

DXNN Application (DX-APP) User Manual

2026/10/23 (v3.3.0)

DEEPX.ai

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Table of contents

1. DXNN Application Overview 4
1.1 DEEPX SDK Architecture 4
1.2 DX-APP Features 5
1.3 DX-APP Core Design & Capabilities 8
2. DX-APP Quick Start 10
2.1 Step 1. Verify the NPU and DX-RT 10
2.2 Step 2. Install the Dependencies 10
2.3 Step 3. Download Models and Sample Videos 10
2.4 Step 4. Build 11
2.5 Step 5. Run an Example 11
2.6 Next Steps 12
3. DX-APP Installation and Build 13
3.1 Overview & System Requirements 13
3.2 Installation on Linux 13
3.3 Installation on Windows 21
4. DX-APP C++ Example 32
4.1 DX-APP C++ Usage Guide 32
4.2 DX-APP C++ Example Tests 42
5. DX-APP Python Example 52
5.1 DX-APP Python Usage Guide 52
5.2 DX-APP Python Example Tests 64
6. DX-APP Multi-Model Graph 70
6.1 Overview 70
6.2 Build and Run 71
6.3 Sample Graphs 74
6.4 Writing a Graph 76
6.5 Decision Logic 81
6.6 Several Sources 84
6.7 Command-Line Options 85
6.8 Checking and Troubleshooting 87
6.9 Python: dx_graph 88
6.10 Standalone Project 89
6.11 Limitations in This Release 90
7. DX-APP Post-processing 91
7.1 DX-APP C++ Post-processing 91

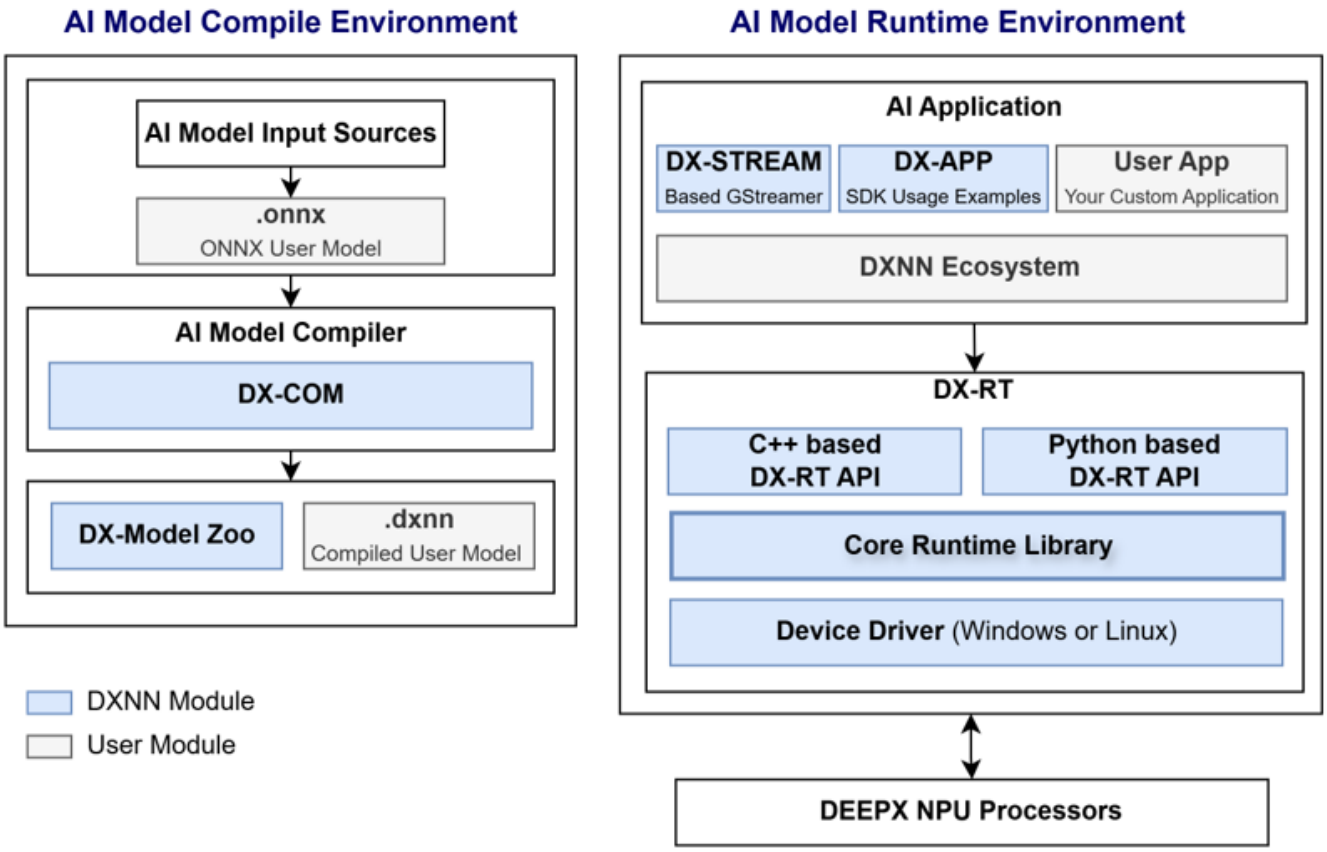
ⓒ Copyright 2026 DEEPX All Rights Reserved.

7.2 DX-APP Python Post-processing 96
8. DX-APP Architecture and Usage Reference 102
8.1 Core Concepts & Architecture 102
8.2 Usage Reference 105
8.3 Advanced Features 109
9. DX-APP Developer Guide 113
9.1 DX-APP DX Tool Guide 113
9.2 DX-APP Example Source Structure 127
9.3 DX-APP YOLO Customizing Guide 143
9.4 DX-APP Agent-Driven Development Guide 154
10. Appendix 176
10.1 Third-Party License Notice 176
11. RELEASE_NOTES 178
11.1 DX-APP v3.3.0 / 2026-10-23 178
11.2 DX-APP v3.2.2 / 2026-08-14 180
11.3 DX-APP v3.2.1 / 2026-08-03 181
11.4 DX-APP v3.2.0 / 2026-07-22 181
11.5 DX-APP v3.1.1 / 2026-05-06 184
11.6 DX-APP v3.1.0 / 2026-04-10 184
11.7 DX-APP v3.0.2 / 2026-02-26 185
11.8 DX-APP v3.0.1 / 2026-02-09 186
11.9 DX-APP v3.0.0 / 2026-01-16 186
11.10 DX-APP v2.1.0 / 2025-11-28 190
11.11 DX-APP v2.0.0 / 2025-09-08 192
11.12 DX-APP v1.11.0 / 2025-07-23 193
11.13 DX-APP v1.10.0 / 2025-06-17 194

ⓒ Copyright 2026 DEEPX All Rights Reserved.

1. DXNN Application Overview

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 highly efficient inference on DEEPX NPU.

© Copyright 2026 DEEPX All Rights Reserved.

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 collection of runnable example applications that execute compiled models on DEEPX NPUs through DX-RT. It provides C++ and Python examples, each in sync and async variants, across 29 AI task categories, including image classification, object detection, face detection, pose estimation, segmentation, depth estimation, and super resolution. DX-APP is intended both as a quick-start runtime package for users and as a reusable application baseline for further customization.

Examples are organized by task, family, and variant under `src/cpp_example/` and `src/python_example/`. Each tree has its own shared runtime layer (`common/`) providing base interfaces, processors, runners, visualizers, and utilities via a factory pattern. Additionally, `src/postprocess/` provides C++ post-processing consumed by pybind11 bindings for `*_cpp_postprocess.py` variants.

1.2 DX-APP Features

DX-APP provides ready-to-use examples for image classification, object detection, segmentation, pose estimation, and related inference tasks.

You can quickly evaluate inference capabilities without modifying the source code and then expand toward task- and model-specific customization. The current example set is backed

ⓒ Copyright 2026 DEEPX All Rights Reserved.

by a task/family/variant source layout, shared post-processing modules, and a DX-ModelZoo-based asset preparation flow.

> **NOTE**
>
> Application performance may vary depending on host system specifications. Each demo includes pre-processing, post-processing, and graphics processing operations.

> **Related Guides**
>
> To run a first example in a few steps, refer to DX-APP Quick Start. For installation and build steps, refer to DX-APP Installation and Build. For several models in one pipeline, refer to DX-APP Multi-Model Graph. For agent workflows, refer to DX-APP Agent-Driven Development Guide. For C++ and Python example usage details, refer to DX-APP C++ Usage Guide and DX-APP Python Usage Guide.

1.2.1 Example Catalog

DX-APP examples are optimized to showcase pre-compiled models on DEEPX NPUs with minimal setup. Assets are prepared through the standard setup flow, and representative examples can be executed using images, videos, or live camera input.

EfficientNet-Lite0, YOLOv8N and BiSeNetV1 are not part of the default `./build.sh` (which builds the `run_demo` targets); build them with `./build.sh --all` or `./build.sh --target <variant>_sync <variant>_async`.

Classification (EfficientNet-Lite0)

- Input: image/video/camera/RTSP (e.g., `224x224` input)
- Output: Top-5 classes (`top_k` in `config.json`)
- Example run

```
./bin/efficientnet-lite0_224x224_async -m ./assets/models/efficientnet-lite0_224x224.dxnn -i ./sample/ILSVRC2012/0.jpeg -l 1
./bin/efficientnet-lite0_224x224_sync  -m ./assets/models/efficientnet-lite0_224x224.dxnn -i ./sample/ILSVRC2012/0.jpeg -l 1
```


© Copyright 2026 DEEPX All Rights Reserved.

Object Detection (YOLOv8N)

- Input: image/video/camera/RTSP
- Output: boxes rendered and logged
- Example run

```
./bin/yolov8-n_640x640_async -m ./assets/models/yolov8-n_640x640.dxnn -i ./sample/img/sample_kitchen.jpg --no-display -l 1
./bin/yolov8-n_640x640_sync  -m ./assets/models/yolov8-n_640x640.dxnn -i ./sample/img/sample_kitchen.jpg --no-display -l 1 -s
```

Face Detection (SCRFD)

- Input: image/video/camera/RTSP
- Output: face boxes, landmarks, log
- Example run

```
./bin/scrfd-500m_640x640_async -m ./assets/models/scrfd-500m_640x640.dxnn -i ./sample/img/sample_face.jpg --no-display -l 1
./bin/scrfd-500m_640x640_sync  -m ./assets/models/scrfd-500m_640x640.dxnn -i ./sample/img/sample_face.jpg --no-display -l 1 -s
```

Pose Estimation (YOLOv8s Pose)

- Input: image/video/camera/RTSP
- Output: person boxes + keypoints
- Example run

```
./bin/yolov8-s-pose_640x640_async -m ./assets/models/yolov8-s-pose_640x640.dxnn -i ./sample/img/sample_kitchen.jpg --no-display -l 1
./bin/yolov8-s-pose_640x640_sync  -m ./assets/models/yolov8-s-pose_640x640.dxnn -i ./sample/img/sample_kitchen.jpg --no-display -l 1 -s
```


© Copyright 2026 DEEPX All Rights Reserved.

Semantic Segmentation (DeepLabV3+)

- Input: image/video/camera/RTSP
- Output: segmentation masks rendered, results saved
- Example run

```
./bin/deeplabv3plus_mobilenetv1_512x512_async -m ./assets/models/
deeplabv3plus_mobilenetv1_512x512.dxnn -i ./sample/img/sample_parking.jpg --no-display -l 1
./bin/deeplabv3plus_mobilenetv1_512x512_sync  -m ./assets/models/
deeplabv3plus_mobilenetv1_512x512.dxnn -i ./sample/img/sample_parking.jpg --no-display -l 1 -s
```

Semantic Segmentation (BiSeNetV1)

- Input: image/video/camera/RTSP
- Output: segmentation masks rendered
- Example run

```
./bin/bisenetv1_1024x2048_async -m ./assets/models/bisenetv1_1024x2048.dxnn -i ./sample/
img/sample_parking.jpg --no-display -l 1
./bin/bisenetv1_1024x2048_sync  -m ./assets/models/bisenetv1_1024x2048.dxnn -i ./sample/img/
sample_parking.jpg --no-display -l 1 -s
```

1.3 DX-APP Core Design & Capabilities

DX-APP is engineered to maximize NPU throughput while minimizing CPU-side bottlenecks.

1.3.1 Post-Processing Libraries

Post-processing for the Python `*_cpp_postprocess.py` variants is implemented in optimized C++ libraries (`src/postprocess/`).

- **C++ Binding:** These libraries are exposed to Python via `pybind11` (`dx_postprocess`), so those variants run model-specific decoding (NMS, box scaling, mask generation) at C++ speed.
- **Variant Choice:** The other Python examples use Python post-processing, and the C++ examples use their own shared processors under `src/cpp_example/common/processors/` .

1.3.2 Execution Paradigms: Sync vs. Async

© Copyright 2026 DEEPX All Rights Reserved.

Examples are provided in two variants to help developers optimize for their specific use cases

- **Synchronous (Sync):** Sequential execution (**Pre** → **Inference** → **Post**). Best for single-image analysis and simplified debugging.
- **Asynchronous (Async):** A multi-threaded design using `RunAsync()` to overlap stages. While the NPU processes Frame **N**, the CPU prepares Frame **N+1** and post-processes Frame **N-1**. This is critical for maximizing **FPS** on real-time video or RTSP streams.

1.3.3 Performance Profiling & Bottleneck Analysis

Every example in DX-APP—regardless of the language (C++/Python) or execution paradigm (Sync/Async)—is equipped with a built-in performance profiler. Upon completion, the console outputs a **Performance Summary** that serves as a critical tool for application tuning.

