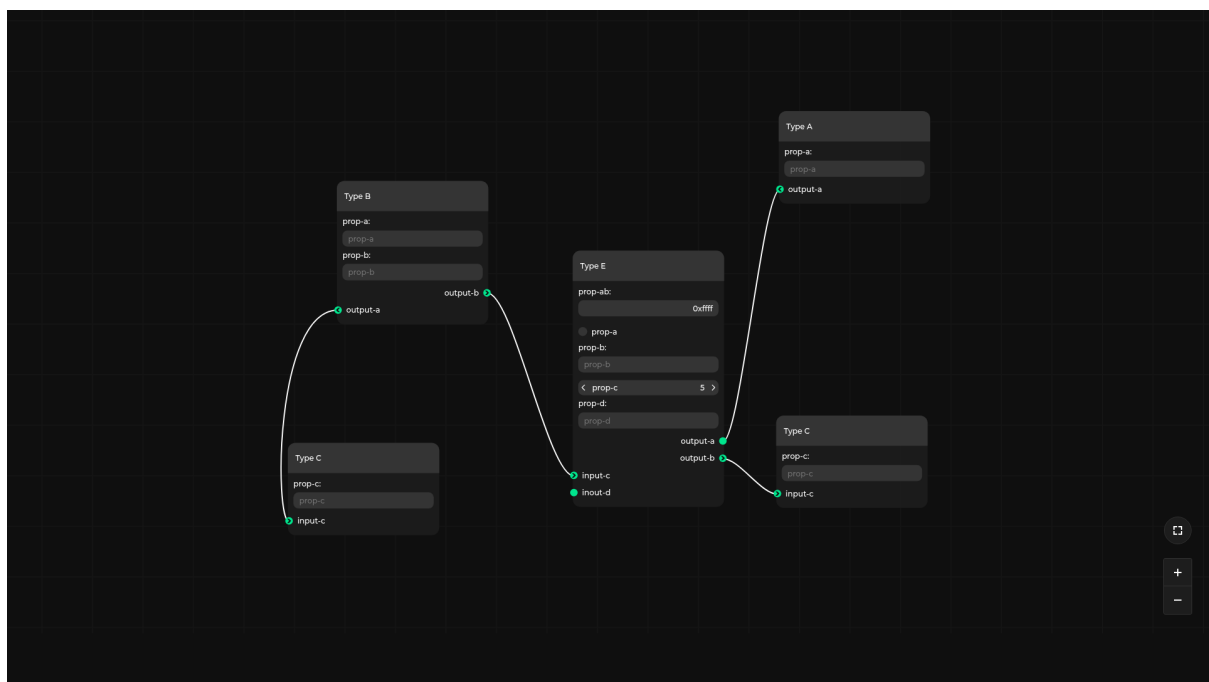
## 11.3 Example of the graph with include mechanisms



