DXNN Streamer (DX-STREAM) User Manual

v2.0.0

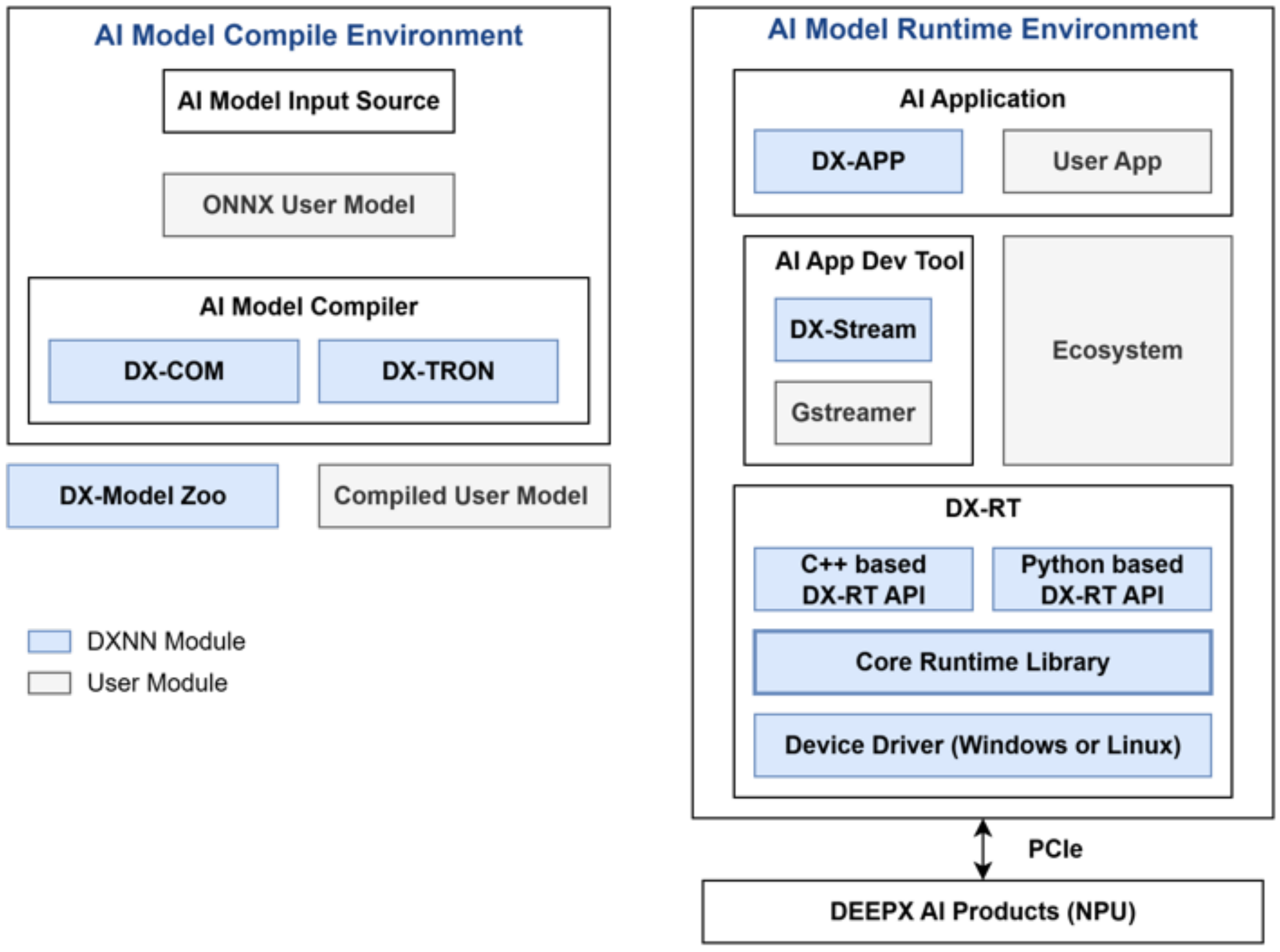
DEEPX.ai

Table of contents

1. DXNN SDK Overview

This chapter provides an overview of the DEEPX SDK architecture and explains each core component and its role in the AI development workflow.

1.1 DEEPX SDK Architecture



DEEPX SDK is an all-in-one software development platform that streamlines the process of compiling, optimizing, simulating, and deploying AI inference applications on DEEPX NPUs (Neural Processing Units). It provides a complete toolchain, from AI model creation to runtime deployment, optimized for edge and embedded systems, enabling developers to build high-performance AI applications with minimal effort.

DX-COM is the compiler in the DEEPX SDK that converts a pre-trained ONNX model and its associated configuration JSON file into a hardware-optimized .dxnn binary for DEEPX NPUs. The ONNX file contains the model structure and weights, while the JSON file defines pre/post-processing settings and compilation parameters. DX-COM provides a fully compiled .dxnn file, optimized for low-latency and high-efficient inference on DEEPX NPU.

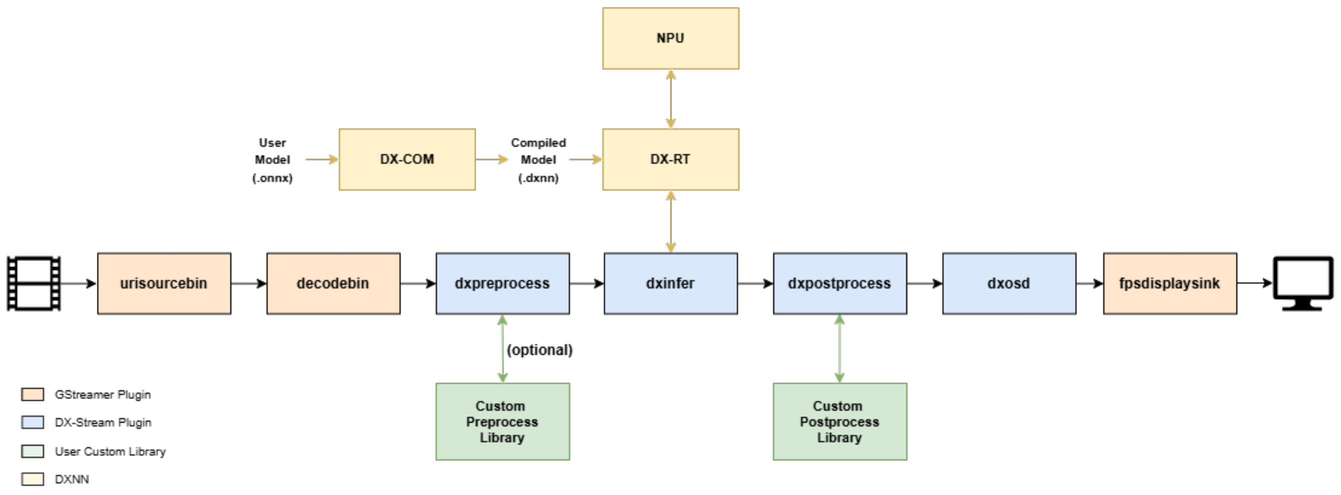
DX-RT is the runtime software responsible for executing ,dxnn models on DEEPX NPU hardware. DX-RT directly interacts with the DEEPX NPU through firmware and device drivers, using PCIe interface for high-speed data transfer between the host and the NPU, and provides C/C++ and Python APIs for application-level inference control. DX-RT offers a complete runtime environment, including model loading, I/O buffer management, inference execution, and real-time hardware monitoring.

DX ModelZoo is a curated collection of pre-trained neural network models optimized for DEEPX NPU, designed to simplify AI development for DEEPX users. It includes pre-trained ONNX models, configuration JSON files, and pre-compiled DXNN binaries, allowing developers to rapidly test and deploy applications. DX ModelZoo also provides benchmark tools for comparing the performance of quantized INT8 models on DEEPX NPU with full-precision FP32 models on CPU or GPU.

DX-STREAM is a custom GStreamer plugin that enables real-time streaming data integration into AI inference applications on DEEPX NPU. It provides a modular pipeline framework with configurable elements for preprocessing, inference, and postprocessing, tailored to vision AI work. DX-Stream allows developers to build flexible, high-performance applications for use cases such as video analytics, smart cameras, and edge AI systems.

DX-APP is a sample application that demonstrates how to run compiled models on actual DEEPX NPU using DX-RT. It includes ready-to-use code for common vision tasks such as object detection, face recognition, and image classification. DX-APP helps developers quickly set up the runtime environment and serves as a template for building and customizing their own AI applications.

1.2 DX-STREAM Architecture



DX-STREAM provides dedicated Gstreamer elements for AI model inference on DEEPX NPU and includes auxiliary elements for developing AI applications.
For most AI model processing, you can quickly build an inference environment by setting the properties of the elements provided by DX-Stream. In addition, we support processing through user-defined libraries in pre/post-processing for the user's unique model inference.
This plugin-based architecture accelerates development and deployment while providing the flexibility needed for a wide range of vision AI applications, including object detection, multi-stream analysis, and real-time tracking.

1.3 DX-STREAM Key Features

The following features highlight what makes DX-Stream a powerful and flexible framework for Vision AI application development.