Key Metrics Collected

- **Stage Latency:** Precise timing for each stage of the pipeline
  : **Read:** Frame/image reading and decoding
  : **Pre-processing:** Resizing and normalization
  : **NPU Inference:** Pure execution time on the DEEPX NPU via DX-RT
  : **Post-processing:** Result decoding (NMS, box scaling, etc.)
  : **Display/I/O:** Time taken to render or save the output
- **End-to-End Throughput (FPS):** The overall frames per second achieved by the entire system.

Strategic Objectives

- **Bottleneck Identification:** Instantly determine if the system is limited by CPU-side tasks (Pre/Post-processing) or NPU throughput. For instance, if post-processing latency is high in a Python script, you can strategically switch to the **C++ Binding** (`dx_postprocess`) variant.
- **Architectural Benchmarking:** Quantitatively validate how much performance is gained by moving from a **Synchronous** to an **Asynchronous** design.
- **Resource Optimization:** Help developers balance NPU utilization and CPU overhead to find the "sweet spot" for their specific hardware and commercial use case.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

2. DX-APP Quick Start

This chapter takes a fresh DX-APP installation to its first inference on a DEEPX NPU in five steps. Each step links to the chapter that explains it in detail.

> **Before you start**
>
> The NPU device driver and **DX-RT 3.5.0 or later** must already be installed. The supported operating systems, the hardware requirements and the driver and DX-RT installation are described in DX-APP Installation and Build.

2.1 Step 1. Verify the NPU and DX-RT

```
# NPU device and driver status
dxcli -s

# DX-RT version: the first line must be DXRT v3.5.0 or later
dxcli --version
```

If `dxcli -s` fails, or DX-RT is older than 3.5.0, install or upgrade the NPU driver and DX-RT before you continue (*Installation on Linux* in DX-APP Installation and Build). The models that `setup.sh` downloads need DX-RT 3.5.0, and an older runtime fails only when a model is loaded.

2.2 Step 2. Install the Dependencies

```
# Build tools, CMake and OpenCV
./install.sh --all
```

2.3 Step 3. Download Models and Sample Videos

```
# Only the models that run_demo.sh and this chapter use
./setup.sh --demo-models
```


© Copyright 2026 DEEPX All Rights Reserved.

```
# Or choose categories and models from a menu
./setup.sh

# Or download every registered model, without prompts
./setup.sh --all
```

Models are saved in `assets/models/` and sample videos in `assets/videos/`. Every `setup.sh` option is listed in DX-APP Installation and Build.

An example run without `-m` downloads its default model if it is missing, but sample videos are not downloaded on demand, so run this step before the video examples below.

2.4 Step 4. Build

```
# Default (minimal) build: the C++ examples run_demo.sh uses, and dx_postprocess
./build.sh

# Full build: every C++ example, the multi-model graph CLI and, when pybind11 is installed in the build's Python, dx_graph
./build.sh --all

# Only the targets you name
./build.sh --target yolov9-s_640x640_sync yolov9-s_640x640_async
```

Executables are written to `bin/`. `./build.sh --target list` lists the build targets (names that start with a digit, such as `3ddfa-v2_mobilenetv1_120x120_sync`, are not listed but build with `--target`), and `./build.sh --clean` rebuilds from scratch.

2.5 Step 5. Run an Example

The default build includes the YOLO11-N object detector, so this step uses it. Without `-m` (`--model` in Python), an example uses its default model, here `assets/models/yolo11-n_640x640.dxnn`; pass `-m <file>` to run another `.dxnn`.

C++

```
# Image, synchronous
./bin/yolo11-n_640x640_sync -i sample/img/sample_street.jpg

# Video, asynchronous (pipelined for throughput)
./bin/yolo11-n_640x640_async -v assets/videos/snowboard.mp4
```


© Copyright 2026 DEEPX All Rights Reserved.

Python

```
# Image, synchronous, Python post-processing
python src/python_example/object_detection/yolo11/yolo11-n_640x640/yolo11-n_640x640_sync.py
\
    --image sample/img/sample_street.jpg

# Video, asynchronous, C++ post-processing (dx_postprocess): the fastest Python variant
python src/python_example/object_detection/yolo11/yolo11-n_640x640/yolo11-
n_640x640_async_cpp_postprocess.py \
    --video assets/videos/snowboard.mp4
```

Run the Python examples with the Python environment that has `dx_engine`, the DX-RT Python package. The default `./build.sh` installs `dx_postprocess` into the `python3` that is first on `PATH`: activate the `dx_engine` environment before you build, or pass `--venv_path <venv>` (or `--python_exec <python>`).

Result

A window shows the detections, and the console prints a performance summary (latency and FPS). On a machine without a display, add `--no-display`. To keep the result, add `--save`: it is written under `artifacts/`.

2.6 Next Steps

- **Interactive demo**: `./run_demo.sh` runs 26 demo tasks (detection, segmentation, pose, depth, super-resolution and more), one model each, on a sample image or video.
- **Several models at once**: after `./build.sh --all`, `bash run_multi_model_demo.sh` runs one of three sample graph scenarios (hand cascade, logistics volume, DMS) from a menu; `--check` validates all 22 shipped graphs without the NPU. See DX-APP Multi-Model Graph.
- **Options and runtime features**: DX-APP C++ Usage Guide, DX-APP Python Usage Guide and DX-APP Architecture and Usage Reference.
- **Your own model**: DX Tool Guide and DX-APP YOLO Customizing Guide.

© Copyright 2026 DEEPX All Rights Reserved.

3. DX-APP Installation and Build

This guide describes the system requirements and the installation instructions on Linux and Windows to use **DX-APP**.

3.1 Overview & System Requirements

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

Hardware Requirements

- **CPU:** amd64(x86_64), aarch64(arm64)
- **RAM:** 8GB RAM (16GB RAM or higher is recommended)
- **Storage:** 4GB or higher available disk space

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 Library must** be installed. Refer to **DX-RT User Manual** for step-by-step installation instructions.

3.2 Installation on Linux

This section describes the software requirements and installation steps for setting up **DX-APP** on Ubuntu-based and Debian-based systems.

3.2.1 Software Requirements on Linux

© Copyright 2026 DEEPX All Rights Reserved.

To run **DX-APP** on Linux, the following components **must** be installed.

- **OS**: Ubuntu 20.04 / 22.04 / 24.04 / 26.04 (x64) and Debian 12 / 13 (x64)
- **DEEPX M1 Runtime Lib Version (DX-RT)**: v3.5.0 or higher
- **NPU Device Driver**: v2.7.0 or higher (RT and PCIe driver). The runtime's own minimum driver, firmware and `.dxnn` versions are printed by `dxcli --version`.
- **dx_engine** (Python): 3.5.x, matching the runtime's major.minor. `import dx_engine` raises `dx_engine and DX-RT versions do not match` otherwise.

All required components are included in the **DXNN All Suite (DX-AS)** package.

The build needs the following tools and libraries. `./install.sh` installs them (see *DX-APP Application Setup* below).

- **CMake**: 3.16 or higher, with make or ninja
- **Compiler**: C++14-compatible (GCC 7.5 or later, Clang)
- **OpenCV**: 4.2.0 or higher (**4.5.5 recommended**), for image I/O and visualization
- **Python**: 3.8 or higher with pip, for the CMake configure step (it generates the graph model registry), the Python examples and the pybind11 modules

Why DX-RT 3.5.0

`./setup.sh` downloads DX Model Zoo 2_5_0. Those `.dxnn` files are container format **v9**, which DX-RT 3.5.0 is the first runtime to load (DX-RT 3.4.2 refuses them: `Model file format version 9 is not supported`). The C++ examples check the floor when they start and stop with `DXRT library version is too low. (required: >= 3.5.0, current: <version>)`; the check
runs after the engine is created, so an older runtime may report its own error on the `.dxnn` first. The Python examples check for DX-RT 3.0.0 or later only, so on DX-RT 3.0.0 to 3.4.x they start and then fail while loading the model, with the DX-RT error above. Check `dxcli --version` before running them. The multi-model graph reads the container version before it creates an engine and names the problem (see DX-APP Multi-Model Graph).

Tested with DX-RT 3.5.0, NPU driver 2.7.0 (RT and PCIe), firmware 2.7.6 and `dx_engine` 3.5.0 on a DX-M1 M.2 card (x86_64, Ubuntu 24.04).

3.2.2 Prerequisites Setup

Step 1. Install DX-RT Device Driver

© Copyright 2026 DEEPX All Rights Reserved.

To set up the build Environment, refer to **Section. Linux Device Driver Installation** in **DX-RT User Manual**.

Once the DX-RT device driver is installed, the system should include both the PCIe driver and the runtime driver. You can verify the installation by checking the loaded kernel modules.

```
lsmod | grep dx

# dxrt_driver 53248 2
# dx_dma 475136 7 dxrt_driver
```

Step 2. Install DX-RT Library

To build and install the DX-RT library, refer to **Section. Build Guide for Cross-compile** in **DX-RT User Manual**.

Once **DX-RT** is built, the runtime library and header files are installed in the following directory.

- Libraries: `/usr/local/lib`
- Headers: `/usr/local/include`

```
set(DXRT_INSTALLED_DIR /usr/local)
```

If necessary, you can modify the installation path by editing `cmake/toolchain.x86_64.cmake`.

3.2.3 DX-APP Application Setup

Step 1. DX-APP Installation Options

To list the **DX-APP** installation options, run the following command.

```
./install.sh --help
```

Without options, `./install.sh` does the same as `./install.sh --all` : it installs the build dependencies (CMake, GCC, Ninja and the Python packages) and OpenCV from the system packages, using `sudo`.

Step 2. OpenCV Installation Options

ⓒ Copyright 2026 DEEPX All Rights Reserved.

If you want to enable CPU/GPU acceleration, OpenCV **must** be manually installed on your system.
During the OpenCV build process, setting the following flags is needed.

- `TBB=ON, IPP=ON, CUDA=ON`

If OpenCV is already installed, manually set the `OpenCV_DIR` path in your toolchain file.

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR x86_64)
set(DXRT_INSTALLED_DIR /usr/local)
set(OpenCV_DIR /your/opencv/installation/dir)
set(onnxruntime_LIB_DIRS /usr/local/lib)
```

Step 3. Build DX-APP and Download Assets

For the shortest path from installation to a first inference, see DX-APP Quick Start. This step is the reference for the build and asset options.

To build `dx_app`, run the following command.

```
./build.sh ## Defaults to minimal build (run_demo C++ targets). Use --clean for a clean build.
```

To download required models and sample videos, run the following command.

```
./setup.sh
```

Assets are downloaded and placed in the `assets/` directory. Without options, `./setup.sh` opens a menu to select categories and models; use `./setup.sh --all` to download everything.

- **Models:** stored under `assets/models/`
- **Videos:** stored under `assets/videos/`

> **NOTE**
>
> The sample models (`.dxnn`) and dataset images are provided for **evaluation and development purposes only** and are not licensed for commercial deployment. For full license details, see Appendix: Third-Party License Notice.

© Copyright 2026 DEEPX All Rights Reserved.

`setup.sh` Options

Option	Description
`--all`	Download all models non-interactively
`--dry-run`	List models that would be downloaded without downloading
`--list`	List available models without downloading
`--workers=<N>`	Parallel download threads (default: 4)
`--category=<name>`	Download models of a specific category only
`--models <m1> [m2...]`	Download specific models by name, `.dxnn` file name or registry model name (also `--models=<m1>,<m2>`)
`--demo-models`	Download only the models used by `run_demo.sh` / `run_demo.bat`
`--no-json`	Skip JSON metadata file downloads
`--manifest=<path>`	Use an alternate manifest JSON file
`--force`	Force overwrite if files already exist (default)
`--no-force`	Skip download if the file already exists
`--force-remove-models`	Re-download the models even with `--no-force` (existing files are overwritten; nothing is deleted)
`--force-remove-videos`	Re-download the sample videos even with `--no-force`
`--docker_volume_path=<path>`	Docker volume path for the assets; required when `setup.sh` runs inside a container (it stops otherwise)
`--verbose`	Enable verbose logging
`--help`	Show the help message

© Copyright 2026 DEEPX All Rights Reserved.

For most users, running only `./setup.sh` is sufficient. Contributor-facing setup details are documented separately in the developer guides.

Post-Processing Unit (PPU) Acceleration Integration

DX-APP utilizes PPU Acceleration to maximize inference efficiency on NPU hardware.

The PPU is engineered to offload computationally intensive post-processing tasks, specifically bounding box decoding and score thresholding, directly to the NPU. This critical architectural shift mitigates the CPU overhead traditionally incurred during post-processing, leading to a substantial enhancement in overall inference throughput.

Key Operational Benefits

- **Improved Processing Speed:** Achieved by enabling the parallel execution of both the core inference and the post-processing operations.
- **Enhanced Throughput:** Provides a significant advantage for real-time applications that require sustained high frame rates.

PPU-Enabled Models

DX-APP registers 14 PPU-accelerated object-detection variants. `SCRFD500M_PPU` and `YOLOV5Pose_PPU` example sources stay on disk and are not registered or built. To run PPU models, use `./run_demo.sh` and select "PPU Pipeline" (YOLOv7-PPU), or filter the example runner / DX Model Tool by name. The PPU variants are in the `object_detection` category, and all except `yolov7_640x640_ppu` need `./build.sh --all` (or `./build.sh --target <variant>`):

```
scripts/run_examples.sh --lang cpp --filter _ppu
./scripts/dx_tool.sh run --lang cpp --filter _ppu
```

Available PPU models: YOLOv3-tiny, YOLOv3 (416 and 608), YOLOv4, YOLOv5-S, YOLOv7, YOLOv7-X, YOLOv8-N, YOLOv8-S, YOLOv9-T, YOLOv10-N, YOLO11-N, YOLO12-N, YOLOX-S.

Step 4. Resolve Shared Library Errors

If you encounter shared library errors (e.g., `libdxrt.so`), update the system's library cache.