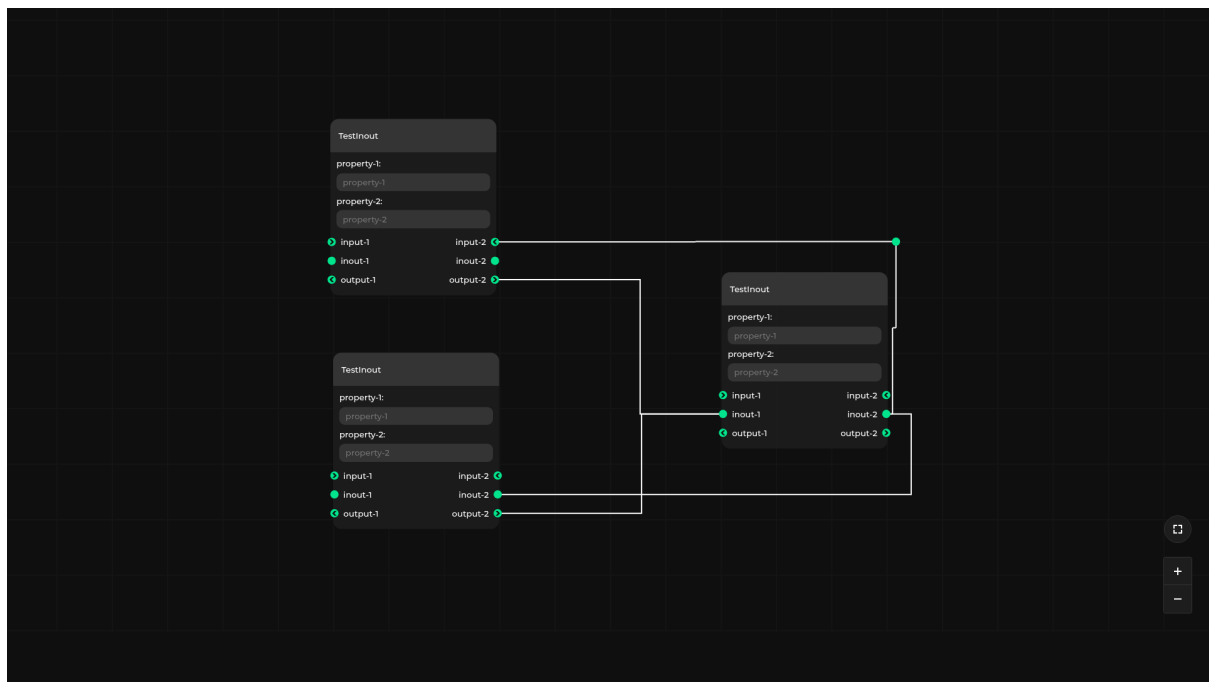
## 11.4 Example of the graph with top-level interfaces exposed



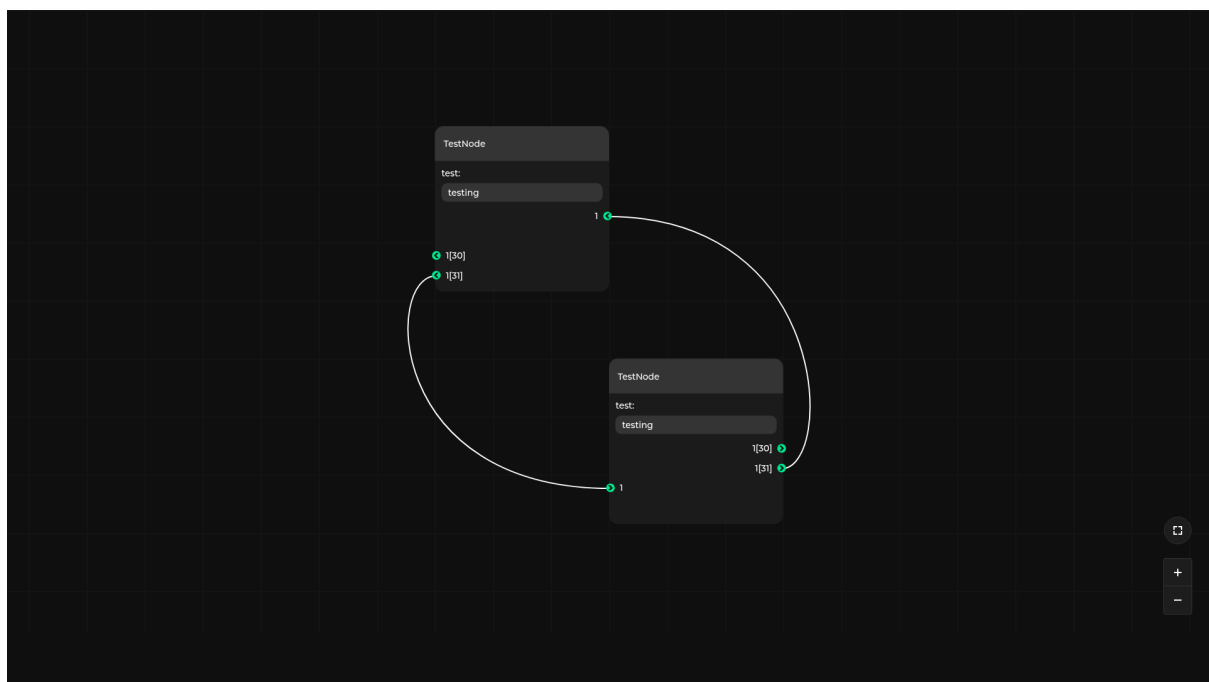
## 11.5 Example of graph with inheritance