Pipeline Modularity

DX-Stream divides the Vision AI data processing flow into discrete, functional units called Elements. Each element performs a specific task within a GStreamer pipeline, enabling modular, scalable, and reusable pipeline design.

Elements

- DxPreprocess – Performs input preprocessing for AI inference.
- DxInfer – Executes AI model inference using the DEEPX NPU.
- DxPostprocess – Processes and formats inference output.
- DxTracker – Tracks objects persistently across video frames.
- DxOsd – Overlays inference results onto video frames.
- DxGather – Merges multiple branched streams into a single unified stream.
- DxInputSelector - Selects one stream among multiple inputs based on PTS and forwards it to a single output.
- DxOutputSelector - Distributes a single input stream to multiple outputs based on predefined routing.
- DxRate – Controls the output frame rate of the stream.
- DxMsgConv – Converts inference results into structured messages.
- DxMsgBroker – Sends messages to external systems or networks.

Inference Workflow

The core inference process in DX-Stream follows this three-stage structure.
- `DxPreprocess → DxInfer → DxPostprocess`

This three-stage pipeline enables seamless integration of AI model inference into multimedia streams while maintaining flexibility for diverse model architectures.

Custom Library Support

The **DxPreprocess** and **DxPostprocess** elements support user-defined custom libraries, allowing developers to integrate application-specific preprocessing and postprocessing logic into the pipeline. This makes it easy to adapt DX-Stream to diverse AI models and deployment requirements.

Efficient AI Inference

The **DxInfer** element executes AI models using the DEEPX NPU, ensuring high-throughput and low-latency performance optimized for edge AI applications.
Inference requires a model in `.dxnn` format, which is generated from an ONNX file using DX-COM.

1.4 DX-STREAM Model Workflow

The model workflow in **DX-STREAM** consists of three key stages, from model preparation to real-time inference on DEEPX NPU hardware.

1. Model Compilation
DX-COM (Compiler) converts `.onnx` model files into .dxnn format, optimized for execution on DEEPX NPU hardware.

2. Model Execution
The compiled `.dxnn` file is loaded onto the NPU.
The host system uses **DX-RT** (Runtime) to

- Allocate computation tasks to the NPU
- Provide input tensors to the inference engine
- Receive output tensors (inference results) from the NPU

3. Inference Process
All deep learning computations are offloaded to the NPU, ensuring high-performance and low-latency inference execution optimized for edge environments.

2. DX-STREAM Installation

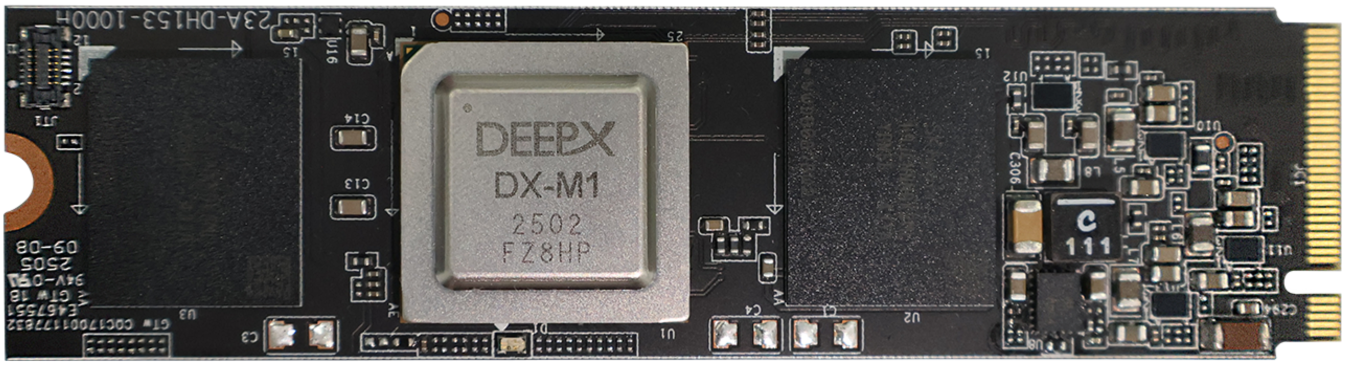
This chapter describes the installation of **DX-STREAM** in both source-based and Docker-based environments.

2.1 System Requirements

This section describes the hardware and software requirements for running **DX-STREAM**.

Hardware and Software Requirements

- **CPU:** amd64(x86_64), aarch64(arm64)
- **RAM:** 8GB RAM (16GB RAM or higher is recommended)
- **Storage:** 4GB or higher available disk space
- **OS:** Ubuntu 18.04 / 20.04 / 22.04 / 24.04 (x64)
- **DX-RT must** be installed 3.0.0 or higher available
- The system **must** support connection to an **M1 M.2** module with the M.2 interface on the host PC.



Note. The **NPU Device Driver** and **DX-RT must** be installed. Refer to **DX-RT User Manual** for detailed instructions on installing NPU Device Driver and DX-RT.

2.2 Install DX-STREAM

DX-STREAM can be installed either by building from source or by using a pre-built Docker image (please refer to DX-AS Docs). This section explains both approaches.

2.2.1 Build from Sources

1. Unzip the Package

Extract the **DX-STREAM** source files.

```
$ unzip dxstream_vX.X.X.zip -d dx_stream
$ cd dx_stream
```

2. Install Dependencies

Run the provided script to automatically install all required packages.

```
$ ./install.sh
```

3. Build **DX-STREAM**

Compile **DX-STREAM**.

```
$ ./build.sh
```

(Optional) Build with debug symbols.

```
$ ./build.sh --debug
```

4. Verify the installation

Check that the plugin is correctly installed.

```
$ gst-inspect-1.0 dxstream
```

Note. If you want to remove **DX-STREAM**, use the following command.

```
$ ./build.sh --uninstall
```

2.3 Run DX-STREAM

This section provides a step-by-step guide of quickly running **DX-STREAM**'s sample pipelines

Requirements

Before you start, ensure the following prerequisites are met.

- Properly install **DX-RT**, **the NPU Device Driver**, and **DX-STREAM** in the correct order.
- Download the sample video and model needed to run the demo.

```
$ cd dx_stream
$ ./setup.sh
```

By running the above command, you can download the resources needed for the demo.

Run Demo Pipelines

Execute the demo script.

```
$ cd dx_stream
$ ./run_demo.sh
```

When the script is executed, you'll be prompted to select a demo from the following options.

```
0: Object Detection (YOLOv7)
1: Face Detection (YOLOV5S_Face)
2: Multi-Object Tracking
3: Pose Estimation
4: Semantic Segmentation
5: Multi-Channel Object Detection
6: Multi-Channel RTSP
7: secondary mode
which AI demo do you want to run:(timeout:10s, default:0)
```

Enter the number corresponding to the desired demo to run it.

If **no** input is provided within 10 seconds, the default option (`0: Object Detection`) will be executed automatically.

Notes. Each demo corresponds to a specific pipeline described in **Chapter 5. Pipeline Examples**.

3. Elements

3.1 DxPreprocess

DxPreprocess is an element that performs input preprocessing on raw video frames, converting them into a format suitable for AI models used by **DxInfer**. Each preprocessed input tensor is assigned an input ID specified by the `preprocess-id` property.

DxInfer must reference this ID in its own configuration to determine which tensor to use for inference.

Multiple **DxPreprocess** elements can co-exist in the same pipeline, enabling support for multi-input models or customized preprocessing steps for different video streams.

Key Features

Color Conversion

DxPreprocess converts the image color format to `RGB` or `BGR` based on the `color-format` property.

- Default format: `RGB`.

Modes of Operation

- **Primary Mode** applies preprocessing to the entire frame. If object detection is performed within the same pipeline, **DxPreprocess** will operate in this mode.
- **Secondary Mode** applies preprocessing to individual object regions detected in this frame. This mode requires object metadata (e.g., from an upstream object detection element).

Region of Interest (ROI)

The Region of Interest (ROI) is defined using the roi property.

- In **Primary Mode**, the specified ROI area is cropped and preprocessed as a whole.
- In **Secondary Mode**, only objects that are fully contained within the ROI are selected for preprocessing.

Processing Interval

The processing interval is controlled by the `interval` property. It skips a specified number of frames before preprocessing the next frame or object. This is useful for reducing processing frequency in resource-constrained environments.

Object Filtering in Secondary Mode
Objects can be filtered based on the following criteria.

- **Class ID**: Use the `class-id` property to process only objects that match the specified class. Non-matching objects are ignored.
- **Size** : Use the `min-object-width` and `min-object-height` properties to exclude objects smaller than the defined size.

Resizing
DxPreprocess resizes full frames or object regions using the `resize-width` and `resize-height` property.
If `keep-ratio` is set to `true`, the aspect ratio is preserved by applying padding.
Padding color is set using the `pad-value` property.

Custom Preprocessing
User-defined preprocessing logic can be implemented by providing.

- `library-file-path` : Path to the custom shared library (`.so`).
- `function-name` : Name of the preprocessing function within the library.

This allows implementation of customized data handling tailored to specific AI models.