```
# Copy your library to /usr/local/lib
sudo cp your_library.so /usr/local/lib

# Update the system's library cache
sudo ldconfig
```

3.2.4 Downloads behind a TLS-inspecting proxy

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Symptom. `setup.sh`, or the automatic model download of an example runner, fails behind a corporate proxy that inspects TLS with:

```
certificate verify failed: Missing Authority Key Identifier
```

Cause. Python 3.13 and later (with urllib3 2.3 or later) verify certificates in X.509 strict mode (`VERIFY_X509_STRICT`). Many proxy CAs do not meet it: they lack an Authority Key Identifier, for example. A conda `python3` first on `PATH` is the typical case (Python 3.14 with OpenSSL 3.5). The system Python 3.12 of Ubuntu 24.04, and the dx-runtime virtual environment built from it, are not affected.

Which Python runs the downloader. `setup_sample_models.sh` picks the first of these:

1. `DXAPP_SETUP_PYTHON` (a path or a command name; one that is not an executable exits 1);
2. the active virtual environment (`VIRTUAL_ENV`);
3. `../venv-dx-runtime`;
4. `python3`.

The script prints its choice:

```
[DXAPP] [INFO] Downloader Python: /usr/bin/python3 (Python 3.12.3, OpenSSL 3.0.13 30 Jan 2024)
```

The ways past it, in the order to try them.

1. Run the downloader with Python 3.12 or older: activate the dx-runtime virtual environment, or pass `DXAPP_SETUP_PYTHON=<python> ./setup.sh` (for example `/usr/bin/python3` on Ubuntu 24.04).
2. Give the proxy's root CA through `REQUESTS_CA_BUNDLE` (or `CURL_CA_BUNDLE`, `SSL_CERT_FILE`, or the OS trust store). This fixes a *missing* root. It does not change strict mode.
3. Opt in with `DXAPP_TLS_RELAX_X509_STRICT=1 ./setup.sh`. It turns off only `VERIFY_X509_STRICT`. The certificate is still required, the certificate chain and the host name are still verified, and the protocol floor stays TLS 1.2. It covers the connection to the model server, direct or through a tunnel; the TLS connection to an `https://` proxy itself stays strict. The downloader prints a warning while it is on:

```
[DXAPP] [WARN]  $DXAPP_TLS_RELAX_X509_STRICT=1: X.509 strict mode is off for these
downloads; the certificate chain and the host name are still verified
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

Verification is never turned off. After a failure on a strict-mode-only check, the downloader prints this hint (`x509_strict_hint` in `scripts/download_models.py`):

> TLS verification failed a strict-mode check (Missing Authority Key Identifier): <python> (Python 3.14.6, OpenSSL 3.5.7 9 Jun 2026) verifies in X.509 strict mode, and the certificate chain presented here - typically a TLS-inspecting proxy's - does not meet it.
> Either run the downloader with a Python 3.12 or older: activate a Python 3.12-or-older virtualenv, or DXAPP_SETUP_PYTHON=<python> ./setup.sh (e.g. /usr/bin/python3 on Ubuntu 24.04);
> or opt in with DXAPP_TLS_RELAX_X509_STRICT=1 ./setup.sh, which turns off only X.509 strict mode: the certificate chain and the host name are still verified.

`python3 scripts/download_models.py --help` documents the variable too.

3.2.5 Cross-compiling for aarch64

`./build.sh --arch aarch64` configures with `cmake/toolchain.aarch64.cmake` and builds into `build_aarch64/` . The default (`--minimal`), `--target` and `--category` cross builds leave the executables in the `build_aarch64/` build tree; `./build.sh --arch aarch64 --all` installs into

ⓒ Copyright 2026 DEEPX All Rights Reserved.

`build_aarch64/release`. A cross build never writes the repository's `bin/`, `lib/` or `include/`, and `./build.sh --clean` leaves them alone too.

- **dx_graph is skipped.** The Python module would be built against the host's Python, so the configure step says:

  ```
  dx_graph Python module skipped: cross build (host x86_64, target aarch64): the module would be built against the host's Python. To build it for the target, pass -DDXAPP_CROSS_PYTHON_GRAPH=ON and -DPython_INCLUDE_DIR=<the target Python's include/python3.X> (not validated here: no target Python on the development host)
  ```

  The opt-in is `-DDXAPP_CROSS_PYTHON_GRAPH=ON -DPython_INCLUDE_DIR=<target include/python3.X>`. Both are needed. This path is not covered by the release tests.
- **dx_postprocess skipped.** `pip` would build it for the host and install it into the host's interpreter, so `build.sh` prints `Cross build (aarch64): dx_postprocess skipped - build it on the target with ./build.sh there.` and installs nothing with `pip` (with `--clean` it still uninstalls `dx_postprocess` from the host's `python3`).
- **Compile check.** `bash scripts/check_cross_compile.sh` compile-checks the graph engine and the graph CLI for aarch64 (`-fsyntax-only`, C++14, the graph targets' `-Werror` flags). `--with-registry` adds the generated registry sources (about 100 s). It needs `aarch64-linux-gnu-g++` (`DXAPP_CROSS_CXX`), the dxrt headers and the OpenCV headers; it prints `check_cross_compile OK: <n> sources compile for aarch64 (aarch64-linux-gnu-g++, -fsyntax-only)`. Nothing is linked or run on aarch64.
- **Validation scope.** Concurrency (the graph engine's pipelining, `--max-inflight`, default 16) is validated on x86_64 with a DX-M1. aarch64 is compile-checked only.

3.3 Installation on Windows

This section details the software requirements and sequential installation steps necessary for setting up the DX-APP environment on Windows systems.

- **Stage 1: Prerequisites** - Verify system requirements
- **Stage 2: Core Runtime** - Install the official DX-RT and M1 Driver
- **Stage 3: Toolchain** - Install Visual Studio 2022 (C++ compiler/build environment)
- **Stage 4: Build & Install** - Compile DX-APP source code using build.bat or the VS IDE

3.3.1 Software Requirements on Windows

© Copyright 2026 DEEPX All Rights Reserved.

To run **DX-APP** on Windows, the following components **must** be installed.

- **OS**: Windows 10 22H2 (build 19045) or later (x64)
- **DX-RT**: v3.5.0 or higher (needed for the v9 `.dxnn` files of Model Zoo 2_5_0; see the note under *Software Requirements on Linux*)
- **Python**: Version 3.8 or higher (required: `build.bat` and the CMake configure step run Python scripts; also used by the Python examples)
- **Compiler**: Visual Studio Community 2022 (required for building C++ examples)

3.3.2 Install DX-RT and M1 Windows Driver

DEEPX provides an official Windows installer for **DXNN Runtime (DX-RT)**, which includes the required runtime libraries and M1 device driver.

Prerequisite checklist (DX-RT Windows Driver)

- Microsoft Visual C++ 2015-2022 Redistributable (x64)
- DEEPX NPU device (e.g., DX-M1) connected via PCIe slot, M.2 slot, or USB 4.0 (USB4 PCIe tunneling required)
- Administrator privileges for driver installation

Visual Studio Community 2022 is the build toolchain (IDE + compiler), while the Microsoft Visual C++ 2015-2022 Redistributable provides the runtime DLLs needed to run the built apps.

For detailed instructions, refer to DEEPX NPU Windows Runtime & Driver.

3.3.3 Install Visual Studio Community 2022

To use **DX-APP** on Windows, Visual Studio Community 2022 **must** be installed with appropriate development tools.

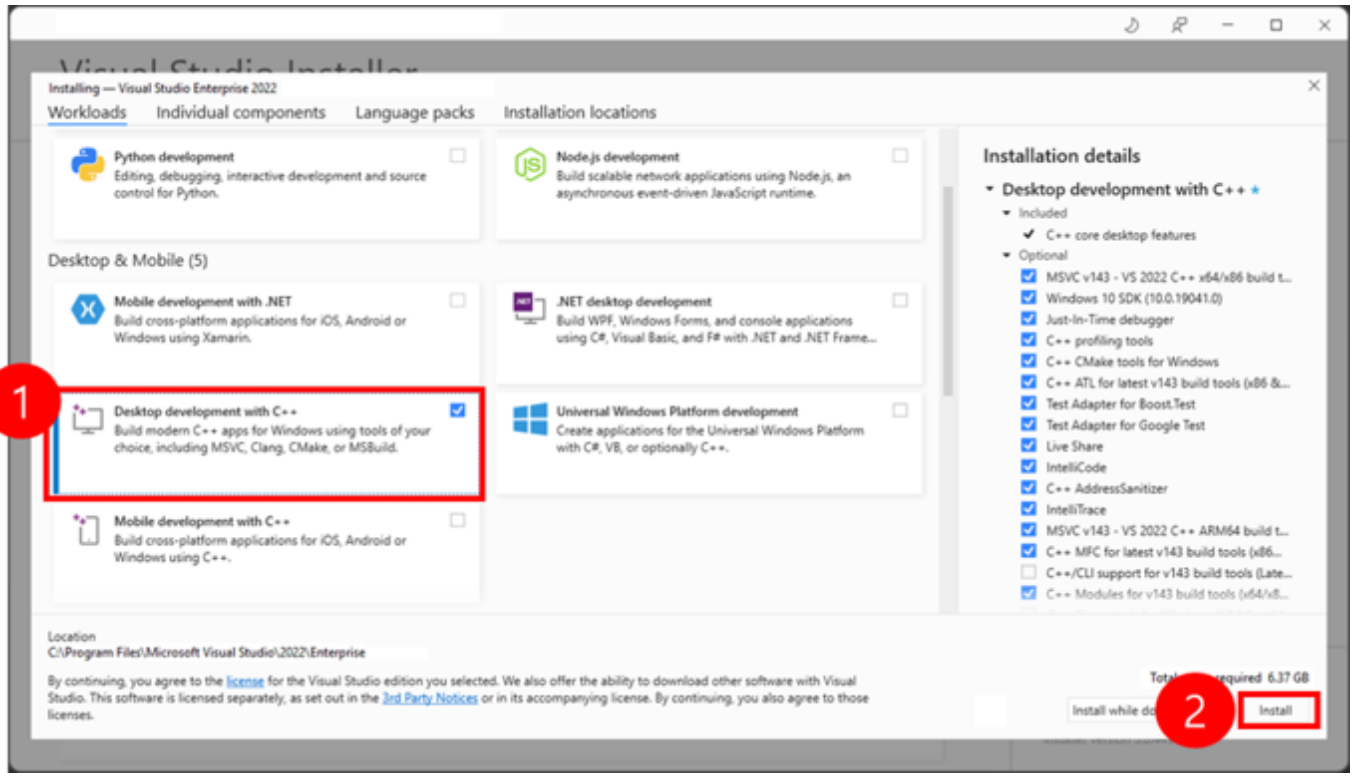
Installation Step

- **Step 1.** Download Visual Studio Community 2022
- **Step 2.** Launch the installer and select the following workload

  Desktop development with C++

- **Step 3.** (Optional) Select additional workloads or individual components as needed
- **Step 4.** Click **Install** to begin the installation process

© Copyright 2026 DEEPX All Rights Reserved.



> **NOTE**
>
> Visual Studio Community 2022 is required; other versions are not tested.

3.3.4 Install VCPKG

VCPKG
is a C++ package manager used for handling third-party dependencies like OpenCV.

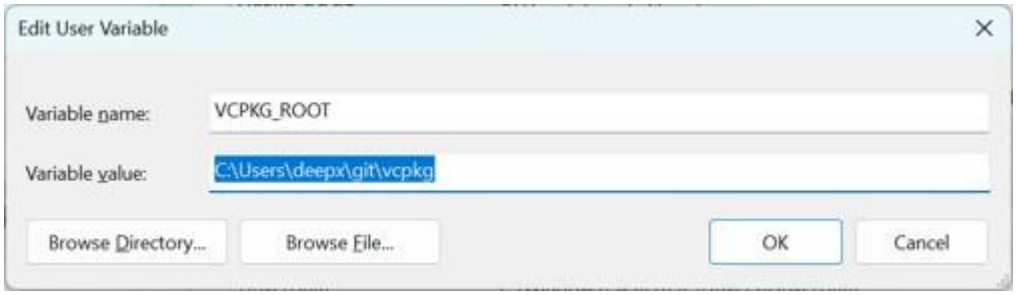
> **NOTE**
>
> If you are using Visual Studio Community 2022, **VCPKG is pre-installed**, so no separate installation is necessary.

If manual installation is required, follow the steps below.

- **Step 1.** Download the vcpkg package from GitHub
- **Step 2.** Open **Command Prompt** and Run the following command
- **Step 3.** Set the user variables
  : Variable Name: `VCPKG_ROOT`
  : Variable Value: Path to your vcpkg installation directory

© Copyright 2026 DEEPX All Rights Reserved.

> **NOTE**
>
> This step is essential to allow Visual Studio to automatically detect and use VCPKG-managed packages like OpenCV.