## 11.6 Example of Input-Output interfaces

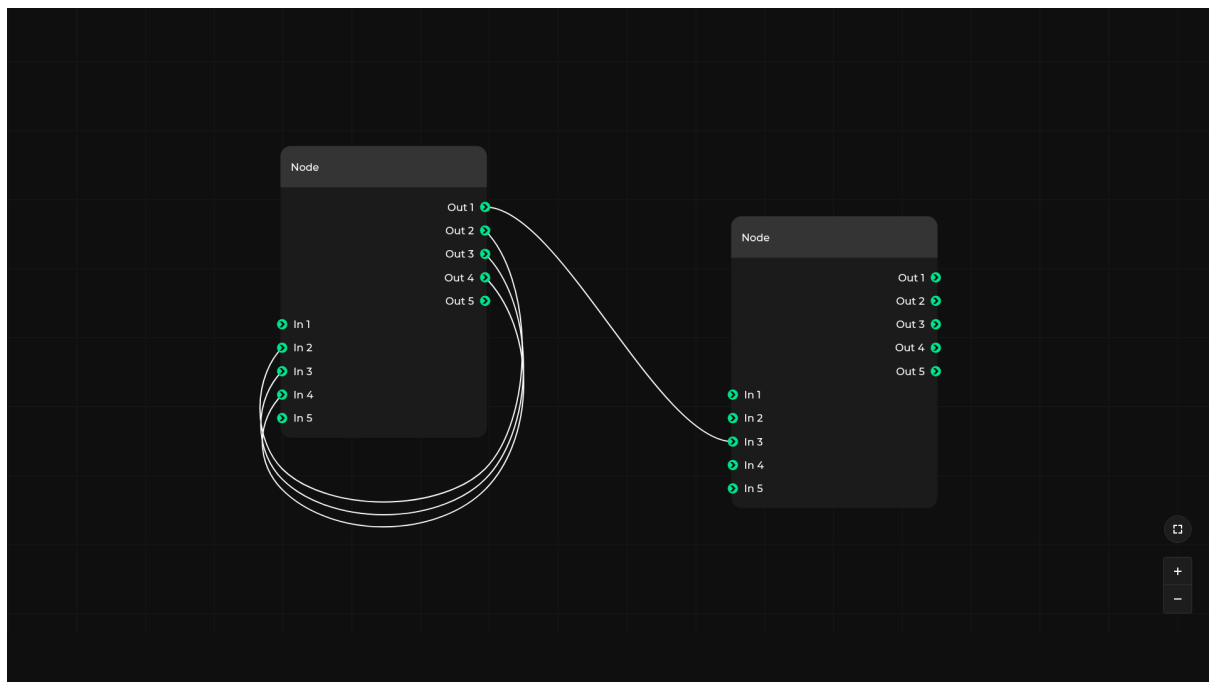


## 11.7 Example of interface grouping