QoS Handling
If the downstream sink element has `sync=true`, input buffers may be dropped based on their timestamps. This helps maintain real-time performance and avoids frame backlog under system load.

H/W Acceleration
If the Rockchip RGA (Raster Graphic Accelerator) module is available in the system environment, the input preprocessing step is offloaded from the CPU to the RGA, enabling hardware-accelerated processing. As a result, CPU usage is reduced and processing becomes more efficient.

Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GstObject
             +----GstElement
                   +----GstBaseTransform
                         +----GstDxPreprocess
```

Properties

This table provides a complete reference to the properties of the **DxPreprocess** element.

Name	Description	Type	Default Value
`name`	Sets the unique name of the DxPreprocess element.	String	`"dxpreprocess0"`
`config-file-path`	Path to the JSON config file containing the element's properties.	String	`null`
`preprocess-id`	Assigns an ID to the preprocessed input tensor.	Integer	`0`
`resize-width`	Specifies the width for resizing.	Integer	`0`
`resize-height`	Specifies the height for resizing.	Integer	`0`
`keep-ratio`	Maintains the original aspect ratio during resizing.	Boolean	`false`
`pad-value`	Padding color value for R, G, B pixels during	Integer	`0`
`color-format`	Specifies the color format for preprocessing.	Enum (`rgb` , `bgr`)	`0` (`rgb`)
`secondary-mode`	Enables Secondary Mode for processing object regions.	Boolean	`false`
`target-class-id`	Filters objects in Secondary Mode by class ID. (`-1` processes all objects).	Integer	`-1`

Name	Description	Type	Default Value
`min-object-width`	Minimum object width for preprocessing in Secondary Mode.	Integer	`0`
`min-object-height`	Minimum object height for preprocessing in Secondary Mode.	Integer	`0`
`roi`	Defines the ROI (Region of Interest) for preprocessing.	Array of Integers	`[-1, -1, -1, -1]`
`interval`	Specifies the interval for preprocessing frames or objects.	Integer	`0`
`library-file-path`	Path to the custom preprocess library, if used.	String	`null`
`function-name`	Name of the custom preprocessing function to use.	String	`null`

Example JSON Configuration

```
{
    "preprocess_id": 1,
    "resize_width": 640,
    "resize_height": 640,
    "keep_ratio": true
}
```

Notes

- For implementing custom preprocess logic, refer to **Chapter 4. Writing Your Own Application** `“Custom Pre-Process Library Documentation”`.
- All properties can also be configured using a JSON file for enhanced usability and flexibility.

3.2 DxInfer

DxInfer is an element that performs AI model inference using the **DEEPX** NPU. It processes input tensors received from **DxPreprocess** elements and produces output tensors for downstream processing.

- **Input tensors: DxInfer** receives preprocessed input tensors from **DxPreprocess** and performs inference using a specified AI model.
- **Output tensors:** Each output tensor is assigned an ID using the `inference-id` property, allowing downstream elements such as **DxPreprocess** to retrieve the correct output.
- **Model configuration:** The AI model used for inference must be specified using the `model-path` property, which points to a compiled `.dxnn` file.

Key Features

Input Tensor Management
Input tensors are linked to **DxInfer** using the `preprocess-id` property. This ensures that the correct tensor from **DXPreprocess** is used for inference.

Output Tensor Management
Each output tensor is uniquely defined in the `inference-id` property. This allows downstream elements like **DXPostprocess** to connect to the correct inference output.

Pipeline Configuration
The recommended pipeline chain is **[DxPreprocess] → [DxInfer] → [DxPostprocess]**. When using features like `secondary-mode`, the configuration must be consistently applied across all three elements (**DxPreprocess , DxInfer and DxPostprocess**).

QoS Handling
If the downstream sink element has `sync-true`, input buffers may be dropped based on their timestamps to maintain real-time processing performance.

Throttle QoS Events
When **DxRate** sends a Throttle QoS Event, **DxInfer** delays inference by the `throttling_delay` interval. This avoids unnecessary NPU computation in low-framerate pipelines and promotes smooth and consistent streaming.

JSON Configuration
All properties can be configured through a JSON file using the `config-file-path` property. This enables reusable, clean, and scalable configuration of inference behavior.

Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GstObject
             +----GstElement
                   +----GstDxInfer
```

Properties

Name	Description	Type	Default Value
`name`	Sets the unique name of the DxInfer element.	String	`"dxinfer0"`
`config-file-path`	Path to the JSON config file containing the element's properties.	String	`null`
`model-path`	Path to the `.dxnn` model file used for inference.	String	`null`
`preprocess-id`	Specifies the ID of the input tensor to be used for **inference**.	Integer	`0`
`inference-id`	Specifies the ID of the output tensor to be used for **postprocess**.	Integer	`0`
`secondary-mode`	Determines whether to operate in primary mode or secondary mode.	Boolean	`false`

Example JSON Configuration

```json
{
    "preprocess_id": 1,
    "inference_id": 1,
    "model_path" : "./dx_stream/samples/models/YOLOV5S_1.dxnn"
}
```

Notes.

- The pipeline must follow **[DxPreprocess] → [DxInfer] → [DxPostprocess]** for correct and stable operation.
- All properties can also be configured using a JSON file for enhanced usability and flexibility.

3.3 DxPostprocess

DxPostprocess is a pipeline element that handles output tensors generated by the **DxInfer** element. It performs model-specific post-processing, such as classification, detection, or segmentation, and produces structured results by creating or modifying `DXObjectMeta`.

The component supports both built-in and user-defined logic, depending on the configured model type. Processed results are attached to `DXFrameMeta`, enabling downstream tasks such as visualization, event handling, or application-specific decisions.

Key Features

Modes of Operation

- **Primary Mode:** Creates new `DXObjectMeta` objects and attaches them to the corresponding `DXFrameMeta`
- **Secondary Mode:** Modifies existing `DXObjectMeta` based on updated inference results from **DxInfer**

Custom Postprocessing

You can use a custom post-processing function by specifying.

- `library-file-path`: Path to the shared object `(.so)` containing your custom function
- `function-name`: Name of the function to be called

This allows full flexibility to implement custom decoding, filtering, or data transformation logic.

Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GstObject
             +----GstElement
                   +----GstBaseTransform
                         +----GstDxPostprocess
```

Properties

Name	Description	Type	Default Value
`name`	Sets the unique name of the DxPostprocess element.	String	`"dxpostprocess0"`
`config-file-path`	Path to the JSON config file containing the element's properties.	String	`null`
`inference-id`	Assigns an ID to the preprocessed input tensor.	Integer	`0`
`secondary-mode`	Enables Secondary Mode for processing object regions.	Boolean	`false`
`library-file-path`	Path to the custom preprocess library, if used.	String	`null`
`function-name`	Name of the custom preprocessing function to use.	String	`null`

Example JSON Configuration

```
{
    "inference_id": 1,
    "library_file_path": "/usr/share/dx-stream/lib/libpostprocess_yolo.so",
    "function_name": "YOLOV5Pose_1"
}
```

Notes.

- All properties can also be configured using a JSON file for enhanced usability and flexibility.
- For implementing custom preprocess logic, refer to **Chapter 4. Writing Your Own Application** `“Custom Post-Process Library Documentation”`.

3.4 DxTracker

DxTracker is a GStreamer element designed to perform Multi-Object Tracking (MOT) on object detection results in a video stream. It uses bounding box metadata from upstream detection elements (such as **DxPostprocess**) to continuously track and assign consistent IDs across frames.

Currently, **DxTracker** supports the `OC_SORT` algorithm as its default tracking method. For more information about `OC_SORT`, refer to the Object Tracking Algorithms document.

Key Features

Multi-Object Tracking (MOT)
DxTracker tracks the movement of detected objects over time. It assigns unique track IDs to each object. Objects that can **not** be reliably tracked are discarded and **not** forwarded downstream.

Configurable Tracking Algorithms
Tracking algorithms and their parameters are defined in a JSON configuration file specified via the `config-file-path` property.

Example JSON configuration

```json
{
    "tracker_name": "OC_SORT",
    "params": {
        "det_thresh": 0.5,
        "max_age": 30,
        "min_hits": 3,
        "iou_threshold": 0.3,
        "delta_t": 3,
        "asso_func": "iou",
        "inertia": 0.2,
        "use_byte": false
    }
}
```

The configuration file must include

- `tracker_name`: Specifies the tracking algorithm (e.g., `OC_SORT`)
- `params`: Contains algorithm-specific parameters

Default Behavior
If no config file is provided, **DxTracker** uses the `OC_SORT` algorithm with built-in default values.

If only `tracker_name` is specified without params, the `default` parameter values for that algorithm are applied.

Limitations
The number of tracked objects is always less than or equal to the number of detected bounding boxes. Algorithm parameters **must** be defined within the `params` block of the JSON file. These can **not** be set as individual GStreamer properties on the element.

Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GstObject
             +----GstElement
                   +----GstBaseTransform
                         +----DxTracker
```

Properties

Name	Description	Type	Default Value
`name`	Sets the unique name of the DxTracker element.	String	`"dxtracker0"`
`config-file-path`	Path to the JSON config file containing the tracking algorithm and parameters.	String	`null`
`tracker-name`	Specifies the name of the tracking algorithm to use.	String	`"OC_SORT"`

Notes.