3.3.5 Build and Install dx_app in Visual Studio Community 2022

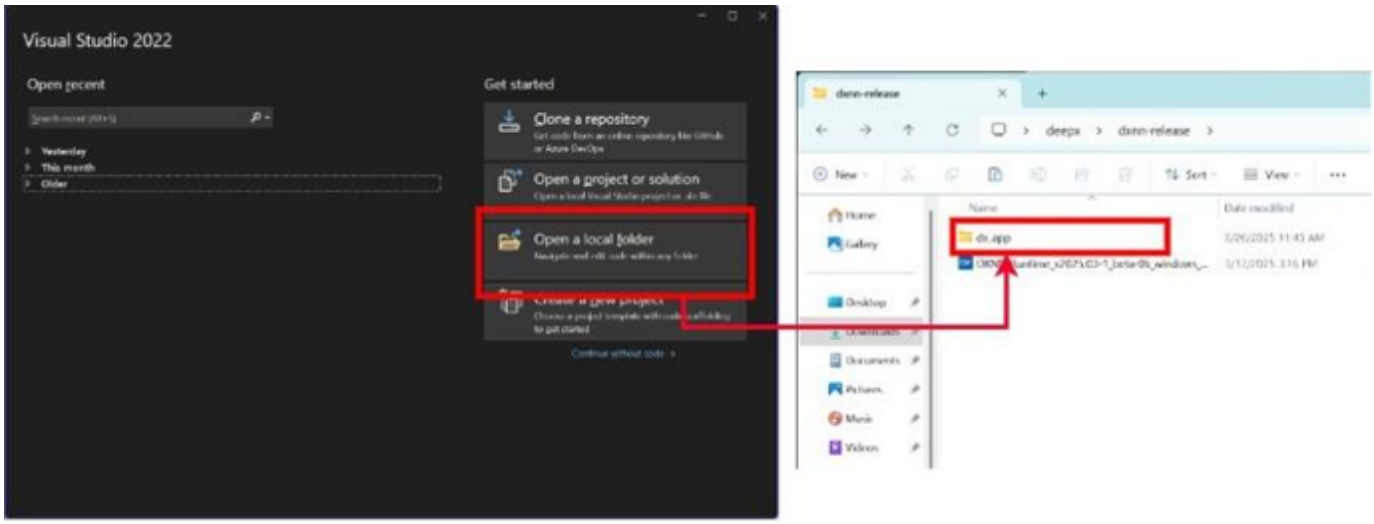
To build and run the `dx_app` application on Windows, follow the steps below using Visual Studio Community 2022.

Step 1. Open Project Folder

- **Step 1-1.** Launch Visual Studio Community 2022
- **Step 1-2.** From the start screen, select **Open a local folder**
- **Step 1-3.** Navigate to and select the `dx_app` project folder

> **IMPORTANT**
>
> You **must** use Visual Studio 2022. Support for other versions (VS 2019, VS Code, etc.) has not been tested and compatibility cannot be guaranteed.

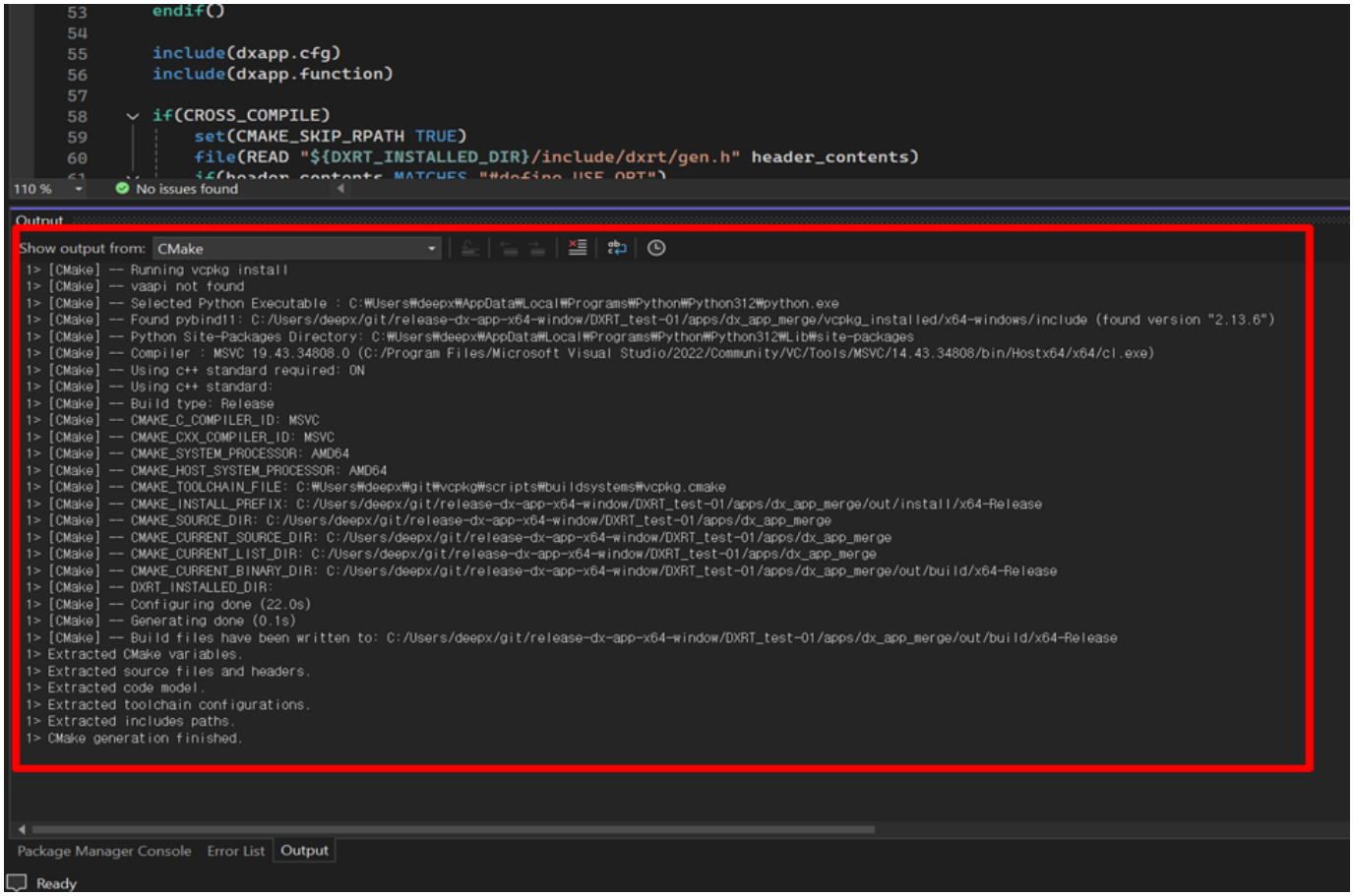
© Copyright 2026 DEEPX All Rights Reserved.



Step 2. Project Configuration

Upon opening the project,

- Dependencies specified in `vcpkg.json` will be automatically downloaded and installed into the `vcpkg_installed` directory.
- CMake will automatically generate the build cache and configuration.



© Copyright 2026 DEEPX All Rights Reserved.

Step 3. (Optional) Edit CMakeSettings

If needed, you can manually specify the following environment variables in `CMakeSettings.json`.

- `DEEPX_SDK_DIR` : Path to the installed DEEPX SDK (DX-RT runtime)
- `OpenCV_DIR` : Path to the OpenCV installation (if manually installed)

```
{
  "name": "CMAKE_TOOLCHAIN_FILE",
  "value": "${env.VCPKG_ROOT}/scripts/buildsystems/vcpkg.cmake",
  "type": "STRING"
},
{
  "name": "DEEPX_SDK_DIR",
  "value": "path/to/dx_rt/installed",
  "type": "STRING"
},
{
  "name": "OpenCV_DIR",
  "value": "${projectDir}/vcpkg_installed/x64-windows/share/opencv",
  "type": "STRING"
}
```

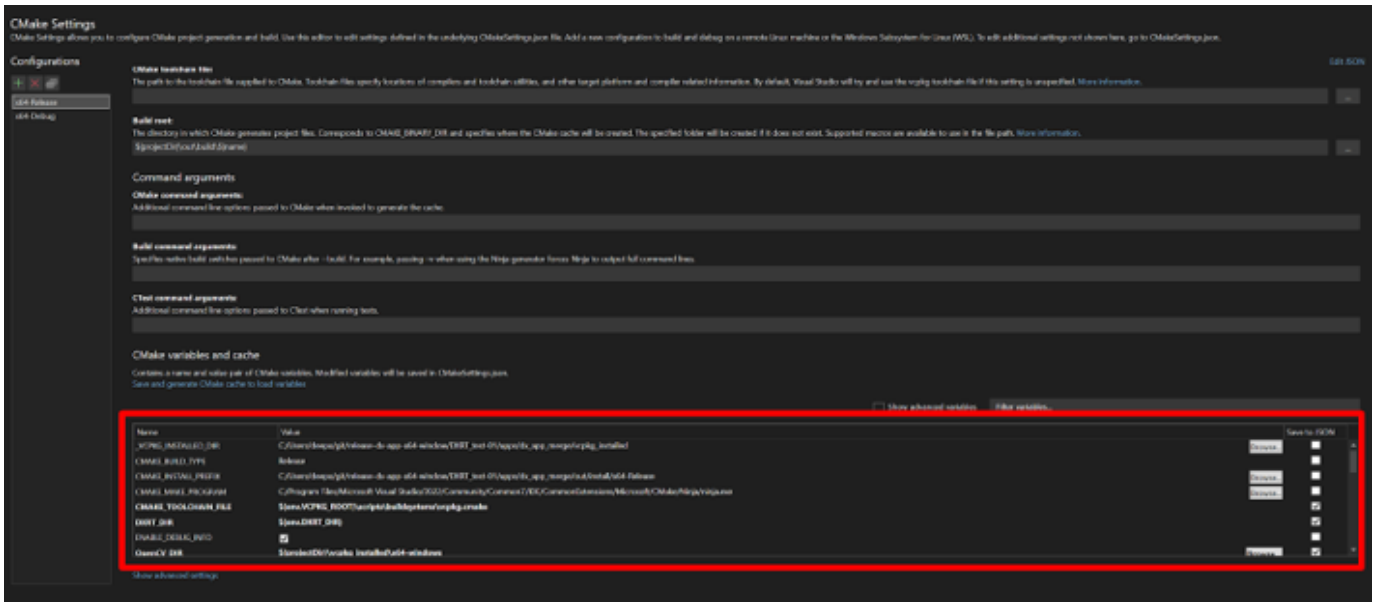
Step 4. Register `PATH` Variable

Ensure the required runtime libraries are accessible with the system's `PATH` environment variable.

- `DEEPX_SDK_DIR` is referenced in CMAKE as `${env.DEEPX_SDK_DIR}`.

> **NOTE**
>
> If you are using `dx_app/vcpkg.json`, OpenCV will be automatically downloaded and installed into `vcpkg_installed/x64-windows` directory during CMake configuration step.

© Copyright 2026 DEEPX All Rights Reserved.



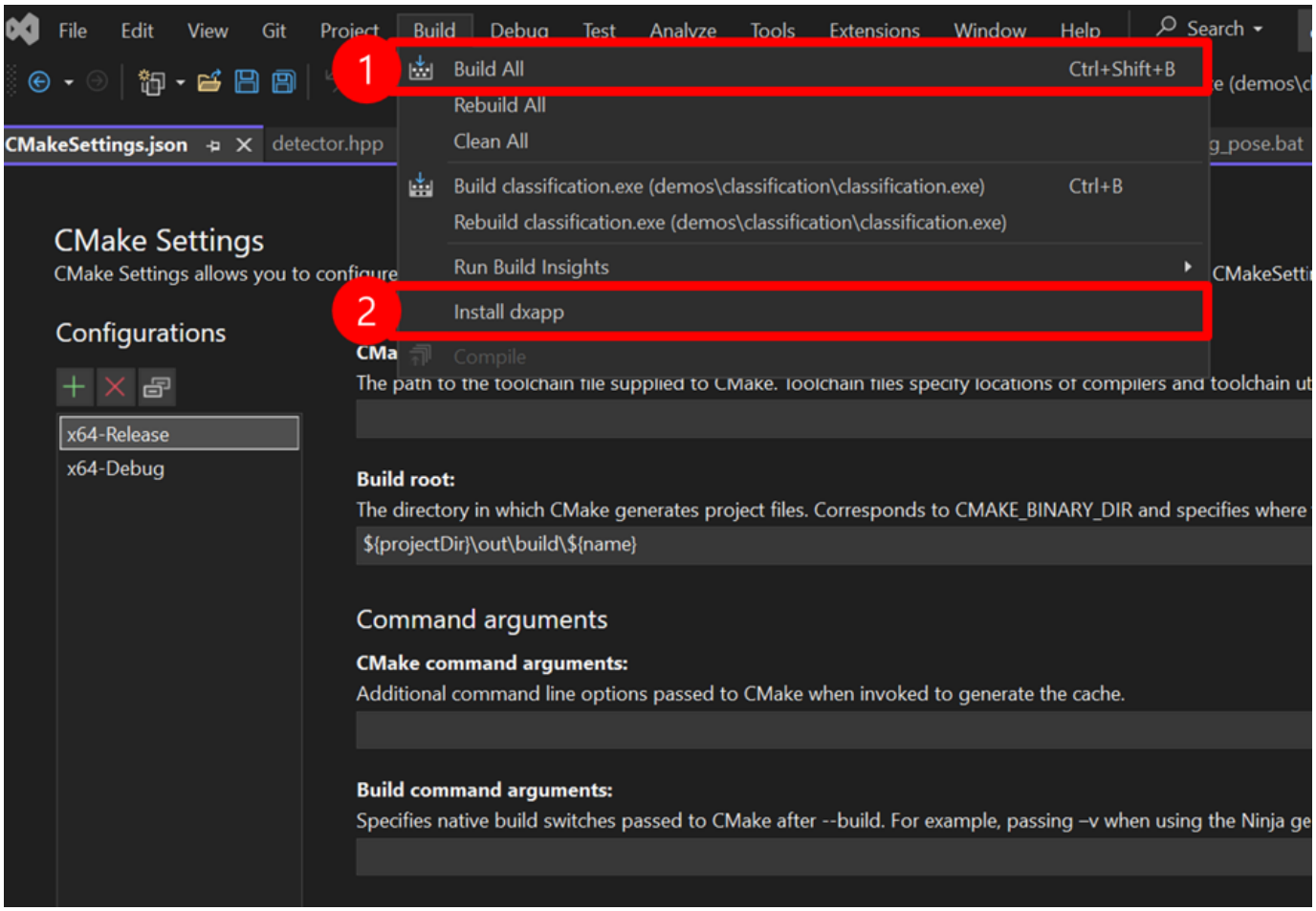
Step 5. Build and Install `dx_app`

To build and install the dx_app application

- **Step 5-1.** Go to the **Build** menu
- **Step 5-2.** Click **Build All** (or **Rebuild All**) to begin the build process

Upon successful compilation, the application executable will be generated under the `bin/` directory.