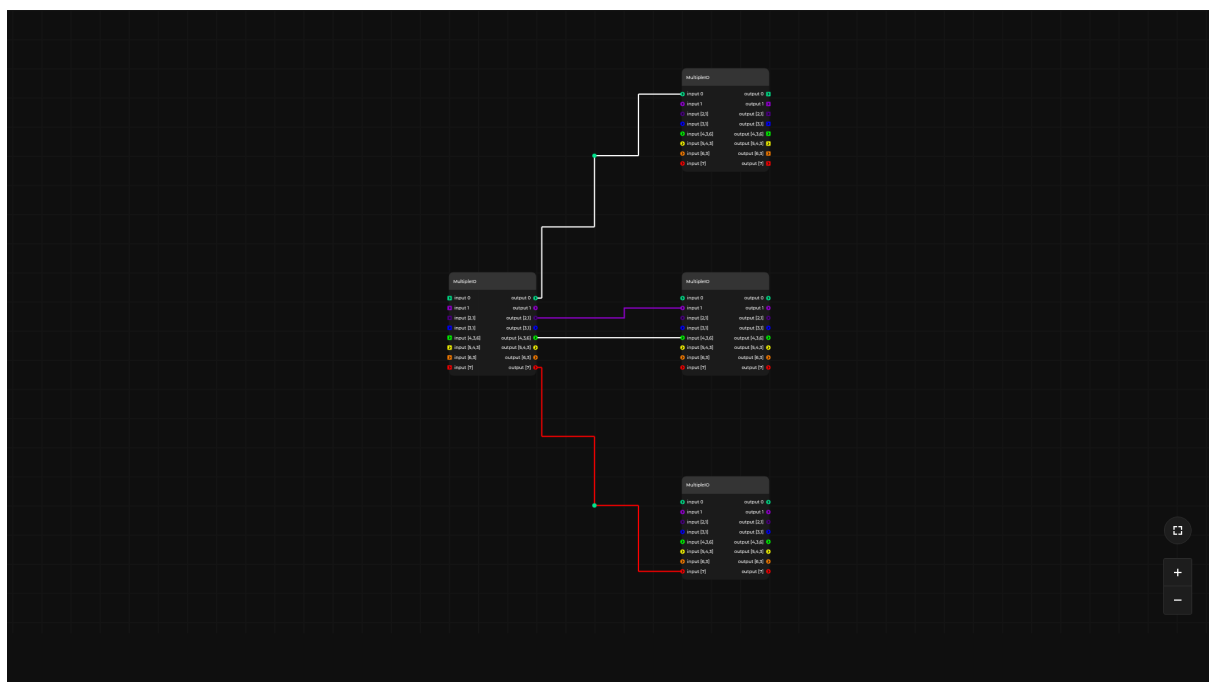




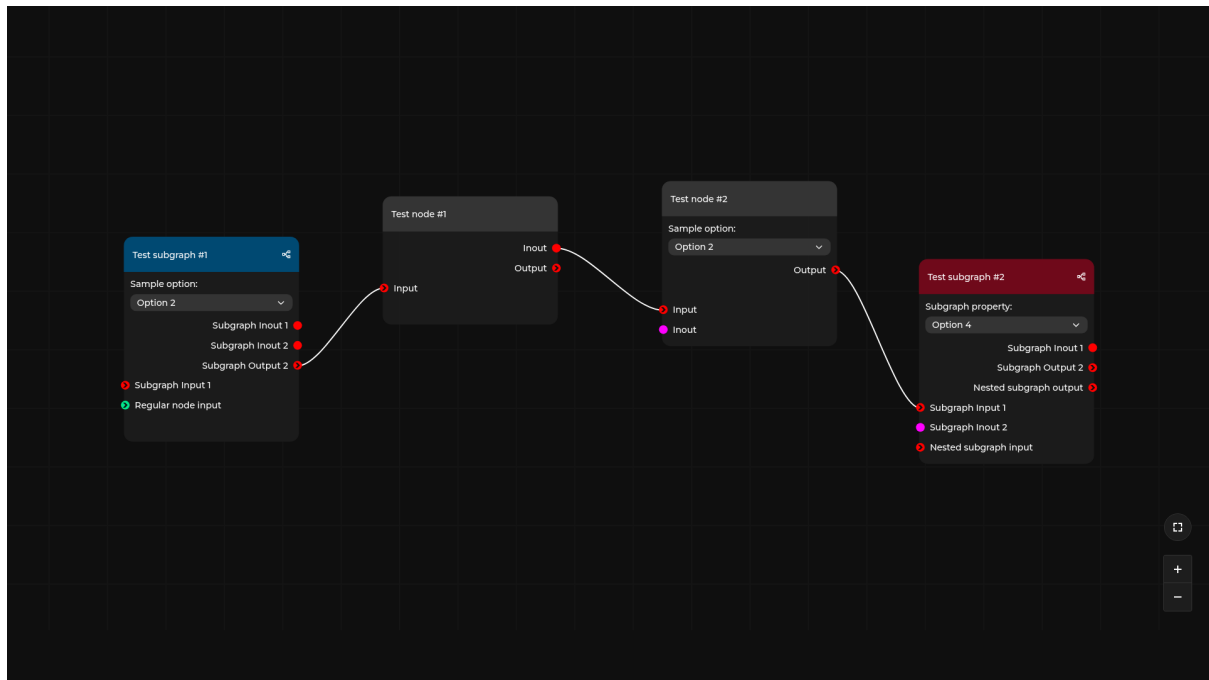
## 11.8 Example of graphs with connection loopbacks