- The JSON configuration file is required to customize tracking algorithm parameters.
- If **no** configuration file is provided, **DxTracker** uses `OC_SORT` with default settings.
- All parameters values **must** be defined within the params section of the JSON file.
- Track IDs are only assigned to successfully tracked objects. Objects that can **not** be tracked are removed and **not** passed downstream.

3.5 DxOsd

DxOsd is a GStreamer element that provides On-Screen Display (OSD) capabilities by overlaying object information onto the original video frame.
It uses object metadata (Object Meta) passed from upstream elements, such as DxPostprocess or DxTracker, to render visual elements directly on the frame.

Key Features

Draw Inference Results

- Draws bounding boxes on video frames
- Displays class labels and confidence scores
- Colors boxes by track ID or class
- Supports segmentation maps, human pose, and facial landmarks

H/W Acceleration

DXOSD supports hardware acceleration when used with Rockchip RGA. The RGA accelerates the color and format conversions needed during the drawing process, enabling more efficient resource usage.

Resolution adjustment

The output buffer of DXOSD can be set to a different resolution than the original buffer. Reducing the output resolution lowers the processing load, enabling more efficient performance.

Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GstObject
             +----GstElement
                   +----GstDxOsd
```

Properties

Name	Description	Type	Default Value
`name`	Sets the unique name of the DxOsd element.	String	`"dxosd0"`
`width`	Sets the width of output buffer	Integer	`640`
`height`	Sets the height of output buffer	Integer	`360`

Notes.

- DXOSD produces output buffers in BGR format. Therefore, to display the output, use a displaysink that supports this format or insert a videoconvert element to convert it to a suitable format.
- In DXOSD’s output buffers, the `DXFrameMeta` metadata is removed. Therefore, it is **not** suitable for use upstream of elements that require this metadata.
- Visualizations include bounding boxes, class names, confidence scores, and additional data like segmentation maps, poses, or face landmarks, depending on available metadata.

3.6 DxGather

DXGather is an element designed to merge multiple branches of a stream that originates from the same source, typically split using a GstTee. It recombines these branches back into a single synchronized output stream, ensuring efficient handling of shared content in multi-branch pipelines.

This makes **DXGather** ideal for scenarios where duplicated streams need to be reunited after parallel processing (e.g., detection, tracking, or visualization).

Key Features

PTS-Based Buffer Merging

- Merges buffers from multiple sink pads only when their Presentation Timestaps (PTS) match.
- Ensures that only synchronized frames are merged together.

Buffer and Metadata Handling

- Assumes that buffer content is identical across all input branches.
- Retains one buffer while others are unreferenced.
- Merges metadata from all branches into the resulting output buffer.

Source Stream Assumption

- Only supports streams split from the same source (e.g., using GstTee).
- **Not** suitable for merging streams from different sources. Use **DxMuxer** in that case.

Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GstObject
             +----GstElement
                   +----GstDxGather
```

Properties

Name	Description	Type	Default Value
`name`	Sets the unique name of the DxGather element.	String	`"dxgather0"`

3.7 DxInputSelector

DxInputSelector is a GStreamer element designed for multi-channel video streaming. It merges frames from multiple input streams into a single synchronized output stream, selecting frames based on Presentation Timestamp (PTS) ordering.

Key Features

Stream Selection

- Among N input streams, DxInputSelector selects the buffer with the smallest PTS and forwards it downstream.
- This approach ensures that the output stream maintains temporal consistency across input channels.

Custom Event Handling

- To properly route sticky events received from multiple sink pads, it first sends a custom routing event before forwarding the sticky event downstream.
- The EOS (End-of-Stream) events received from each sink pad are **not** forwarded downstream directly. instead, they are converted into custom EOS events. This ensures that when EOS is received from one of the N connected input streams, the pipeline does **not** receive a global EOS, and only the EOS for that specific stream is handled.
- Only after receiving EOS events from all input channels does the element forward a global EOS event downstream.

Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GstObject
             +----GstElement
                   +----GstDxInputSelector
```

Properties

Name	Description	Type	Default Value
`name`	Sets the unique name of the DxInputSelector element.	String	`"dxinputselector0"`

Notes.

- If an incoming buffer does **not** contain `DXFrameMeta`, the element creates a new `DXFrameMeta` and assigns the sink pad index as the `stream_id`.
- This metadata tagging is essential for downstream elements that rely on stream identification, such as `DxOutputSelector`.

3.8 DxOutputSelector

DxOutputSelector is a GStreamer element used in multi-channel streaming pipelines. It receives a unified stream from `DxInputSelector` and redistributes the buffers to N output pads based on metadata, such as `stream_id`.

Key Features

Buffer Routing

- Buffers arriving at the single sink pad are routed to the correct `src` pad using the `stream_id` field found in the associated `DXFrameMeta`.
- This enables seamless demultiplexing of multiple logical video streams within a unified pipeline.

Custom Event Handling

- `DxOutputSelector` processes sticky events (e.g., caps, segment) using custom routing events generated by `DxInputSelector`. This ensures that each `src` pad receives the correct contextual information.

Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GstObject
             +----GstElement
                   +----GstDxOutputSelector
```

Properties

Name	Description	Type	Default Value
`name`	Sets the unique name of the DxOutputSelector element.	String	`"dxoutputselector0"`

Notes.

- `DxOutputSelector` requires asynchronous operation between its upstream and downstream elements. To achieve this, a `queue` element must be added to each of its src pads. Failure to do so may result in abnormal pipeline hangs.
- The `stream_id` from the `DXFrameMeta` of the buffer received on the sink pad is parsed and used as the index of the src pad. Therefore, if the `stream_id` does **not** correspond to a valid src pad index in `DxOutputSelector`, an error will occur.
- `DxOSD` removes the original buffer and creates a new one for drawing. During this process, `DXFrameMeta` is discarded, which can cause issues if `DxOSD` is placed upstream of `DxOutputSelector`.

3.9 DxRate

DxRate is an element that adjusts the framerate of a video stream to match a defined target framerate.
DxRate archives this by dropping or duplicating frames based on the timestamps of incoming buffers.
If the input stream already matches the desired framerate, no frames are altered.
Otherwise, **DxRate** modifies the output buffer timestamps to maintain consistent intervals, ensuring the smooth and accurate playback at the specified framerate.

Key Features

Framerate Adjustment

- Ensures the output stream matches the target `framerate` by dropping or duplicating frames.
- The output buffer's timestamps are adjusted relative to the first input buffer's timestamp.

Throttle QoS Event

- If `throttle` is set to `true`, a Throttle QoS Event is sent upstream when frames are dropped.
- **DxInfer** can respond this event t by applying throttling, using the `throttling_delay` value.
- To enable this function properly, **DxRate must** be placed downstream of **DxInfer** in the pipeline.

Framerate and Video Speed

- **Framerate** refers to the number of frames per second (FPS) for visual smooth playback.
- **Video speed** refers to playback speed (e.g., fast-forward), which is a separate concept.

Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GstObject
             +----GstElement
                   +----GstBaseTransform
                         +----DxRate
```

Properties

Name	Description	Type	Default Value
`name`	Sets the unique name of the DxRate element.	String	`"dxrate0"`
`framerate`	Sets the target framerate (FPS). This property must be configured.	Integer	`0`
`throttle`	Determines whether to send Throttle QoS Events upstream on frame drops.	Boolean	`false`

Notes.

- The `framerate` property is mandatory and **must** be explicitly set for the element to function.

3.10 DxMsgConv

DxMsgConv is an element that processes inference metadata from upstream **DxPostprocess** elements and converts it into message payloads in various formats.
The converted payloads are then passed to the DxMsgBroker element, which transmits them to a broker server.

A user-defined custom library is required to implement the actual message format logic. Refer to the **Chapter 4. Writing Your Own Application** `“Custom Pre-Process Library Documentation”`.

Key Features

Metadata Conversion

- Converts metadata generated by **DxPostprocess** into structured message formats.
- Requires a custom shared library to define and implement the required message formats.

Integration with DxMsgBroker

- Passes formatted payloads downstream to **DxMsgBroker**, which handles delivery to external systems.

Configuration

- Message format conversion can be customized using a configuration file.

Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GstObject
             +----GstElement
                   +----GstBaseTransform
                         +----GstDxMsgConv
```

Properties

Name	Description	Type	Default Value
`config-file-path`	Path to the configuration file containing private properties for message formats. (optional).	String	`null`
`library-file-path`	Path to the custom message converter library. **Required**.	String	`null`
`message-interval`	Frame interval at which message is converted.	Integer	`1`
`include-frame`	Flag whether to include frame data in the message.	Boolean	`false`

Notes.

- Ensure that the `config-file-path` property specifies a valid configuration file with all necessary properties for the desired message format.

3.11 DxMsgBroker

DxMsgBroker is a sink element that transmits payload messages to an external message broker (e.g., MQTT, Kafka)
It is typically placed after the **DxMsgConv** element in the pipeline and is responsible for delivering the formatted messages to the configured broker server.
Connection settings are managed using the `conn-info` property.