ⓒ Copyright 2026 DEEPX All Rights Reserved.



3.3.6 Alternative Build Method: Using build.bat

The updated `build.bat` now generates `build_internal.bat` from `CMakeSettings.json` and drives both the DX-APP build and the pybind C++ module build.

Key behaviors

- Generates `build_internal.bat` based on the selected CMake configuration (toolchain, paths, generator).
- Validates environment (e.g., `DEEPX_SDK_DIR`) and cleans stale CMake cache to avoid generator/toolset mismatches.
- Builds and installs DX-APP executables/libraries, then builds the pybind C++ module.

Prerequisites

- `DEEPX_SDK_DIR` set to the DEEPX SDK installation directory
- Visual Studio 2022 with Desktop development with C++ workload
- CMake available in `PATH`

© Copyright 2026 DEEPX All Rights Reserved.

Usage

From the project root

```
REM Asks for the build scope (Enter = minimal)
build.bat

REM run_demo C++ examples only
build.bat --minimal

REM Every example, installed to out\install\x64-Release and copied to bin\
build.bat --all

REM C++ examples of one category ("list" shows the categories)
build.bat --category <name>
```

`build_internal.bat` is written alongside, then executed to configure, build, and install outputs. Use the generated `dxapp.sln` (under the build directory, `./out/build/x64-Release/`) if you want to open in Visual Studio 2022 for further development.

Visual Studio solution Generation

After successful execution of build.bat, the script generates the necessary solution files for development within the IDE.

- **[Important] Open Solution:** Open the generated solution file at `out\build\x64-Release\dxapp.sln` **using Visual Studio 2022**
  NOTE. Opening with other versions may cause compatibility issues or build failures.
- **Access & Customization**: Use Visual Studio 2022 for debugging, development, and further customization. All project targets and configurations are accessible through the VS 2022 interface.

Build Output Structure

```
dx_app/
├── out/                # Build directory created by build.bat
│   ├── build/          # CMake build files
│   │   ├── x64-Release/   # Release build configuration
│   │   │   ├── dxapp.sln  # Visual Studio 2022 solution file
│   │   │   ├── *.vcxproj  # Project files for each target
│   │   │   └── ...        # Other build artifacts
│   │   └── ...
│   └── install/          # Installation directory
└── bin/                # Executables copied by build.bat, plus the DX-RT and vcpkg DLLs
```

3.3.7 Graph engine on Windows

ⓒ Copyright 2026 DEEPX All Rights Reserved.

The graph engine, the `multi_model_graph_{sync,async}` CLI and the generated model registry are **not tested on Windows in this release**. They are still part of the Windows CMake project: `build.bat --all` and **Build All** compile them (with `graph_engine_test`), and every configure generates the registry; only `dx_graph` is skipped. The Windows support below is prepared and verified on Linux only:

- **MSVC stays C++17.** Under `/std:c++14` the tree's own filesystem switch picks `<experimental/filesystem>`, which current MSVC STL rejects, and the Windows build has only ever been C++17. C++14 is enforced where it can be checked: GCC/Clang `-Werror=c++17-extensions` and `scripts/check_cross_compile.sh`.
- **Graph targets get `/bigobj` and two warnings as errors.** `/bigobj` (a generated registry source holds up to 64 factories), `/w15038 /we5038` (C5038, members initialized out of order) and, for targets with a `switch` over an enum, `/w14062 /we4062` (C4062, an enumerator not handled). Tests check that the flags are emitted, not that MSVC accepts them.
- **Python 3 is needed to configure.** The configure step runs `scripts/gen_model_registry.py`. Without a Python 3 it stops with `No Python 3 interpreter for the graph model registry` and names the fix: install Python 3 (the Microsoft Store stub does not work) or pass `-DPython3_EXECUTABLE=<path to python3>`.
- **Ctrl-C** goes through `SetConsoleCtrlHandler`: the first press asks for a graceful stop, a repeat within 200 ms counts as the same request, a later press ends the process with `STATUS_CONTROL_C_EXIT`. **Ctrl-Break** ends it at once, as before. It is tested on Linux against a stub `<windows.h>`.
- **Tests find the binaries** in `bin\Release` (or `bin\RelWithDebInfo`, `bin\Debug`), else `bin\`, with `.exe` names.
- **Smoke tests.** `tests\windows\run_tests.bat graph` runs the graph CLI smoke tests (`--help`, `--list-models`, `--check`; no NPU).
- **dx_graph has no Windows build.** The CMake decision function skips it for MSVC.

3.3.8 Run Example Executable Files On Windows

After building and installing dx_app, you can execute the demo applications using provided batch scripts.

Step 1. Execute `setup.bat`

Run the `setup.bat` script to download models and sample videos. Without options, a menu lets you select categories and models. Use `setup.bat --all` to download everything, or `setup.bat --demo-models` to download only the demo models.

- The downloaded assets will be placed in the `assets` folder.

© Copyright 2026 DEEPX All Rights Reserved.

Step 2. Run Examples

You can run the examples using the same command line instructions as in Linux, but using the `.exe` extension for executables.

classification example (not in the minimal build: build it with `build.bat --all` or `build.bat --category image_classification`)

```
./bin/efficientnet-lite0_224x224_async.exe -m ./assets/models/efficientnet-lite0_224x224.dxnn -i ./sample/ILSVRC2012/0.jpeg
```

object detection example

```
./bin/yolov7_640x640_sync.exe -m ./assets/models/yolov7_640x640.dxnn -i ./sample/img/sample_kitchen.jpg -l 10
```


© Copyright 2026 DEEPX All Rights Reserved.

4. DX-APP C++ Example

4.1 DX-APP C++ Usage Guide

This guide explains how to navigate and use the C++ example tree in DX-APP.

4.1.1 Overview

The C++ examples are located under `src/cpp_example/` and are organized by:

- **task**
- **model family**
- **variant** (the `.dxnn` stem; sync and async are separate entry files inside that folder)

All examples share a common runtime layer under `src/cpp_example/common/` that provides base interfaces, processors, runners, input sources, visualizers, and utilities. This is the C++ counterpart of `src/python_example/common/` — both languages implement the same factory-based architecture with the same core modules. Each variant directory contains thin entry-point source files and a factory that wires shared components together.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Representative task directories include:

- `image_classification/`
- `object_detection/` — PPU models are a family here, for example `yolo_ppu/`
- `face_detection/`
- `pose_estimation/`
- `semantic_segmentation/`
- `instance_segmentation/`
- `depth_estimation/`
- `face_recognition/`
- `image_denoising/`
- `low_light_enhancement/`
- `super_resolution/`
- `hand_landmark/`
- `oriented_object_detection/`
- `person_attribute/`
- `person_reid/`
- `face_landmark/`

For the full repository-level structure, refer to DX-APP Example Source Structure.

4.1.2 Architecture & Design Pattern

Architecture Strategy

Shared Runtime Layer (`common/`)

The `common/` directory is the engine behind all C++ examples:

Module	Contents	Role
`common/base/`	4 interface headers (.hpp)	`IFactory` , `IPreprocessor` / `IPostprocessor` , `IVisualizer` , `IInputSource`
`common/config/`	`model_config.hpp`	Loads `config.json` (input size, labels, thresholds)

© Copyright 2026 DEEPX All Rights Reserved.

Module	Contents	Role
`common/processors/`	Postprocessor / preprocessor headers	Shared decode logic for all model families
`common/runner/`	Runner headers	A dedicated sync/async runner pair per task type
`common/inputs/`	5 source headers	Image, Video, Camera, RTSP input abstraction
`common/visualizers/`	Visualizer headers	Task-specific result rendering
`common/utility/`	Utility headers	Labels, preprocessing, profiling, run_dir, colorspace, verify_serialize

Unlike Python's generic `SyncRunner` / `AsyncRunner`, C++ runners are **task-specific**: each task type has a dedicated sync/async pair (e.g., `sync_detection_runner.hpp`, `async_detection_runner.hpp`).

Factory Pattern Implementation

Each variant directory has a `factory/<variant>_factory.hpp` that implements the task factory:

The entry point delegates to a task-specific runner:

```
// object_detection/yolov9/yolov9-s_640x640/yolov9-s_640x640_sync.cpp
auto factory = std::make_unique<dxapp::v_yolov9_s_640x640::Yolov9Factory>();
dxapp::SyncDetectionRunner<dxapp::v_yolov9_s_640x640::Yolov9Factory> runner(std::move(facto
ry));
return runner.run(argc, argv);
```

Directory Pattern & File Pattern

Each task directory contains model families. Each variant directory is named after the `.dxnn` stem and holds that variant's factory, `config.json`, and sync/async entries.

Example:

```
src/cpp_example/object_detection/yolov9/
└── yolov9-s_640x640/
    ├── factory/
    │   └── yolov9-s_640x640_factory.hpp
    ├── yolov9-s_640x640_sync.cpp
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    ├── yolov9-s_640x640_async.cpp
    └── config.json
```

The CMake target and `bin/` executable use the same stem: `yolov9-s_640x640_sync`.

Common files:

- `<task>/<family>/<variant>/config.json`: that variant's runtime settings
- `<variant>/factory/<variant>_factory.hpp`: factory header wiring shared `common/` components
- `<variant>_sync.cpp`: synchronous entry for that variant
- `<variant>_async.cpp`: asynchronous entry for that variant

4.1.3 Execution Framework

Execution Variants

Synchronous Flow (`xx_sync.cpp`)

Use this variant when you want:

- simpler control flow
- easier step-by-step debugging
- single-image or low-complexity usage examples

Asynchronous Flow (`xx_async.cpp`)

Use this variant when you want:

- higher throughput
- better overlap of pipeline stages
- real-time image/video processing patterns

CLI Interface

All C++ examples use `cxxopts` for argument parsing and share a consistent interface:

Flag	Short	Type	Description
`--model_path`	`-m`	string	Path to `.dxnn` model file. Optional: if omitted, the example's

© Copyright 2026 DEEPX All Rights Reserved.

Flag	Short	Type	Description
			default model is used and auto-downloaded when missing
`--image_path`	`-i`	string	Input image file or directory
`--video_path`	`-v`	string	Input video file
`--camera_index`	`-c`	int	Camera device index
`--rtsp_url`	`-r`	string	RTSP stream URL
`--save`	`-s`	bool	Save rendered output to a run directory
`--save-dir`	—	string	Base output directory (default: `artifacts/cpp_example`)
`--dump-tensors`	—	bool	Dump input/output tensors to `.bin` files
`--loop`	`-l`	int	Inference repeat count (default: auto)
`--no-display`	—	bool	Disable visualization window, output FPS only
`--drop-frames`	—	bool	Preview drops stale frames instead of pacing the pipeline (default: off, every frame is shown)
`--show-log`	—	bool	Enable verbose log output (default: quiet)
`--config`	—	string	Model config JSON path. If omitted, the factory's built-in defaults are used; the

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Flag	Short	Type	Description
			variant's `config.json` is not read automatically
`--help`	`-h`	—	Show usage

- **Input source:** `--image_path`, `--video_path`, `--camera_index`, and `--rtsp_url` form a mutually exclusive group. If none is specified, a **default sample image** is automatically selected based on the task type.
- **Raspberry Pi camera:** on a Raspberry Pi, `--camera_index N` opens the CSI camera through a libcamera GStreamer pipeline (`libcamerasrc`) and camera `N` is camera `N` of `rpicam-hello --list-cameras`; a USB webcam node keeps the V4L2 path. `install.sh` installs `gstreamer1.0-libcamera` when it detects a Pi camera.
- **Windows camera and video files:** the examples open cameras (`--camera_index N`) and video files (`--video_path`) with OpenCV's MSMF hardware transforms off (`OPENCV_VIDEOIO_MSMF_ENABLE_HW_TRANSFORMS=0`), set when the program starts: with them on, a USB webcam takes about 7 s to open (off, about 0.15 s at the same frame rate), and the OpenCV build used here has no FFmpeg backend, so a video file is decoded through MSMF, where hardware transforms can make a file that decodes fine without them read no frames. To keep them on, set `OPENCV_VIDEOIO_MSMF_ENABLE_HW_TRANSFORMS=1` before running; a value you set is not changed.

NOTE

Image-only tasks: 11 tasks accept `--image_path` input only; `--video_path`, `--camera_index` and `--rtsp_url` are rejected at runtime with an error. The list is `isComparisonOnlyTask()` in `src/cpp_example/common/utility/common_util.hpp`, plus the `object_pose_estimation` and `3d_object_detection` checks in the pose and 3D detection runners under `src/cpp_example/common/runner/`.