## 11.9 Example of Multiple Input-Output node



## 11.10 Example of a graph with graph nodes



## 11.11 Communication with Pipeline Manager via postMessage

The demo allows to check communication between Pipeline Manager and its embedding web-page.

Go to [Frontend-to-Frontend demo](#).

## INDEX

### A

`add_include()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 75  
`add_include_subgraph()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 75  
`add_interface_to_group_interface()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 75  
`add_node_description()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 76  
`add_node_interface_group()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 76  
`add_node_type()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 77  
`add_node_type_additional_data()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 77  
`add_node_type_as_category()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 77  
`add_node_type_category()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 78  
`add_node_type_color()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 78  
`add_node_type_from_spec()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 78  
`add_node_type_icon()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 78  
`add_node_type_interface()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 78  
`add_node_type_parent()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 79  
`add_node_type_property()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 79  
`add_node_type_property_group()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 80  
`add_node_type_style()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 81  
`add_node_type_subgraph_id()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 81  
`add_node_type_url()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 81  
`add_subgraph_from_spec()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 81  
`create_and_validate_spec()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 82  
`create_connection()` (`pipeline_manager.dataflow_builder.dataflow_graph.DataflowGraph`  
`method`), 93  
`create_graph()` (`pipeline_manager.dataflow_builder.dataflow_graph.DataflowGraph`  
`method`), 90  
`create_node()` (`pipeline_manager.dataflow_builder.dataflow_graph.DataflowGraph`  
`method`), 94  
`create_property()` (`pipeline_manager.specification_builder.SpecificationBuilder`  
`method`), 82  
`create_subgraph_node()` (`pipeline_manager.dataflow_builder.dataflow_graph.DataflowGraph`  
`method`), 94  
`DataflowGraph` (class in `pipeline_manager.dataflow_builder.dataflow_graph`),

93

99

## E

enable\_interface\_group\_by\_default()  
(pipeline\_manager.specification\_builder.SpecificationBuilder  
method), 83

## G

get() (pipeline\_manager.dataflow\_builder.dataflow\_graph.DataflowGraph  
method), 95

get() (pipeline\_manager.dataflow\_builder.entities.Node  
method), 96

get\_by\_id() (pipeline\_manager.dataflow\_builder.dataflow\_graph.DataflowGraph  
method), 95

get\_graph\_by\_id()  
(pipeline\_manager.dataflow\_builder.dataflow\_graph.DataflowGraph  
method), 91

get\_graph\_by\_property()  
(pipeline\_manager.dataflow\_builder.dataflow\_graph.DataflowGraph  
method), 91

get\_interfaces\_by\_regex()  
(pipeline\_manager.dataflow\_builder.entities.Node  
method), 96

get\_node\_description()  
(pipeline\_manager.specification\_builder.SpecificationBuilder  
method), 83

get\_number\_of\_dynamic\_interfaces()  
(pipeline\_manager.dataflow\_builder.entities.Node  
method), 96

get\_property() (pipeline\_manager.dataflow\_builder.entities.Node  
method), 97

get\_subgraph\_by\_name()  
(pipeline\_manager.dataflow\_builder.dataflow\_graph.DataflowGraph  
method), 91

get\_subgraphs()  
(pipeline\_manager.dataflow\_builder.dataflow\_graph.DataflowGraph  
method), 92