Key Features

Message Sending

- Receives payloads from upstream elements and publishes them to the selected broker.
- Compatible with broker systems such as MQTT and Kafka.

Connection Information

- The `conn-info` property specifies the connection in the format `host:port`.
- Additional connection details (e.g., `SSL/TLS` settings) can be specified through an optional configuration file.

Pipeline Integration

- Used as a sink element, typically placed just after **DxMsgConv** in the pipeline.

Hierarchy

```
GObject
 +----GInitiallyUnowned
       +----GstObject
             +----GstElement
                   +----GstBaseSink
                         +----GstDxMsgBroker
```

Properties

Name	Description	Type	Default Value
`broker-name`	Name of message broker system (mqtt or kafka). **Required**.	String	`mqtt`
`conn-info`	Connection info in the format `host:port`. **Required**.	String	`null`
`topic`	The topic name for publishing messages. **Required**.	String	`null`
`config`	Path to the broker configuration file (optional).	String	`null`

Notes

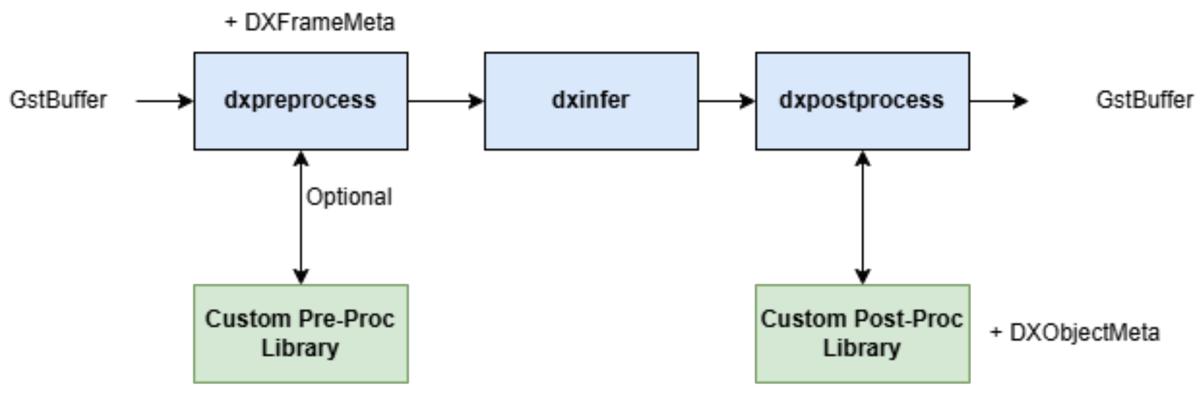
- The `broker-name` property is **required** for selecting a broker system (e.g., MQTT, Kafka).
- The `conn-info` property is **required** and **must** be sent to establish a connection to the broker (the `host:port` format).
- The `topic` property is **required** and defines the topic for messages publication.
- An optional configuration file can provide advanced settings, such as `SSL/TLS` encryption and authentication details.
- Ensure the `libmosquitto-dev` (for MQTT) and `librdkafka` (for Kafka) libraries are installed for proper broker support.

4. Writing Your Own Application

This chapter describes how to integrate a custom AI model and implement user-defined logic within the **DX-STREAM** pipeline. It assumes that your model has already been compiled into `.dxnn` format using **DX-COM**. For details on model compilation, refer to **DX-COM User Manual**.

This guide focuses on how to configure and integrate custom logic into the **DX-STREAM** pipeline using modular elements such as **DxPreprocess, DxInfer,** and **DxPostprocess**.

4.1 Custom Library for Model Inference



The **DX-STREAM** inference pipeline is composed of the following elements.

DxPreprocess

- Allocates `DXFrameMeta` based on the `GstBuffer` received from upstream.
- Performs the preprocessing algorithm as defined by elements properties.
- For custom preprocessing algorithms, a **Custom Pre-Process Library** can be built and integrated.
- See the **dxpreprocess** section in the Elements documentation for details.

DxPostprocess

- Receives the input tensor created by `dxpreprocess`.
- Performs inference using the `dxinfer` element (**DX-RT**).
- Access the output tensor from `dxinfer` and executes the custom postprocessing algorithm defined in a custom library.
- A custom postprocessing implementation is required for each model.
- Example libraries for common vision tasks can be found in `dx_stream/custom_library/postprocess_library`.

4.1.1 **Writing Custom Pre-Process Function**

For models requiring additional preprocessing beyond the default functionality, you can implement a **Custom Pre-Process Function** using a user-defined library.

Implementation Example

```
extern "C" void CustomPreprocessFunc(DXFrameMeta *frame_meta, DXObjectMeta *object_meta,
void* input_tensor)
{
    // Preprocessing logic
}
```

DXFrameMeta

- Access the original buffer through `frame_meta -> _buf` and create an input image by copying the buffer.

DXObjectMeta

- In **Secondary Mode**, metadata for each object is passed to the function.
- In **Primary Mode**, no object metadata is available. (`nullptr`)

input_tensor

- The address of the input tensor generated through user-defined preprocessing.
- It is pre-allocated based on the input tensor size specified by the `dxpreprocess` property and passed to the user. Therefore, users **must not** free or reallocate this memory.

Library Integration

To build the custom object library, use a `meson.build` file and compile as follows.

```
dx_stream_dep = declare_dependency(
    include_directories : include_directories('/usr/local/include/dx_stream'),
    link_args : ['-L/usr/local/lib', '-lgstdxstream'],
)

gst_dep = dependency('gstreamer-1.0', version : '>=1.14',
    required : true, fallback : ['gstreamer', 'gst_dep'])

opencv_dep = dependency('opencv4', required: true)

libcustompreproc = shared_library('custompreproc',
    'preprocess.cpp',
    dependencies: [opencv_dep, gst_dep, dx_stream_dep],
    install: true,
    install_dir: plugins_install_dir + '/lib'
)
```

Specify the library path and function name in the JSON configuration file for `dxpreprocess` as follows.

```
{
    "library_file_path": "./install/gstreamer-1.0/lib/libcustompreproc.so",
    "function_name": "CustomPreprocessFunc"
}
```

4.1.2 **Writing Custom Post-Process Function**

Postprocessing is essential for interpreting and converting the model’s output tensor into meaningful results. To do this, a custom post-process library **must** be implemented to match your model’s architecture and output format.

Output Tensor Parsing

To check the structure of the output tensor, use the following command. This prints the tensor shape for each output.

```
$ parse_model -m YOLOv7.dxnn

Example output:

outputs:
  onnx::Reshape_491, FLOAT, [1, 80, 80, 256]
  onnx::Reshape_525, FLOAT, [1, 40, 40, 256]
  onnx::Reshape_559, FLOAT, [1, 20, 20, 256]
```

The example shows three blobs with NHWC dimensions. Use this information to implement the custom postprocessing logic.

Implementation Example

```
extern "C" void YOLOV7(std::vector<dxs::DXTensor> network_output,
                       DXFrameMeta *frame_meta, DXObjectMeta *object_meta)
{
    // Convert output tensor to bounding box information
}
```

Library Integration

Build the custom library using a `meson.build` script.

```
project('postprocess_yolov5s', 'cpp', version : '1.0.0', license : 'LGPL',
default_options: ['cpp_std=c++11'])

dx_stream_dep = declare_dependency(
    include_directories : include_directories('/usr/local/include/dx_stream'),
    link_args : ['-L/usr/local/lib', '-lgstdxstream'],
)

gst_dep = dependency('gstreamer-1.0', version : '>=1.14',
    required : true, fallback : ['gstreamer', 'gst_dep'])

opencv_dep = dependency('opencv4', required: true)

yolo_postprocess_lib = shared_library('postprocess_yolo',
    'postprocess.cpp',
    dependencies: [opencv_dep, gst_dep, dx_stream_dep, dxrt_dep],
    install: true,
    install_dir: '/usr/share/dx-stream/lib'
)
```

Specify the library path and function name in the JSON configuration file for `dxpostprocess` as follows.

```
{
    "library_file_path": "./install/gstreamer-1.0/lib/libyolo_postprocess.so",
    "function_name": "yolo_post_process"
}
```

4.1.3 **Differences in Post-Processing Logic Based on Inference Mode**

Primary Mode

- Inference is performed on the entire frame.
- Postprocessing is responsible for creating new objects (`DXObjectMeta`) based on the model's output.
- These new objects are then added to the associated `DXFrameMeta` .

Secondary Mode

- Inference is performed per object, based on existing metadata.
- Postprocessing is applied to modify or enrich existing object metadata.
- The `DxObjectMeta` structure contains the input object information, which is passed to the postprocess function for update or enhancement.

4.2 Custom Message Convert Library

Custom message conversion in **DX-STREAM** requires implementing a user-defined library that converts inference metadata into the desired message format.

Functions to Implement

Your custom library **must** define the following four functions.