`face_recognition`, `face_attribute`, `person_attribute`, `person_reid`, `image_retrieval`, `visual_place_recognition`, and the embedding-model variants under `zero_shot_image_classification` (CLIP image encoders) and `image_classification/casvit` need a crop of a pre-detected subject, or a committed gallery to rank against. Running a single embedding model on a raw stream without a preceding detector would not produce valid results.

`object_pose_estimation`, `3d_object_detection` do not take a video frame at all: `3d_object_detection` (SFA3D) consumes a LiDAR point cloud (`sample/kitti/velodyne/*.bin`) and refuses any other file type, and DOPE scores the pose of a static object from a single image.

© Copyright 2026 DEEPX All Rights Reserved.

4.1.4 Getting Started

To set up, build and run a first C++ example, follow DX-APP Quick Start. The default `./build.sh` builds only the examples that `run_demo.sh` uses; build any other example by its target names, for example:

```
./build.sh --target yolov9-s_640x640_sync yolov9-s_640x640_async
```

4.1.5 Advanced Operations & Debugging

Runtime Features

Auto-Download

When `-m` / `--model_path` is omitted, the runner uses this example's default model from `config/model_registry.json` and, if the file is missing, downloads it via `setup_sample_models.sh`. If the download fails, an error message with manual download instructions is displayed. An explicit `-m` path that does not exist is an error (no download). Missing input files, including videos, are not downloaded; run `./setup.sh` or `./setup_sample_videos.sh` first.

Default Input Fallback

If no input source is provided, the runner automatically selects a default sample image appropriate for the task type (e.g., `sample/img/sample_street.jpg` for object detection). A log message indicates which default was applied.

Signal Handling

All C++ binaries - every single-model runner (`installSignalHandlers()`) and `multi_model_graph_sync` / `multi_model_graph_async` - install SIGINT/SIGTERM handlers. Pressing Ctrl+C triggers a graceful shutdown with clean resource release (the graph CLI finishes the frames in flight in every stream and finalizes `--report` / `--output` / `--events`). A later Ctrl+C, more than 200 ms after the first, terminates the process at once (for a shutdown that is itself stuck); repeats within 200 ms count as the same request. SIGTERM is always graceful, however often it arrives.

Preview window. The window opens at half the screen size with the frame's aspect ratio kept (`DXAPP_SCREEN_W` / `DXAPP_SCREEN_H`, else `xdpyinfo`, else 1920x1080), the same rule as the Python runners' `show_output()` on Linux (on Windows the C++ runners use 1920x1080 unless `DXAPP_SCREEN_W` / `DXAPP_SCREEN_H` is set). Every frame is shown by

© Copyright 2026 DEEPX All Rights Reserved.

default: a window slower than the NPU paces the pipeline, so measure throughput with `--no-display`. `--drop-frames` selects a depth-1 preview that discards stale frames (the sync preview is then capped at 10 fps, the async one at 60 fps, and the async summary prints `dropped (stale)`).

Output order (async runners)

Async runners get results back from dxrt's completion threads in any order. Each frame gets an index when it is submitted, and the display thread releases frames through a reorder buffer (`common/utility/frame_reorder.hpp`), so display, save and `DXAPP_VERIFY` output (written where the buffer releases a frame) follow the input order and equal the sync runner's frame by frame. The runner's one postprocessor is shared by dxrt's callback threads and serialized (`common/processors/serialized_postprocessor.hpp`). SuperPoint's tracker is updated by its visualizer, after the reorder buffer, so it also sees frames in input order.

- At the end of a video the runner waits until every submitted frame has reached the display thread and, with `--save`, has been written. If nothing progresses for 5 s it prints `[DXAPP] [WARN] Output frames stopped draining; saved video may be truncated.` (without `--save`: `... the last frames may be missing from the display and DXAPP_VERIFY records.`) and stops.
- On an interrupted run (Ctrl+C, SIGTERM, or closing the window) that wait ends at once, and the frames the reorder buffer already holds are still released in index order.
- The reorder buffer holds at most twice the runner's in-flight depth. If a frame never comes back from dxrt, the lowest buffered frame is released once that cap is passed, so the display cannot stall; from then on output order is no longer guaranteed.

Output Management (`--save`)

When `--save` is enabled, a timestamped directory is created (e.g., `artifacts/cpp_example/{model}_sync-image-{name}-{timestamp}/`, `_async` for async binaries) containing `run_info.txt`, saved images/video, and optional tensor dumps. A video is saved as `output.mp4` (H.264, else MPEG-4); a PC with neither encoder, such as Windows N/KN editions, gets `output.avi` (MJPG) instead.

Configuration Management (`--config`)

Runtime parameters (thresholds, top-k, etc.) live in `<task>/<family>/<variant>/config.json`, next to that variant's entry. A single-model extract may keep `config.json` beside the entry.

Verification & Diagnostics

Numerical Verification (`DXAPP_VERIFY`)

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Set `DXAPP_VERIFY=1` to serialize post-processing results. Each run writes `logs/verify/{model}.json` (override the directory with `DXAPP_VERIFY_DIR`) holding the last frame, and `{model}.frames.jsonl` with one record per frame (`"frame": 0, 1, …` in input order). Writes are serialized and the JSON is replaced atomically, so a reader never sees a half-written file. Every runner dumps its real results, including tiled super-resolution (statistics of the stitched output), 3D detection and YOLOPv2 (boxes plus drivable/lane mask statistics); SuperPoint records carry each frame's keypoints.

Frame numbers count per process, so with `-l N` on a video the records of later loops continue the numbering. `{model}` is the model file's stem (for example `yolopv2_384x640.json` for `yolopv2_384x640.dxnn`).

Tensor Dump for Debugging (`--dump-tensors`)

Dumps raw input/output tensors as `.bin` files. On an exception in a sync runner, the input and output tensors are auto-dumped for debugging.

Environment Variables Reference

Variable	Description
`DXAPP_SAVE_IMAGE`	Save visualization to the specified file path (no `--save` required)
`DXAPP_VERIFY`	When `1`, dump JSON verification data
`DXAPP_VERIFY_DIR`	Directory for `DXAPP_VERIFY` output (default `logs/verify`)
`DXAPP_PROGRESS`	When `1`, print `[PROGRESS] frames=N fps=F avg_fps=A read_ms=… pre_ms=… infer_ms=… post_ms=…` once a second and at exit (front ends); multi_model_graph prints `node_ms=<id>:<ms>,…` instead of the stage fields
`DXAPP_ROOT`	The root that repository-relative paths (`sample/...`, a model's `config.json`, galleries) are read against. Default: the repository the program was built from; an exported graph project uses the folder holding its `.dxapp-root` (its `run.sh` sets this variable)

© Copyright 2026 DEEPX All Rights Reserved.

4.1.6 Exporting a graph as a standalone project

`python3 scripts/export_graph_project.py --graph <graph.json> --out <dir>` writes a multi-model graph as a CMake project that needs only the DEEPX runtime, CMake, a C++17 compiler and OpenCV 4: `./setup.sh` builds it, `./run.sh` runs it and writes `results/`. Each "User logic" node becomes a C++ class stub in `user_logic/` that passes data through until you edit it. See `src/cpp_example/multi_model_graph/README.md`, "Take a graph away".

4.1.7 Supplementary Information

Component Relationships

`src/postprocess/` contains C++ post-processing implementations that are **not** linked by the C++ examples, except for four modules (`sfa3d`, `superpoint`, `dope`, `yolopv2`) whose sources are compiled into them. They are consumed by the pybind11 bindings (`src/bindings/python/dx_postprocess/`) to enable `*_cpp_postprocess.py` variants in Python.

The C++ examples rely on their own shared processors in `src/cpp_example/common/processors/`.

See also: DX-APP C++ Post-processing Overview

Developer Resources

- For contributor workflows, use DX Tool Guide
- For test execution, use DX-APP C++ Example Tests
- For repository layout details, use DX-APP Example Source Structure

© Copyright 2026 DEEPX All Rights Reserved.

4.2 DX-APP C++ Example Tests

4.2.1 Overview

The project provides a comprehensive Python-based test suite located in `tests/cpp_example/`. These tests ensure that C++ binaries are functional, handle arguments correctly, produce correct visualizations, and report performance (FPS per pipeline stage) on the DEEPX NPU.

4.2.2 Test Classification & Prerequisites

Test Categories

The suite is organized into tiered categories based on execution speed and scope.

CLI Tests (Fast)

- **Files**: `test_cli_help.py`, `test_cli_basic.py`
- **Scope**: Validates `--help` output, invalid argument handling, and no-argument behaviors for all executables.

E2E (End-to-End) Tests (Slow)

- **File**: `test_e2e.py`
- **Scope**: Performs real inference on images and videos using `.dxnn` models to verify the full pipeline and NPU utilization.
  : Auto-discovers all (executable, model) pairs from `bin/` and `assets/models/`.
  : Assets: Uses real models from `assets/models/` and test data from `sample/img/` and `assets/videos/`.
  : Parameters: Default loop count configurable via `--loop`.
  : Timeouts: no time limit for image inference except TTA models (300 seconds); models in `E2E_HEAVY_MODELS` (`tests/test_helpers/constants.py`) run at most 2 loops. Video inference: 2000 seconds (about 33 minutes).
  : Video E2E and `--save` video tests run every stream-capable model except the W6 face detectors (`VIDEO_TOO_SLOW_MODELS` in `tests/test_helpers/constants.py`). Super-resolution's stream path runs on a 6-frame low-resolution clip (`test_super_resolution_stream_e2e`).

Specialized Tests

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Visualization Tests

- **File**: `test_visualization.py`
- **Scope**: Runs all sync + async binaries with `DXAPP_SAVE_IMAGE` and verifies image output is produced.
  : Output directory: `tests/test_visualization_result/cpp_example/{sync,async}/<task>/`
  : Can also be run standalone: `python test_visualization.py`

Feature Tests

File	Marker	Scope
`test_save_mode.py`	`save_mode`	`--save` / `--save-dir` output and run directory creation
`test_dump_tensors.py`	`dump_tensors`	`--dump-tensors` tensor file generation
`test_verify.py`	`verify`	`DXAPP_VERIFY` output of every downloaded model, sync and async: one non-empty record, per-frame file
`test_verify_dumps.py`	`verify`	Runtime checks of past `DXAPP_VERIFY` regressions (async classification/detection, restoration and tiled SR, 3D detection, YOLOPv2 masks)
`test_sync_async_parity.py`	`verify`	Async output equals sync frame by frame (video clip, at dxrt's default I/O buffer count): every video-capable async runner, plus loops and an interrupt mid-video. The sweep leaves out the image-only tasks (super_resolution among them) and the registry rows marked `image_only`; `espcn-x4_17x17`, `realesrgan-x2_192x192`, `sfa3d_608x608` and `arcface_mobilefacenet_112x112`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

File	Marker	Scope
		are compared in image mode instead.
test_multi_loop.py	multi_loop	-l N loop count behavior
test_signal_handling.py	signal_handling	SIGINT graceful shutdown; one Ctrl-C under timeout is one request; a later Ctrl-C ends a run stuck in its output (a FIFO)

test_graph_engine.py (marker graph) runs the graph engine's graph_engine_test binary: it fails on a missing summary line, any failure, an OpenCV or GLib log line on stderr, a Config file not found line or any file left in its scratch TMPDIR; skipped hardware cases are reported as a pytest SKIP naming them.

Other test files:

File	Marker	Scope
test_graph_cli.py	graph, graph_e2e, graph_parity	Multi-model graph CLI. graph: no NPU, --check and --list-models. graph_e2e: every sample graph end to end on hardware. graph_parity: sync and async executors give byte-identical --report JSON
test_graph_python.py	graph, graph_e2e, graph_parity	The dx_graph Python binding of the graph engine (skipped when dx_graph is not built)
test_graph_export.py	contract, graph_export, graph_e2e	A graph exported with scripts/export_graph_project.py, built and run outside the repository (graph_export needs no NPU)
test_e2e_camera_rtsp.py	e2e_camera, e2e_rtsp	Camera (-c) and RTSP (-r) input: each executable runs for --stream-duration seconds, then gets SIGINT; passes on

ⓒ Copyright 2026 DEEPX All Rights Reserved.

File	Marker	Scope
		exit 0, frames > 0 and FPS > 0
test_headless_display.py	e2e, e2e_image	An example started without a display server and without --no-display runs headless instead of crashing
test_download_hints.py	contract	Recovery hints name scripts/download_models.py, never ./setup.sh --models
test_common_unit.py	—	Runs the common_unit_test binary (plain-assert tests of common/)
test_harness_helpers.py	cli, help	Test harness helpers: executable discovery, headless display, bounded subprocess runs
test_preprocessor_scale_regression.py	golden, contract	Golden checks of the grayscale resize preprocessor's per-axis scale (DnCNN, ESPCN, SuperPoint keypoints)

Environment Setup

Test Requirements

Before running tests, ensure the environment is prepared.

For CLI Tests

- Executables must be built in the bin/ directory.
- Run ./build.sh --all from the project root to test every executable. The default ./build.sh builds only the run_demo targets, so only those are tested; use --target <name> to build a specific one.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

For E2E Tests (Additional)

- **Models**: Run `./setup_sample_models.sh` to populate `assets/models/`. Pass `--all` to download every model non-interactively.
- **Test Data**: Ensure images exist in `sample/img/` and videos in `assets/videos/` (Run `./setup_sample_videos.sh`).