GraphBuilder (class in Property (class in  
pipeline\_manager.dataflow\_builder.dataflow\_graph.DataflowGraph),  
90 100

## I

id (pipeline\_manager.dataflow\_builder.dataflow\_graph.DataflowGraph  
property), 95

init\_subgraph\_node()  
(pipeline\_manager.dataflow\_builder.entities.Node  
static method), 97

Interface (class in  
pipeline\_manager.dataflow\_builder.entities), 99

InterfaceConnection (class in  
pipeline\_manager.dataflow\_builder.entities),

## L

load\_graphs() (pipeline\_manager.dataflow\_builder.dataflow\_graph.DataflowGraph  
method), 92

load\_specification()  
(pipeline\_manager.dataflow\_builder.dataflow\_graph.DataflowGraph  
method), 92

## M

metadata\_add\_interface\_styling()  
(pipeline\_manager.specification\_builder.SpecificationBuilder  
method), 84

metadata\_add\_layer()  
(pipeline\_manager.specification\_builder.SpecificationBuilder  
method), 84

metadata\_add\_node\_style()  
(pipeline\_manager.specification\_builder.SpecificationBuilder  
method), 84

metadata\_add\_param()  
(pipeline\_manager.specification\_builder.SpecificationBuilder  
method), 84

metadata\_add\_url()  
(pipeline\_manager.specification\_builder.SpecificationBuilder  
method), 85

move() (pipeline\_manager.dataflow\_builder.entities.Node  
method), 97

## N

name (pipeline\_manager.dataflow\_builder.entities.Node  
property), 98

Node (class in pipeline\_manager.dataflow\_builder.entities),  
91

## P

pipeline\_manager.dataflow\_builder.dataflow\_graph.DataflowGraph  
property), 95

pipeline\_manager.dataflow\_builder.entities,  
100

## R

render() (pipeline\_manager.specification\_builder.SpecificationBuilder  
method), 85

render\_node\_type\_interface()  
(pipeline\_manager.specification\_builder.SpecificationBuilder  
method), 85

reset() (pipeline\_manager.specification\_builder.SpecificationBuilder  
method), 86

## S

`save()` (`pipeline_manager.dataflow_builder.dataflow_builder.GraphBuilder`  
method), 92

`scaling` (`pipeline_manager.dataflow_builder.dataflow_graph.DataflowGraph`  
property), 95

`set_node_description()`  
(`pipeline_manager.specification_builder.SpecificationBuilder`  
method), 86

`set_number_of_dynamic_interfaces()`  
(`pipeline_manager.dataflow_builder.entities.Node`  
method), 98

`set_property()` (`pipeline_manager.dataflow_builder.entities.Node`  
method), 98

`SpecificationBuilder` (class in  
`pipeline_manager.specification_builder`),  
74

## T

`to_json()` (`pipeline_manager.dataflow_builder.dataflow_graph.DataflowGraph`  
method), 95

`to_json()` (`pipeline_manager.dataflow_builder.entities.Interface`  
method), 99

`to_json()` (`pipeline_manager.dataflow_builder.entities.InterfaceConnection`  
method), 100

`to_json()` (`pipeline_manager.dataflow_builder.entities.Node`  
method), 99

`to_json()` (`pipeline_manager.dataflow_builder.entities.Property`  
method), 100

## U

`update_spec_from_other()`  
(`pipeline_manager.specification_builder.SpecificationBuilder`  
method), 86

## V

`validate()` (`pipeline_manager.dataflow_builder.dataflow_builder.GraphBuilder`  
method), 93