Creating an Application

Table of contents

List of Figures

In this section, we'll address:

- how to define an Application class
- how to configure an Application
- how to define different types of workflows
- how to build and run your application

Note

This section covers basics of applications running as a single fragment. For multi-fragment applications, refer to the distributed application documentation.

Defining an Application Class

The following code snippet shows an example Application code skeleton:

Ingested Tab Module

Tip

This is also illustrated in the hello_world example.

It is also possible to instead launch the application asynchronously (i.e. non-blocking for the thread launching the application), as shown below:

Ingested Tab Module

Tip

This is also illustrated in the ping_simple_run_async example.

Configuring an Application

An application can be configured at different levels:

1. providing the GXF extensions that need to be loaded (when using GXF operators)
2. configuring parameters for your application, including for:
    1. the operators in the workflow
    2. the scheduler of your application
3. configuring some runtime properties when deploying for production

The sections below will describe how to configure each of them, starting with a native support for YAML-based configuration for convenience.

YAML Configuration support

Holoscan supports loading arbitrary parameters from a YAML configuration file at runtime, making it convenient to configure each item listed above, or other custom parameters you wish to add on top of the existing API. For C++ applications, it also provides the ability to change the behavior of your application without needing to recompile it.

Note

Usage of the YAML utility is optional. Configurations can be hardcoded in your program, or done using any parser of your choosing.

Here is an example YAML configuration:

```
string_param: "test" float_param: 0.50 bool_param: true dict_param: key_1: value_1 key_2: value_2
```

Ingesting these parameters can be done using the two methods below:

Ingested Tab Module

Tip

This is also illustrated in the video_replayer example.

⚠ **Attention**

With both `from_config` and `kwargs`, the returned `ArgList` /dictionary will include both the key and its associated item if that item value is a scalar. If the item is a map/dictionary itself, the input key is dropped, and the output will only hold the key/values from that item.

Loading GXF extensions

If you use operators that depend on GXF extensions for their implementations (known as GXF operators), the shared libraries (`.so`) of these extensions need to be dynamically loaded as plugins at runtime.

The SDK already automatically handles loading the required extensions for the built-in operators in both C++ and Python, as well as common extensions (listed here). To load additional extensions for your own operators, you can use one of the following approach:

Ingested Tab Module

> **Note**
>
> To be discoverable, paths to these shared libraries need to either be absolute, relative to your working directory, installed in the `lib/gxf_extensions` folder of the holoscan package, or listed under the `HOLOSCAN_LIB_PATH` or `LD_LIBRARY_PATH` environment variables.

Configuring operators

Operators are defined in the `compose()` method of your application. They are not instantiated (with the `initialize` method) until an application's `run()` method is called.

Operators have three type of fields which can be configured: parameters, conditions, and resources.

Configuring operator parameters

Operators could have parameters defined in their `setup` method to better control their behavior (see details when creating your own operators). The snippet below would be the implementation of this method for a minimal operator named `MyOp`, that takes a string and a boolean as parameters; we'll ignore any extra details for the sake of this example:

Ingested Tab Module

> **Tip**
>
> Given an instance of an operator class, you can print a human-readable description of its specification to inspect the parameter fields that can be configured on that operator class:
>
> Ingested Tab Module

Given this YAML configuration:

> myop_param: string_param: "test" bool_param: true bool_param: false # *we'll use this later*

We can configure an instance of the `MyOp` operator in the application's `compose` method like this:

Ingested Tab Module

Tip

This is also illustrated in the ping_custom_op example.

If multiple `ArgList` are provided with duplicate keys, the latest one overrides them:

Ingested Tab Module

Configuring operator conditions

By default, operators with no input ports will continuously run, while operators with input ports will run as long as they receive inputs (as they're configured with the `MessageAvailableCondition`).

To change that behavior, one or more other conditions classes can be passed to the constructor of an operator to define when it should execute.

For example, we set three conditions on this operator `my_op`:

Ingested Tab Module

Tip

This is also illustrated in the conditions examples.

ⓘ **Note**

You'll need to specify a unique name for the conditions if there are multiple conditions applied to an operator.

Configuring operator resources

Some resources can be passed to the operator's constructor, typically an allocator passed as a regular parameter.

For example:

Ingested Tab Module

Configuring the scheduler

The scheduler controls how the application schedules the execution of the operators that make up its workflow.

The default scheduler is a single-threaded `GreedyScheduler`. An application can be configured to use a different scheduler `Scheduler` (`C++` / `Python`) or change the parameters from the default scheduler, using the `scheduler()` function (`C++` / `Python`).

For example, if an application needs to run multiple operators in parallel, the `MultiThreadScheduler` or `EventBasedScheduler` can instead be used. The difference between the two is that the MultiThreadScheduler is based on actively polling operators to determine if they are ready to execute, while the EventBasedScheduler will instead wait for an event indicating that an operator is ready to execute.