Available Markers (`pytest.ini`)

`cli`, `help`, `e2e`, `e2e_image`, `e2e_stream`, `e2e_camera`, `e2e_rtsp`, `e2e_short`, `visualization`, `async_exec`, `sync_exec`, `save_mode`, `dump_tensors`, `verify`, `multi_loop`, `signal_handling`, `contract`, `golden`, `graph`, `graph_e2e`, `graph_parity`, `graph_export`

4.2.3 Test Infrastructure

Shared Module (`tests/test_helpers/`)

All test files import shared constants and utilities from `tests/test_helpers/`:

- `constants.py`: `TASK_IMAGE_MAP`, `MODEL_IMAGE_OVERRIDE`, `MULTI_MODEL_EXECUTABLES`, path constants
- `utils.py`: `setup_environment()`, `discover_cpp_executables()`, `strip_variant_suffix()`, `dxnn_for_exe()`
- `platform_paths.py`: per-OS `bin/` directory and `.exe` names
- `proc.py`: subprocess runs with a timeout (SIGTERM first, then SIGKILL)
- `pytest_support.py`: shared pytest options (`--loop`, `--camera-index`, `--rtsp-url`, `--stream-duration`)
- `verify.py`: readers and comparers for `DXAPP_VERIFY` output

Asset Management

Assets required for testing are automatically managed by the setup scripts in the project root:

- **Models**: Use `setup_sample_models.sh` to download models.
- **Videos**: Use `setup_sample_videos.sh` to acquire sample video files required for E2E (End-to-End) pipeline verification.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

4.2.4 Test Execution Guide

Execution Methods

Method A. Unified Test Runner (Recommended)

The `run_tc.sh` script provides a high-level interface for running standardized test suites directly from the project root.

```
cd ../../  # Go to project root (dx_app/)

# Run only C++ tests (CLI + E2E stream)
./run_tc.sh --cpp

# Run only C++ CLI tests (fast)
./run_tc.sh --cpp --cli

# Run only C++ E2E stream tests (all models)
./run_tc.sh --cpp --e2e

# Run only C++ E2E image tests (faster, skips stream)
./run_tc.sh --cpp --e2e-quick

# Run only C++ E2E image + stream tests for representative models (one per task,
`E2E_SHORT_MODELS`)
./run_tc.sh --cpp --e2e-short

# Run C++ tests with code coverage (standalone)
./run_tc.sh --cpp --coverage

# Show all available options
./run_tc.sh --help
```

TIP

Use `--e2e-quick` during development. It takes ~2–3 minutes, compared to 8–10 minutes for a full test. Use `--coverage` for coverage analysis — it cannot be combined with --cli, --e2e, etc.

Method B. Manual Execution via Pytest

For granular control, run `pytest` directly from `tests/cpp_example/`.

Basic Usage

ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
# Install requirements
pip install -r requirements.txt

# Run every test not marked e2e (feature, visualization, verify and graph tests still run real binaries and can take long)
pytest -m "not e2e"

# Run only CLI tests (fast)
pytest -m cli

# Run only E2E tests (slow, requires models and test data)
pytest test_e2e.py -v

# Run all tests including E2E with verbose output
pytest -v -s
```

Advanced Filtering

```
cd tests/cpp_example

# Test specific model (all variants)
pytest -m e2e -k "yolov7"          # All yolov7 variants
pytest -m e2e -k "yolov7_640x640_async"  # Only that async variant

# Test multiple models
pytest -m e2e -k "yolov7 or yolov8"
pytest -m e2e -k "scrfd or yolov5"

# Exclude variants
pytest -m e2e -k "yolov7 and not async"  # Only yolov7 sync variants
pytest -m e2e -k "yolov5 and not ppu"    # yolov5 without ppu variants

# Combine with markers
pytest -m "e2e and async_exec" -k "yolov7"  # Only async yolov7 tests
pytest -m "e2e and sync_exec" -k "yolov5"   # Only sync yolov5 tests

# Run specific test function
pytest test_e2e.py::test_stream_inference_e2e[yolov7_640x640_async]
pytest test_e2e.py::test_image_inference_e2e[scrfd-500m_640x640_async]

# Multiple specific tests
pytest test_e2e.py::test_stream_inference_e2e[yolov7_640x640_async] \
       test_e2e.py::test_stream_inference_e2e[yolov8-n_640x640_async]
```

Result Analysis

Performance Reporting

© Copyright 2026 DEEPX All Rights Reserved.

After running the E2E stream (video) tests, a performance report is automatically generated to provide deep insights into the inference pipeline efficiency.

- **Console Output**: A formatted table with real-time FPS metrics.
- **CSV File**: A detailed log saved as `tests/cpp_example/performance_reports/performance_report_YYYYMMDD_HHMMSS.csv`.

The report includes

- **E2E FPS:** Overall pipeline throughput.
- **Read FPS:** Speed of frame ingestion.
- **Preprocess FPS:** Speed of image transformation (resizing, normalization).
- **Inference FPS:** Pure NPU model execution speed.
- **Postprocess FPS:** Speed of result parsing (NMS, coordinate scaling).
- **Bottleneck Detection:** For async variants, the slowest of the Read / Preprocess / Inference / Postprocess stages is marked with an **asterisk (*)** for quick optimization targeting.

Test Coverage Summary

`./run_tc.sh --cpp --coverage` runs one pytest pass, `-m "not e2e_stream and not visualization and not multi_loop and not signal_handling" --coverage`. It covers the CLI tests, E2E image tests and the save_mode, dump_tensors, verify, contract, golden and graph tests. Stream E2E, visualization, multi-loop and signal-handling tests are not included. Test counts depend on the executables in `bin/` (`./build.sh --all` builds a sync and an async binary per variant, about 1,000) and on the downloaded models.

4.2.5 Advanced Analysis: Code Coverage

Code coverage measures how much of the source code is exercised during testing. This is essential for ensuring the robustness of the NPU inference pipeline.

Coverage Analysis Strategy

To generate code coverage reports, you need to build the project with instrumentation and install the following tools

- **Step 1.** Build executables with coverage instrumentation.
- **Step 2.** Install coverage tools (`gcovr` is recommended; `lcov` is supported as a fallback).

ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
# Install gcovr (recommended - supports XML, HTML, JSON)
sudo apt-get install gcovr -y

# OR install lcov (HTML only)
sudo apt-get install lcov -y

# [Recommended] Build with debug mode for the most reliable coverage results
cd ../../  # Go to project root (dx_app/) from tests/cpp_example/
./build.sh --clean --coverage --type debug --all

# [Alternative] Build with relwithdebinfo for faster builds (less accurate coverage)
./build.sh --clean --coverage --type relwithdebinfo --all

# Verify coverage build (check for .gcno files)
find build_x86_64 -name "*.gcno" | head
```

Build & Environment

- `relwithdebinfo` : Recommended for development (1.5–2x slower, optimized with debug info)
- `debug` : Most detailed coverage (3–5x slower, no optimization, full symbol info)

Report Generation

Trigger coverage analysis using the following commands

Execution Flow (Manual vs Unified)

Manual Execution (from `tests/cpp_example/`)

```
# Quick E2E with coverage (image tests only, 2-3 minutes)
pytest -m e2e -k "test_image_inference_e2e" --coverage -v

# Full E2E tests with coverage (8-10 minutes)
pytest -m e2e --coverage -v

# Run specific models with coverage
pytest -m e2e -k "yolov7" --coverage -v
```

Unified Test Runner (from project root)

```
# [Recommended] Build with debug mode, then run coverage
./build.sh --clean --coverage --type debug --all && ./run_tc.sh --cpp --coverage

# Run C++ coverage (standalone)
./run_tc.sh --cpp --coverage
```

Report Interpretation (HTML Report & Filtering)

ⓒ Copyright 2026 DEEPX All Rights Reserved.

After running tests with `--coverage`, several reports are generated in `tests/cpp_example/coverage/`

- **Console summary:** line coverage (gcovr); line, function and branch coverage with the lcov fallback.
- **HTML report:** Located at `html/index.html`. Shows line-by-line visualization with uncovered code in **red**.
- **XML (Cobertura) & JSON:** Timestamped for CI/CD integration.

View the HTML Report

```
# Open in your browser
firefox tests/cpp_example/coverage/html/index.html
# or
xdg-open tests/cpp_example/coverage/html/index.html
```

Report Features

- Overall and file-by-file coverage statistics.
- Line-by-line visualization with **uncovered code highlighted in red**.
- Branch coverage analysis.

Coverage Filtering Rules

- **Included:** All source files within the `src/` directory.
- **Excluded:** System headers (`/usr/*`), `third_party/*`, `extern/*`, and the `tests/` directory itself.

4.2.6 Repository Checks

If you modify the examples or add a model, `scripts/ci_checks.sh` checks the repository's consistency (model registry, factory layout, generated sources, Python syntax) without an NPU.

```
bash scripts/ci_checks.sh         # run every check, then print a PASS / SKIP / FAIL summary
bash scripts/ci_checks.sh --list  # list the check names
```

A check whose prerequisite is missing on this machine (for example the DX-RT headers or a cross compiler) prints `SKIP` with the reason, and the run still exits 0. Only a `FAIL` makes it exit 1.

© Copyright 2026 DEEPX All Rights Reserved.

5. DX-APP Python Example

5.1 DX-APP Python Usage Guide

This guide explains how to navigate and use the Python example tree in DX-APP.

5.1.1 Overview

The Python examples are located under `src/python_example/` and are organized by:

- **task**
- **model family**
- **variant** (the `.dxnn` stem; each folder has sync, async, and C++ post-process entries)

All examples share a common runtime layer under `src/python_example/common/` that provides base interfaces, processors, runners, input sources, visualizers, and utilities. This is the Python counterpart of `src/cpp_example/common/` — both languages implement the same factory-based architecture with the same core modules. Each variant directory contains thin entry-point scripts and a factory that wires shared components together.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Representative task directories include:

- `image_classification/` — EfficientNet, AlexNet, ResNet, MobileNet, CasViT, etc.
- `object_detection/` — YOLOv3/v4/v5/v6/v7/v8/v9/v10/v11/v12/v13, YOLO26, YOLOX, EfficientDet, RT-DETR, NanoDet, DAMOYOLO, SSD. PPU variants live under families such as `yolo_ppu/`
- `face_detection/` — SCRFD, YOLOv5Face, YOLOv7Face, RetinaFace
- `pose_estimation/` — YOLOv8-Pose
- `semantic_segmentation/` — BiSeNet, DeepLabV3+, SegFormer
- `instance_segmentation/` — YOLOv8Seg, YOLOv26Seg
- `depth_estimation/` — FastDepth, SCDepthV3
- `face_recognition/` — ArcFace
- `oriented_object_detection/` — YOLOv26OBB
- `image_denoising/`, `low_light_enhancement/`, `super_resolution/`
- `hand_landmark/` — Hand landmark estimation
- `person_attribute/` — Attribute recognition (DeepMAR)
- `person_reid/` — Person re-identification (RepVGG)
- `face_landmark/` — Face alignment / 3D landmark (3DDFA v2)

For the full repository-level structure, refer to DX-APP Example Source Structure.

5.1.2 Architecture & Design Pattern

Architecture Strategy

Shared Runtime Layer (`common/`)

The `common/` directory is the engine behind all Python examples:

Module	Role
`common/base/`	Abstract interfaces: task factories (`IDetectionFactory` , `IClassificationFactory` , …), `IPreprocessor` / `IPostprocessor` , `IVisualizer` , `IInputSource`
`common/config/`	

© Copyright 2026 DEEPX All Rights Reserved.

Module	Role
	`load_config()` / `validate_config()` — reads a flat `--config` JSON
`common/variant_config.py` , `common/variants.py`	Build processors from `<variant>/config.json` ; resolve a variant key through `config/model_registry.json`
`common/processors/`	Shared processors covering all model families
`common/runner/`	`SyncRunner` , `AsyncRunner` , `run_dir` , `verify_serialize` , `sr_tiling` , `args` — generic execution engines with built-in profiling
`common/inputs/`	Input source abstraction: image, video, camera, RTSP
`common/visualizers/`	Task-specific visualizers (detection, segmentation, pose, etc.)
`common/utility/`	Labels, preprocessing, profiling, drawing helpers

Factory Pattern & Model Registry

Each variant directory has a `factory/{variant}_factory.py` that implements the task factory (for example `IDetectionFactory`). The factory is config-driven: every variant folder carries the same family factory class, which builds the processor classes and their arguments from that folder's `config.json` (`preprocessor` , `postprocessor` , `visualizer`). For `yolov9-s_640x640` , an anchor-free model, the postprocessor is `YOLOv8Postprocessor` , which decodes the whole YOLOv8 to YOLOv13 detection family:

```
"preprocessor":  { "class": "LetterboxPreprocessor", ... },
"postprocessor": { "class": "YOLOv8Postprocessor", ... },
"visualizer":    { "class": "DetectionVisualizer", ... }
```

```
class Yolov9Factory(IDetectionFactory):
    def __init__(self, config: dict = None, variant: str = None):
        self.variant = variant or default_variant(_FAMILY_DIR)
        self.spec = load_variant_config(_FAMILY_DIR, self.variant)
        self.config = {**(self.spec.get("config") or {}), **(config or {})}

    def create_postprocessor(self, input_width: int, input_height: int):
```


© Copyright 2026 DEEPX All Rights Reserved.

```
        # builds the "postprocessor" entry of config.json: YOLOv8Postprocessor
        return self._build("postprocessor", input_width, input_height)
```

The entry-point script fixes the variant and delegates to the runner:

```
from factory import Yolov9Factory
from common.runner import SyncRunner
from common.runner.entry import run_entry

run_entry(_script, factory_cls=Yolov9Factory, runner_cls=SyncRunner,
          description="yolov9-s_640x640 sync inference",
          fixed_variant="yolov9-s_640x640")
```

Model Registry (`config/model_registry.json`)

A JSON registry stores per-model metadata (task, postprocessor type, input dimensions, thresholds). `scripts/add_model.sh` copies a reference variant of the same family into a new `<task>/<family>/<variant>/` directory (factory, `config.json` and the four entry scripts), renames it, and adds the variant to this registry. When a matching postprocessor already exists, onboarding a model needs no new source code.