- `dxmsg_create_context` : Initializes the message conversion context using the provided configuration file
- `dxmsg_delete_context` : Deletes and releases all resources associated with the context
- `dxmsg_convert_payload` : Converts the metadata into the target message format
- `dxmsg_release_payload` : Releases all resources used by the generated payload

Implementation Example

```
#include "dx_msgconvl_priv.hpp"

extern "C" DxMsgContext *dxmsg_create_context(const gchar *file) {
    DxMsgContext *context = g_new0(DxMsgContext, 1);
    context->_priv_data = (void *)dxcontext_create_contextPriv();
    dxcontext_parse_json_config(file, (DxMsgContextPriv *)context->_priv_data);
    return context;
}

extern "C" void dxmsg_delete_context(DxMsgContext *context) {
    dxcontext_delete_contextPriv((DxMsgContextPriv *)context->_priv_data);
    g_free(context);
}

extern "C" DxMsgPayload *dxmsg_convert_payload(DxMsgContext *context, DxMsgMetaInfo
*meta_info) {
    DxMsgPayload *payload = g_new0(DxMsgPayload, 1);
    gchar *json_data = dxpayload_convert_to_json(context, meta_info);
    payload->_size = strlen(json_data);
    payload->_data = json_data;
    return payload;
}

extern "C" void dxmsg_release_payload(DxMsgContext *context, DxMsgPayload *payload) {
    g_free(payload->_data);
    g_free(payload);
}
```

Library Integration

Build the custom library.

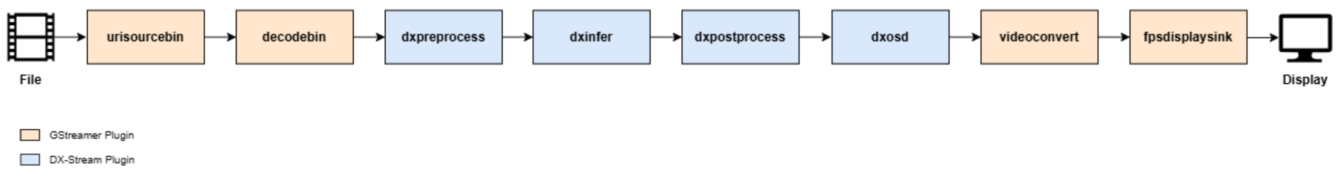
```
custom_msgconv_lib = shared_library('custom_msgconv',
    'dx_msgconvl.cpp',
    include_directories: [include_directories('.')],
    install: true,
    install_dir: '/opt/dx_stream/msgconv/lib'
)
```

5. Pipeline Example

5.1 Single Stream Pipeline

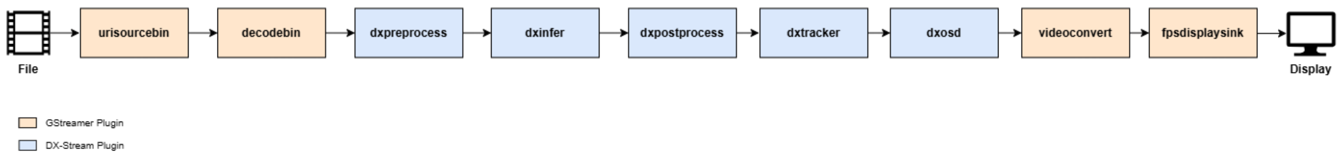
This section describes model inference for a single video stream.
Although the example pipeline performs inference using the YOLOv7 model, it can be easily adapted to various vision tasks with different models. The pipeline below reads video frames from a file and performs a pre-infer-post process tailored to YOLOv7.

The inference results are then drawn using `dxosd`, and the final output is displayed via a display sink.



The pipeline in the figure is defined in `dx_stream/dx_stream/pipelines/single_network/object_detection/run_YOLOV7.sh` and can be used as a reference for execution.

When performing object detection on a single stream input, multi-object tracking can be enabled by adding dxtracker. As shown below, by inserting the tracker after the YOLO object detection, detected objects can be tracked over time.



The pipeline in the figure is defined in `dx_stream/dx_stream/pipelines/tracking/run_YOLOV5S_tracker.sh` and can be used as a reference for execution.

Explanation

Element Descriptions

- **`urisourcebin`**: Specifies the input video file. The `uri` property **must** be set to the file path of the video you wish to process.
- **`decodebin`**: Decodes the input video stream.
- **`dxpreprocess`**: Applies pre-processing according to the configuration file specified in the `config-file-path`.
- **`dxinfer`**: Runs inference using the YOLOv7 model. The model configuration file path is specified in `config-file-path`.
- **`dxpostprocess`**: Post-processes the model's output tensor to extract metadata. The configuration file path is specified in `config-file-path`.
- **`dxtracker`**: Tracks objects detected by the YOLO model using the OC-SORT algorithm.
- **`dxosd`**: Draws object detection results (e.g., bounding boxes, class labels, etc.) on the video frames.
- **`fpsdisplaysink`**: Displays the video frames along with the FPS (frames per second) information. The `sync=false` property ensures that all frames are displayed without being dropped.

Usage Notes

Custom Models

This pipeline is **not** limited to the YOLOv7 model. It can be easily adapted for other AI tasks by updating the corresponding model and configuration files.

- **Classification**: Image classification tasks
- **Segmentation**: Ppixel-wise semantic segmentation
- **Pose Estimation**: Detecting human keypoints

Update the `config-file-path` property in the `dxpreprocess`, `dxinfer`, and `dxpostprocess` elements to match your model's configuration.

Pipeline Behavior

This pipeline runs synchronously because it does **not** include `queue` elements. Each element waits for the previous one to finish processing before continuing. This simplifies data flow but may affect performance on multi-core systems.

Sink Element Options

You can replace `fpsdisplaysink` with other display options.

- **`ximagesink`** : Displays video in an X11 window environment
- **`autovideosink`** : Automatically selects the most suitable video sink for the platform

Object Detection Requirement

- The dxtracker element requires bounding box information. Therefore, object detection **must** precede tracking in the pipeline.

Visualization

- The dxosd element overlays both detection and tracking results. Each object is assigned a unique track ID by the dxtracker element, which is visualized along with the bounding box.

Buffer Handling

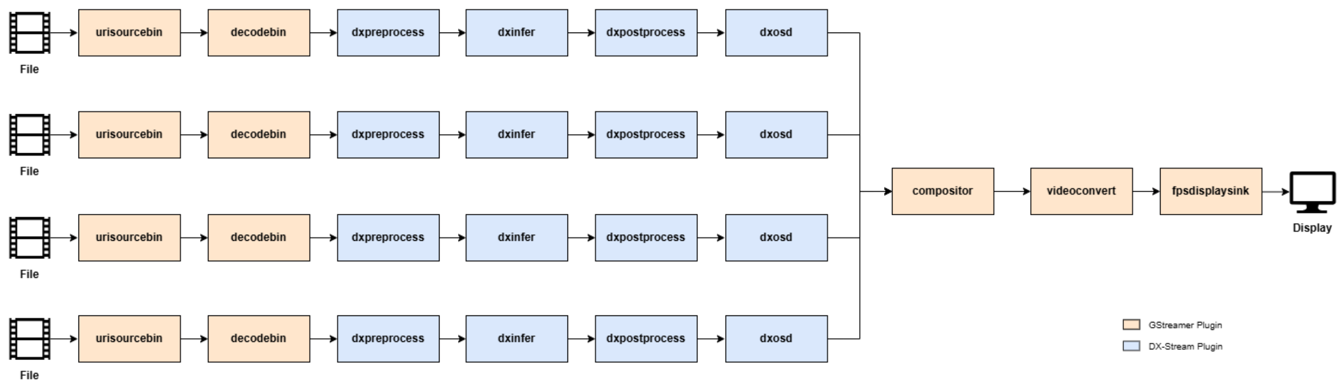
- In this example, `fpsdisplaysink` is set with `sync=false` , which means no frames are -dropped—all frames are displayed as-is.
- If you want to sync playback with the original video frame rate or reduce latency, set `sync=true` .

5.2 Multi Stream Pipeline

There are two possible approaches to building a pipeline structure for multi-stream inference.

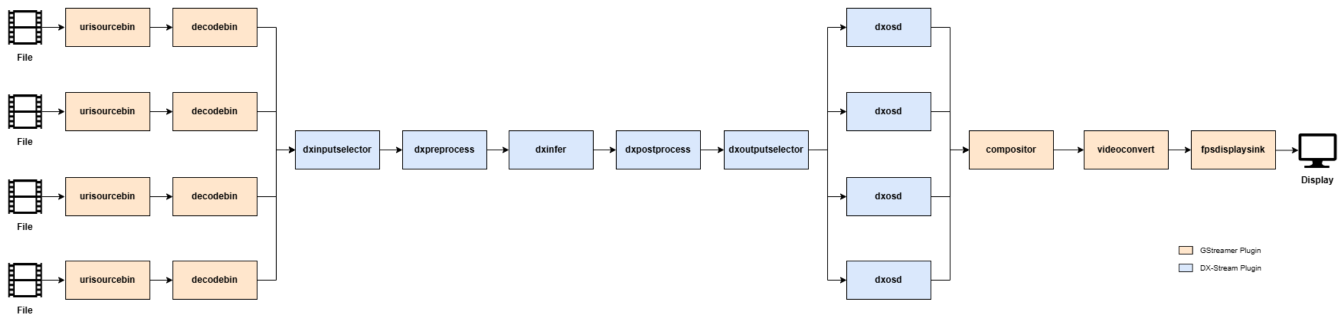
One approach is to configure an independent inference sub-pipeline for each stream, as shown in the figure below.