The code snippet belows shows how to set and configure a non-default scheduler:

Ingested Tab Module

Tip

This is also illustrated in the multithread example.

Configuring runtime properties

As described below, applications can run simply by executing the C++ or Python application manually on a given node, or by packaging it in a HAP container. With the latter, runtime properties need to be configured: refer to the App Runner Configuration for details.

Application Workflows

Note

Operators are initialized according to the topological order of its fragment-graph. When an application runs, the operators are executed in the same topological order. Topological ordering of the graph ensures that all the data dependencies of an operator are satisfied before its instantiation and execution. Currently, we do not support specifying a different and explicit instantiation and execution order of the operators.

One-operator Workflow

The simplest form of a workflow would be a single operator.

Fig. 12 *A one-operator workflow*

The graph above shows an **Operator** (C++ / Python) (named MyOp) that has neither inputs nor output ports.

- Such an operator may accept input data from the outside (e.g., from a file) and produce output data (e.g., to a file) so that it acts as both the source and the sink operator.
- Arguments to the operator (e.g., input/output file paths) can be passed as parameters as described in the section above.

We can add an operator to the workflow by calling add_operator (C++ / Python) method in the compose() method.

The following code shows how to define a one-operator workflow in compose() method of the App class (assuming that the operator class MyOp is declared/defined in the same file).

Ingested Tab Module

Linear Workflow

Here is an example workflow where the operators are connected linearly:

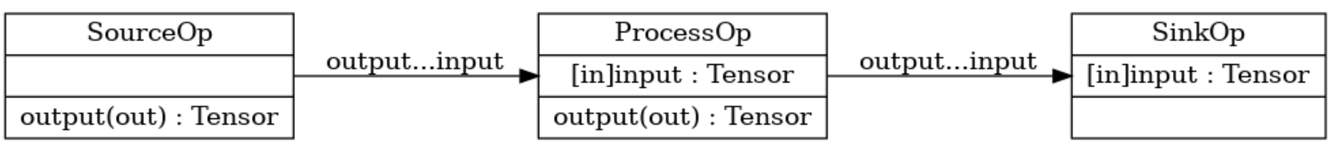


Fig. 13 *A linear workflow*

In this example, **SourceOp** produces a message and passes it to **ProcessOp**. **ProcessOp** produces another message and passes it to **SinkOp**.

We can connect two operators by calling the add_flow() method (C++ / Python) in the compose() method.

The add_flow() method (C++ / Python) takes the source operator, the destination operator, and the optional port name pairs. The port name pair is used to connect the output port of the source operator to the input port of the destination operator. The first element of the pair is the output port name of the upstream operator and the second element is the input port name of the downstream operator. An empty port name (“”) can be used for specifying a port name if the operator has only one input/output port. If there is only one output port in the upstream operator and only one input port in the downstream operator, the port pairs can be omitted.

The following code shows how to define a linear workflow in the `compose()` method of the `App` class (assuming that the operator classes `SourceOp`, `ProcessOp`, and `SinkOp` are declared/defined in the same file).

Ingested Tab Module

Complex Workflow (Multiple Inputs and Outputs)

You can design a complex workflow like below where some operators have multi-inputs and/or multi-outputs:

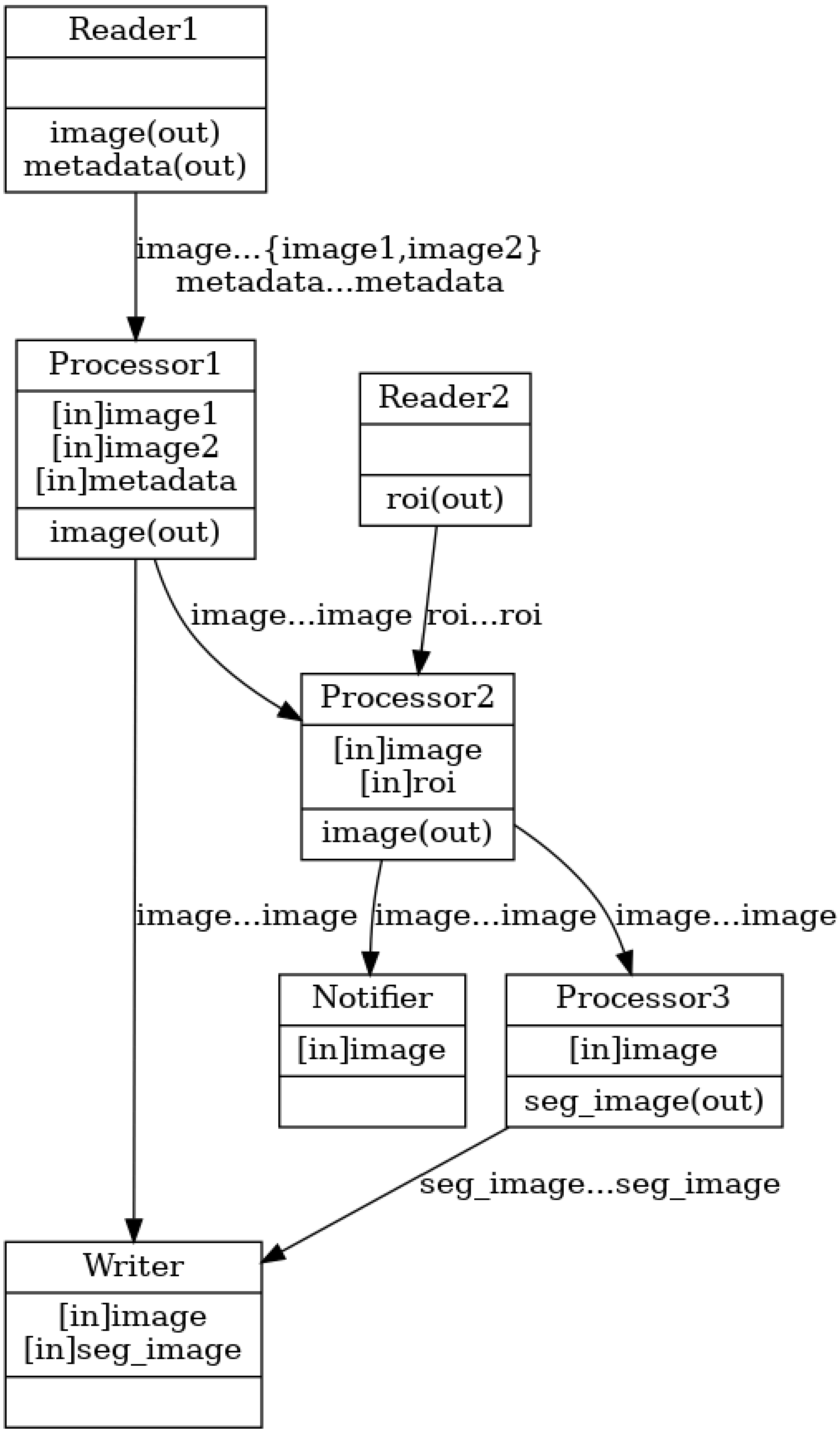

Reader1
image(out)
metadata(out)
image...{image1,image2}
metadata...metadata
Processor1
[in]image1
[in]image2
[in]metadata
image(out)
Reader2
roi(out)
image...image
roi...roi
Processor2
[in]image
[in]roi
image(out)
image...image
image...image
image...image
Notifier
[in]image
Processor3
[in]image
seg_image(out)
seg_image...seg_image
Writer
[in]image
[in]seg_image

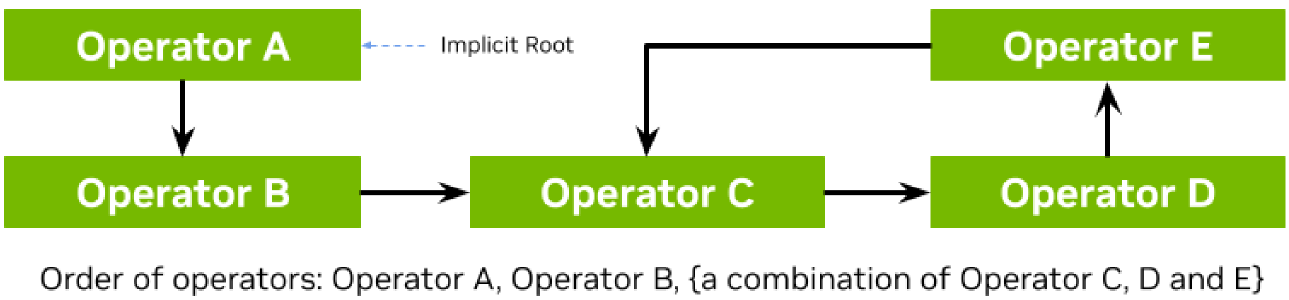
Fig. 14 *A complex workflow (multiple inputs and outputs)*

Ingested Tab Module

If there is a cycle in the graph with no implicit root operator, the root operator is either the first operator in the first call to `add_flow` method (`C++` / `Python`), or the operator in the first call to `add_operator` method (`C++` / `Python`).

Ingested Tab Module

If there is a cycle in the graph with an implicit root operator which has no input port, then the initialization and execution orders of the operators are still topologically sorted as far as possible until the cycle needs to be explicitly broken. An example is given below:



Building and running your Application

Ingested Tab Module

> **Note**
>
> Given a CMake project, a pre-built executable, or a python application, you can also use the Holoscan CLI to package and run your Holoscan application in a OCI-compliant container image.