In this case, each sub-pipeline performs inference asynchronously, maximizing the utilization of hardware resources.



The pipeline in the figure is defined in `dx_stream/dx_stream/pipelines/multi_stream/run_multi_stream_YOLOV5S.sh` and can be used as a reference for execution.

The other approach uses **DxInputSelector** and **DxOutputSelector**.
DxInputSelector is an `N:1` element that receives buffers from multiple input streams and forwards the one with the smallest PTS downstream first.
The selected buffer passes through a single inference pipeline for processing, and **DxOutputSelector** then redistributes the results back to their corresponding streams.



The pipeline in the figure is defined in `dx_stream/dx_stream/pipelines/multi_stream/run_multi_stream_single_infer_YOLOV5S.sh` and can be used as a reference for execution.

Explanation

Element Descriptions

- `conpositor` : Draws multiple stream buffers received through sink pads at specified positions. In the example above, it is used for tiled display of inference results from each stream.
- `dxinputselector` : Selects the buffer with the smallest PTS among multiple input streams received through sink pads and pushes it downstream.
- `dxoutputselector` : Routes buffers received from upstream back into multiple output streams.

Usage Notes

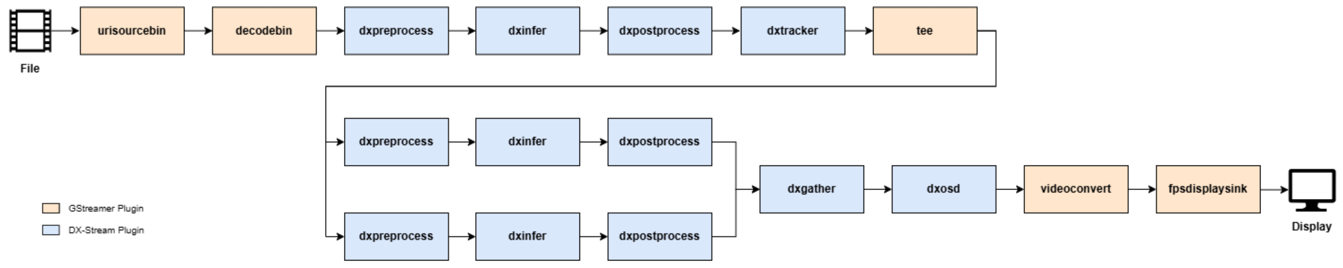
- Depending on the model size, using a multi sub-pipeline structure may put a burden on memory resources
- While sub-pipeline structures can offer advantages in processing speed, the performance gain may be negligible compared to selector-based pipelines depending on the environment.

5.3 Secondary Inference Pipeline

This section describes the secondary mode inference pipeline, which performs object-level inference based on objects detected in the primary mode. A single input stream is processed through primary inference for object detection, followed by tracking.

Using a `tee` element, the buffer is duplicated into two streams. One stream proceeds to secondary inference, where additional tasks such as face detection and Re-ID feature extraction are performed, with the results stored in each object's metadata.

Then, the `dxgather` element collects the outputs from the secondary inference streams and merges them into a single buffer, converting the data back into a unified stream. Finally, the results are rendered for visualization.



The pipeline in the figure is defined in `dx_stream/dx_stream/pipelines/secondary_mode/run_secondary_mode.sh` and can be used as a reference for execution.

Explanation

Element Descriptions

- **`tee`**: Duplicates a single stream into multiple streams. The duplicated output buffers originate from the same source buffer.
- **`dxgather`**: Merges buffers originating from the same source but split by tee, and combines their inference results into a single buffer.

Usage Notes

Configure file setting

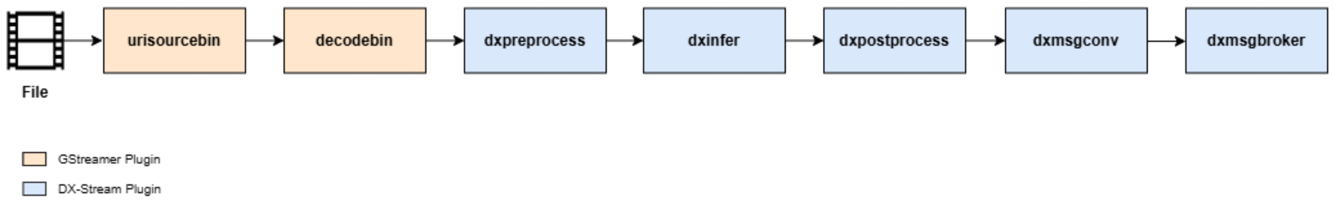
- This pipeline runs three different models concurrently. Therefore, be careful to configure each model's preprocess_id and inference_id properly to avoid unexpected behavior.

Secondary post process

- When implementing a custom post-processing function for secondary inference mode, modify the provided DXObjectMeta directly. Removing or re-allocating the metadata may lead to unintended behavior.

5.4 MsgBroker Pipeline

The pipeline demonstrates how to process a local video file with the YOLOv7 model for object detection, convert the results into a structured message format (e.g., JSON), and publish them to a message broker such as MQTT or Kafka for downstream applications.



The pipeline in the figure is defined in `dx_stream/dx_stream/pipelines/broker/*.sh` and can be used as a reference for execution.

Explanation

Element Descriptions

- **`dxmsgconv`** : element that processes inference metadata from upstream **DxPostprocess** elements and converts it into message payloads in various formats.
- **`dxmsgbroker`** : sink element that transmits payload messages to an external message broker (e.g., MQTT, Kafka)

Usage Notes

Metadata Conversion

The `dxmsgconv` element requires a configuration file to define the message format. Update the `config-file-path` property to point to your message conversion configuration file.

Message Publishing

The `dxmsgbroker` element requires the following properties.

- **`conn-info`** : Connection information for the broker in the format `[host]:[port]` .
- **`topic`** : The topic name for publishing the messages.

Broker Server Application

MQTT

MQTT (Message Queuing Telemetry Transport) is a lightweight, publish/subscribe messaging protocol.

- **Install Mosquitto Server/Client**

```
$ sudo apt update
$ sudo add-apt-repository ppa:mosquitto-dev/mosquitto-ppa
$ sudo apt install mosquitto mosquitto-clients

$ sudo systemctl status mosquitto
● mosquitto.service - Mosquitto MQTT Broker
     Loaded: loaded (/lib/systemd/system/mosquitto.service; enabled; vendor preset:
enabled)
     Active: active (running) since Wed 2024-09-11 15:49:49 KST; 18min ago
       Docs: man:mosquitto.conf(5)
             man:mosquitto(8)
   Main PID: 19577 (mosquitto)
     Tasks: 1 (limit: 76827)
     Memory: 904.0K
     CGroup: /system.slice/mosquitto.service
            └─19577 /usr/sbin/mosquitto -c /etc/mosquitto/mosquitto.conf
```

- **CA certificates / Server keys, certificates**

```
** Test Root CA
$ openssl genrsa -out ca.key 2048
$ openssl req -new -x509 -days 360 -key ca.key -out ca.crt -subj "/C=KR/ST=KK/L=SN/
O=DXS/OU=Test/  CN=TestCA"

** Server Key and Certificate(Server's CN must match the host)
$ openssl genrsa -out server.key 2048
$ openssl req -new -out server.csr -key server.key -subj "/C=KR/ST=KK/L=SN/O=DXS/
OU=Server/  CN=DXS-BROKER"
$ openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out
server.crt -days 360
$ openssl verify -CAfile ca.crt server.crt

$ ls -al
ca.crt  ca.key  ca.srl  server.crt  server.csr  server.key

$ cat /etc/mosquitto/mosquitto.conf
listener 8883
cafile /etc/mosquitto/ca_certificates/ca.crt
certfile /etc/mosquitto/certs/server.crt
```

```
keyfile /etc/mosquitto/certs/server.key
...
```

- **Id/Password**

```
$ sudo mosquitto_passwd -c /etc/mosquitto/passwd user        ##create passwd file and
add id
Password:
Reenter password:
$ sudo mosquitto_passwd -b /etc/mosquitto/passwd user1 1234  ##add id
$ sudo mosquitto_passwd -D /etc/mosquitto/passwd user1       ##remove id
$ sudo chmod 644 /etc/mosquitto/passwd

$ cat /etc/mosquitto/mosquitto.conf
password_file /etc/mosquitto/passwd
allow_anonymous false
...
```

And for secure connection, you need to set the config file of dxmsgbroker as follows.

- `broker_mqtt.cfg`

```
### username / password
#username = user
#password = 1234

### client-id, if not defined a random client idwill be generated
##client-id = client1

### enable ssl/tls encryption
tls_enable = 1

### the PEM encoded Certificate Authority certificates
#tls_cafile = <path to CA certificate file>

### the path to a file containing the CA certificates, 'openssl rehash <path to capath>'
each time   you add/remove a certificate.
tls_capath = <path to directroy containing CA certificates>

### Path to the PEM encoded client certificate.
tls_certfile = <path to certificate file>

### Path to the PEM encoded client certificate.
tls_keyfile = <path to key file>
```

Kafka

Apache **Kafka** is a distributed event streaming platform designed for high-throughput, fault-tolerant, and scalable handling of real-time data streams.

- **Install Kafka**

```
$ sudo apt update
$ sudo apt-get install default-jdk
$ wget https://downloads.apache.org/kafka/3.9.0/kafka_2.13-3.9.0.tgz
$ tar -xzf kafka_2.13-3.9.0.tgz
$ cd kafka_2.13-3.9.0
```

- **Zookeeper**

```
$ bin/zookeeper-server-start.sh config/zookeeper.properties
```

- **Kafka server**

```
$ bin/kafka-server-start.sh config/server.properties
```

- **Producer**

```
$ bin/kafka-console-producer.sh --topic test-topic --bootstrap-server localhost:9092
```

- **Consumer**

```
$ bin/kafka-console-consumer.sh --topic test-topic --from-beginning --bootstrap-server
localhost:9092
```

For more information about Kafka, refer to https://kafka.apache.org/

- **Kafka Client(GstDxMsgBroker) Configuration**

GstDxMsgBroker uses standard OpenSSL PEM keys.

- **Secure Connection**

To set up Kafka SSL, begin by generating and configuring SSL certificates for both the Kafka broker and the client. Follow the detailed instructions in the librdkafka SSL guide to create and validate these certificates. Update the Kafka broker configuration file (server.properties) with the appropriate SSL settings, such as keystore, truststore paths, and passwords. Finally, configure the Kafka client with SSL options, including the paths to the keystore and truststore files, ensuring secure communication between the client and broker.

- **Kafka Server Configuration**

`server.properties`

```
# SSL
ssl.protocol=TLS
ssl.enabled.protocols=TLSv1.2,TLSv1.1,TLSv1
ssl.keystore.type=JKS
ssl.keystore.location=<path to broker keystore file>
ssl.keystore.password=<broker keystore password>
ssl.key.password=<key password>
ssl.truststore.type=JKS
ssl.truststore.location=<path to broker truststore file>
ssl.truststore.password=<broker truststore password>
# To require authentication of clients use "require", else "none" or "request"
ssl.client.auth = required
```

And for secure connection, you need to set the config file of dxmsgbroker as follows.

`broker_kafka.cfg`

```
#######################################
### kafka client configuration
#######################################
[kafka]
### for frame data
message.max.bytes=10485760

### for secure transmission
#security.protocol=ssl
## CA certificate file for verifying the broker's certificate.
ssl.ca.location=<path to CA certificate file>

## Client's certificate
ssl.certificate.location=<path to client certificate file>

## Client's key
ssl.key.location=<path to client key file>

## Key password, if any
ssl.key.password=KEY_PASSWORD
```

Script Descriptions

The server application, which receives data from the MQTT, KAFKA pipeline to log messages or render images, can be found in `/usr/share/dx-stream/bin`.

- `mqtt_sub_example.py` (mqtt_sub_example)

Runs an MQTT server that logs the messages received.

```
python3 /usr/share/dx-stream/bin/mqtt_sub_example.py -p <PORT> -n <HOSTNAME> -t <TOPIC>
```

- `mqtt_sub_example_frame.py`

Displays frames included in the messages using OpenCV.

Note. The `include_frame` option in the DxMsgConv config JSON **must** be set to true for this feature to work.

```
python3 /usr/share/dx-stream/bin/mqtt_sub_example_frame.py -p <PORT> -n <HOSTNAME> -t
<TOPIC>
```

- `kafka_consume_example.py` (kafka_consume_example)

Runs an KAFKA server that logs the messages received.

```
python3 /usr/share/dx-stream/bin/kafka_consume_example.py <HOSTNAME> <TOPIC>
```

- `kafka_consume_example_frame.py`

Displays frames included in the messages using OpenCV.

Note. The include_frame option in the DxMsgConv config JSON **must** be set to true for this feature to work.

```
python3 /usr/share/dx-stream/bin/kafka_consume_example_frame.py <HOSTNAME> <TOPIC>
```

6. Troubleshooting and FAQ

6.1 Installation

TBD

6.2 Runtime

6.2.1 Rendering Issues

Rendering in GStreamer is generally performed using a sink element. If the specified displaysink element is **not** supported on the current PC environment, the pipeline may exhibit abnormal behavior. Therefore, it is essential to select an appropriate displaysink element based on the system's environment:

- For CPU-based environments, ximagesink or xvimagesink can be used.
- For GPU-based environments, glimagesink or similar elements are recommended.
- Add videoconvert before display sink element

```
$ gst-launch-1.0 ..... ! videoconvert ! autovideosink
```

6.2.2 Buffer Delays in Sink Element

When a PC has low performance or is under heavy load, GStreamer pipelines may experience delays in delivering buffers to the sink element (e.g., `ximagesink`, `glimagesink`, etc.). This can result in issues such as

- Stuttering or lagging video playback.
- Warning messages like "buffering too slow" or "dropped frames."
- Overall pipeline performance degradation.

Optimize PC Performance

- Terminate Background Processes: Free up CPU/GPU resources by closing unnecessary programs.
- Use Lower-Resolution Videos: Reduce decoding and rendering workload by downscaling input video resolution.

Optimize the GStreamer Pipeline

Add queue Elements:

- Insert queue elements at bottleneck points to decouple processing speeds.

```
gst-launch-1.0 filesrc location=video.mp4 ! decodebin ! queue ! autovideosink
```

Use Asynchronous Rendering:

- Disable real-time playback synchronization with `sync=false`.

```
gst-launch-1.0 ... autovideosink sync=false
```

6.2.3 Kafka Pipeline

```
%3|1736310124.667|FAIL|rdkafka#producer-1| [thrd:localhost:9092/bootstrap]: localhost:
9092/bootstrap: Connect to ipv4#127.0.0.1:9092 failed: Connection refused (after 0ms in
state CONNECT)
ERROR: Pipeline doesn't want to pause.
ERROR: from element /GstPipeline:pipeline0/GstDxMsgBroker:dxmsgbroker0: GStreamer error:
state change failed and some element failed to post a proper error message with the
reason for the failure.
```

The following error might occur when using a message broker with Kafka, as shown in the log above. To resolve this, verify whether the Kafka broker is running. If it is **not** running, perform the steps below to start it and ensure normal operation.

- Check Running Status

```
$ ps -ef | grep kafka
```

- If **not** running, follow these steps

Create a utilities directory and install required dependencies

```
$ mkdir utils && cd utils
$ sudo apt update
$ sudo apt-get install default-jdk
$ wget https://downloads.apache.org/kafka/3.9.0/kafka_2.13-3.9.0.tgz
$ tar -xzf kafka_2.13-3.9.0.tgz
$ cd kafka_2.13-3.9.0
```

In separate terminal sessions, execute the following commands

Terminal 1: Start Zookeeper

```
$ bin/zookeeper-server-start.sh config/zookeeper.properties
```

Terminal 2: Start Kafka Broker

```
$ bin/kafka-server-start.sh config/server.properties
```

Keep these terminals running while executing the pipeline to ensure proper operation.

7. Change Log

7.1 Version 2.0.0 (Aug 2025)

Changed

- Code Examples: The PostProcess examples have been separated and implemented on a per-model basis for clarity.
- DX-RT v3.0.0 Compatibility: This version has been updated to ensure full compatibility with DX-RT v3.0.0.
- Model Support: Inference is now restricted to models (DXNN v7) produced by DX-COM v2.0.0 and later versions. Modified dx-gather event handling logic.
- Removed unnecessary print statements.
- feat: enhance build script and update installation documentation
- Added OS and architecture checks in the build script
- Updated CPU and OS specifications in the installation documentation for clarity

Fixed

- Bug Fix: Addressed and alleviated a processing delay issue within the dx-inputselector.
- Corrected a post-processing logic error in the SCRFD model when in secondary inference mode.
- Fixed a bug in dx_rt that occurred when processing multi-tail models.
- feat: improve error handling for setup scripts
- feat: add support for X11 video sink on Ubuntu 18.04 across multiple scripts
- Force X11 video sink on Ubuntu 18.04
- Improved compatibility across OS versions
- Updated multiple pipeline scripts
- Added OS version check for Ubuntu 18.04

Added

- feat: add uninstall script and enhance color utility functions
- Introduced a new uninstall.sh script for cleaning up project files and directories

7.2 Version 1.7.0 (Jul 2025)

Changed

- Improved the buffer queue management mechanism. Instead of locking inputs based on queue size within the push thread, the system now adds a req_id to the buffer and utilizes a wait function for more efficient processing.
- auto run setup script when a file not found error occurs during example execution
- apply colors and handle errors in scripts

Fixed

- dxpreprocess, dxosd: Resolved a video corruption issue that occurred in some streams. The problem was traced to incorrect stride and offset calculations from GstVideoMeta. The calculation now correctly uses GstVideoInfo included in the caps, ensuring stable video rendering.