Directory & File Pattern

Each
task directory contains model families. Each variant is a subdirectory named after the `.dxnn` stem and holds that variant's factory, `config.json` , and thin entry scripts.

```
src/python_example/object_detection/yolov8/
└── yolov8-n_640x640/
    ├── factory/
    │   └── yolov8-n_640x640_factory.py
    ├── config.json
    ├── yolov8-n_640x640_sync.py
    ├── yolov8-n_640x640_async.py
    ├── yolov8-n_640x640_sync_cpp_postprocess.py
    └── yolov8-n_640x640_async_cpp_postprocess.py
```

5.1.3 Execution Framework

Execution Variants

Pure Python Flow (`*_sync.py` , `*_async.py`)

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Use these when you want:

- easier logic inspection — post-processing is readable Python in `common/processors/`
- Python-first experimentation
- simpler debugging during algorithm development

C++ Post-process Flow (`*_cpp_postprocess.py`)

Use these when you want:

- faster post-processing — uses C++ via pybind11 (`dx_postprocess`)
- more realistic performance validation

CLI Interface

All Python examples use `argparse` via `common/runner/args.py` and share a consistent interface:

Flag	Short	Type	Description
`--model`	`-m`	string	Path to `.dxnn` model file. Optional: if omitted, the example's default model is used and auto-downloaded when missing
`--variant`	—	string	Variant key (the `.dxnn` stem, e.g. `yolov5-s_640x640`); sets the model path from `config/model_registry.json` . An explicit `--model` takes precedence
`--image`	`-i`	string	Input image file or directory
`--video`	`-v`	string	Input video file
`--camera`	`-c`	int	Camera device index

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Flag	Short	Type	Description
`--rtsp`	`-r`	string	RTSP stream URL
`--save`	`-s`	flag	Save rendered output to a run directory
`--save-dir`	—	string	Base output directory (default: `artifacts/python_example/`)
`--dump-tensors`	—	flag	Dump input/output tensors to `.npy` files
`--loop`	`-l`	int	Inference repeat count (default: 1; bare `--loop` = 2)
`--display`	—	flag	Show the visualization window (default: on)
`--no-display`	—	flag	Disable visualization window
`--show-log`	—	flag	Enable verbose log output (default: quiet)
`--config`	—	string	Model config JSON path (auto-detected if omitted)
`--fast-postprocess`	—	flag	Opt-in faster postprocessing variant where available (standard path is the default)
`--sr-tile-halo`	—	int	Tile overlap in low-resolution pixels for tiled super-resolution, 0..4 (default: 4)
`--output`	`-o`	string	Accepted by some examples (see `--help`) but not used by the

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Flag	Short	Type	Description
			runners in this release; save results with `--save` / `--save-dir` or `DXAPP_SAVE_IMAGE`
`--calib-dir`	—	string	SFA3D only. Accepted but not applied in this release: calibration is read from `<stem>.txt` next to the `.bin`, or from `../calib/<stem>.txt`
`--image2-dir`	—	string	SFA3D only. Accepted but not applied in this release: the camera image is read from `../image_2/<stem>.png` or `.jpg`
`--help`	`-h`	—	Show usage

- **Input source:** `--image`, `--video`, `--camera`, and `--rtsp` form a mutually exclusive group. If none is specified, the variant's **default sample image** (`default_image` in its `config.json`) is used; image-only examples instead print a hint to pass `--image` and exit.
- **Windows camera and video files:** the examples set `OPENCV_VIDEOIO_MSMF_ENABLE_HW_TRANSFORMS=0` (OpenCV's MSMF hardware transforms off) before `cv2` is imported, so it covers every camera, video file and stream they open: with them on, a USB webcam takes about 7 s to open; off, about 0.15 s at the same frame rate. To keep them on, set `OPENCV_VIDEOIO_MSMF_ENABLE_HW_TRANSFORMS=1` before running; a value you set is not changed.

© Copyright 2026 DEEPX All Rights Reserved.

> **NOTE**
>
> **Image-only examples:** a variant whose `config.json` sets `"image_only": true` (8 tasks such as `face_recognition`, `person_reid`, `3d_object_detection`, plus the CasViT ReID and some CLIP variants) rejects `--video`, `--camera` and `--rtsp`; with `"include_stream_inputs": false` those options are not registered at all.
>
> `face_recognition`, `face_attribute`, `person_attribute`, `person_reid`, `image_retrieval`, `visual_place_recognition` need a crop of a pre-detected subject, or a committed gallery to rank against. Running a single embedding model on a raw stream without a preceding detector would not produce valid results.
>
> `object_pose_estimation`, `3d_object_detection` do not take a video frame at all: `3d_object_detection` (SFA3D) consumes a LiDAR point cloud (`sample/kitti/velodyne/*.bin`) and refuses any other file type, and DOPE scores the pose of a static object from a single image.

5.1.4 Getting Started

To set up and run a first Python example, follow DX-APP Quick Start. The Python scripts need no build of their own; the default `./build.sh` installs `dx_postprocess`, which the `*_cpp_postprocess.py` variants use.

5.1.5 Advanced Operations & Debugging

Runtime Features

Auto-Download

When `--model` is omitted, the runner uses this example's default model from `config/model_registry.json` and, if the file is missing, downloads it via `setup_sample_models.sh`. If the download fails, an error message with manual download instructions is displayed. An explicit `--model` path that does not exist is an error (no download). Missing input files, including videos, are not downloaded; run `./setup.sh` or `./setup_sample_videos.sh` first.

Default Input Fallback

If no input source is provided, the runner automatically selects a default sample image appropriate for the task type (e.g., `sample/img/sample_street.jpg` for object detection). A log message indicates which default was applied:

© Copyright 2026 DEEPX All Rights Reserved.

```
[DXAPP] [INFO] No input specified. Using default sample: sample/img/sample_street.jpg
```

Signal Handling

Both runners install SIGINT/SIGTERM handlers (`common/runner/interrupts.py`), mirroring the C++ runners: the first Ctrl-C or a SIGTERM stops gracefully (the video writer is finalized and the summary printed); a later Ctrl-C, more than 200 ms after the first, terminates at once; repeats within 200 ms count as one request; SIGTERM never escalates. Python runs signal handlers between bytecodes, so during one long native call the escalation waits until the call returns. The async pipeline uses a SENTINEL chain to propagate the stop through its queues.

The handlers are installed only while a runner's `run()` executes and the previous handlers are restored when it returns, so importing the runner modules changes nothing and each `run()` starts with a fresh first request. A stop outside the stream loop (during setup or in image mode) prints `Interrupted by user.` and exits with status 0 instead of a `KeyboardInterrupt` traceback.

Output order

The async runner waits for dxrt's results in submit order (one wait worker and FIFO queues between the workers), so saved output and `DXAPP_VERIFY` records follow the input order and equal the sync runner's frame by frame. The live window also shows frames in input order, but when nothing is saved it is a best-effort preview: it draws at most about 60 frames a second and, when the window falls behind, drops the oldest waiting frame instead of slowing inference (`offer_latest` in `common/runner/async_runner.py`).

Output Management (`--save`)

When `--save` is enabled, a timestamped directory is created (e.g., `artifacts/python_example/{script}-image-{name}-{timestamp}/`, where `{script}` is the entry script's name such as `yolov9-s_640x640_sync`) containing `run_info.txt`, saved images/video, and optional tensor dumps.

Headless Mode

On Linux, when `DISPLAY` / `WAYLAND_DISPLAY` are absent, `cv2.imshow()` is skipped automatically (Windows and macOS always open the window). Use `--no-display` for explicit headless operation.

Model Configuration (`--config`)

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Runtime parameters (thresholds, top-k, etc.) come from `<task>/<family>/<variant>/config.json` via `load_variant_config`, with alias normalization (`score_threshold` → `conf_threshold`). The thin script in that folder fixes the variant. A single-model extract may keep `config.json` beside the entry script.

Fast Postprocessing (`--fast-postprocess`)

An opt-in, performance-oriented postprocessing path. By default the runner always uses the standard postprocessor; passing `--fast-postprocess` selects a faster variant **only for the model families that provide one** — other models keep the standard path and print a warning, so the flag is always safe to add. It does not affect the `*_cpp_postprocess` scripts, whose post-processing runs in C++. The optimization skips work the standard path performs *before* thresholding (e.g. running `sigmoid` over the full anchor grid, taking `argmax` over every anchor, or upsampling each mask to the full input resolution).

There are two accuracy tiers:

Tier	Output vs. standard path	Model families
Exact	Byte-identical (verified by parity unit tests)	Object detection — every variant using `YOLOv5Postprocessor` (YOLOv3/v4/v5/v6/v7), EfficientDet
Approximate	Differs only at sub-pixel mask boundaries (binary masks agree at high IoU)	Instance segmentation (YOLOv8-seg family), YOLACT, SegFormer semantic segmentation, depth estimation (values differ by up to ~3/255)

- **Exact tier (detection):** the fast path gates anchors *before* the expensive decode. Because objectness/score gating is monotonic and the survivors are decoded with the same formulas, the detections are bit-for-bit identical to the standard path — it can be enabled with no accuracy cost.
- **Approximate tier (segmentation):** the fast path crops and resizes masks at prototype (or output) resolution instead of upsampling each mask to the full input resolution first. This changes only sub-pixel boundary interpolation; on the profiled workloads the measured speedup is roughly 2.3x for instance segmentation and 4.9x for SegFormer-style semantic segmentation (the exact figure depends on the model, resolution, and host CPU).

© Copyright 2026 DEEPX All Rights Reserved.

When to enable. The benefit is largest when postprocessing dominates end-to-end latency — typically high-resolution feature grids or large anchor×class counts (dense detection heads, many-class detectors, high-resolution segmentation). Profile your own workload rather than assuming a fixed factor.

When to keep the default. The standard path is always the default and remains the accuracy reference. For the approximate (segmentation) tier, keep the standard path when exact mask boundaries matter.

```
# Object detection (exact tier) — identical results, faster decode
python src/python_example/object_detection/yolov7/yolov7_640x640/yolov7_640x640_sync.py \
    --model assets/models/yolov7_640x640.dxnn --image sample/img/sample_street.jpg --fast-
postprocess
```

Verification & Diagnostics

Numerical Verification (DXAPP_VERIFY)

Set `DXAPP_VERIFY=1` to serialize post-processing results. Each run writes `logs/verify/{model}.json` (override the directory with `DXAPP_VERIFY_DIR`) holding the last frame, and `{model}.frames.jsonl` with one record per frame (`"frame": 0, 1, …` in input order). Writes are serialized and the JSON is replaced atomically, so a reader never sees a half-written file. Every runner dumps its real results, including tiled super-resolution (statistics of the stitched output), 3D detection and YOLOPv2 (boxes plus drivable/lane mask statistics); SuperPoint records carry each frame's keypoints and the descriptor size. The `*_cpp_postprocess` scripts dump too (raw arrays as `output_stats` / `output_stats_list`, for the scripts whose post-processing returns arrays).

Frame numbers count per process, so with `--loop N` on a video the records of later loops continue the numbering.

Tensor Dump for Debugging (--dump-tensors)

Dumps raw input/output tensors as `.npy` files. On exception, tensors and a `reason.txt` are auto-dumped for debugging.

Model Validation (optional)

```
# Run NPU inference for all supported models
bash scripts/validate_models.sh --lang py
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

Environment Variables Reference

Variable	Description
`DXAPP_SAVE_IMAGE`	Save visualization to the specified file path (no `--save` required)
`DXAPP_VERIFY`	When `1`, dump JSON verification data
`DXAPP_VERIFY_DIR`	Directory for `DXAPP_VERIFY` output (default `logs/verify`)
`DXAPP_PROGRESS`	When set (any value but `0`), print `[PROGRESS] frames=N fps=F avg_fps=A read_ms=… pre_ms=… infer_ms=… post_ms=…` once a second and at exit (front ends)
`DXAPP_SR_TILE_HALO`	Tile overlap for tiled super-resolution, 0..4 (overridden by `--sr-tile-halo` and config.json `sr_tile_halo`)
`DXAPP_SCREEN_W` / `DXAPP_SCREEN_H`	Screen size used to size the display window (both must be set)

5.1.6 Supplementary Information

Component Relationships

The `*_cpp_postprocess.py` variants depend on the shared Python binding exposed from `src/bindings/python/dx_postprocess/`.

See also: DX-APP Pybind PostProcess Overview

Developer Resources

- For contributor workflows, use DX Tool Guide
- For test execution, use DX-APP Python Example Tests
- For repository layout details, use DX-APP Example Source Structure

© Copyright 2026 DEEPX All Rights Reserved.

5.2 DX-APP Python Example Tests

5.2.1 Overview

The Python Example project includes a comprehensive test framework built on `pytest`. It allows developers to validate script integrity, verify CLI arguments, and perform automated performance benchmarking across different models and hardware variants.

Key Features

- **Framework-Based:** Uses common base classes to ensure consistent testing logic across the entire repository.
- **Layered Testing Strategy:** Provides a comprehensive validation path - **Unit** → **CLI** → **E2E**.
- **Smart Mocking:** Combines high-speed software mocks (hardware-free) with real NPU hardware validation.
- **Centralized Configuration:** Model metadata comes from `config/model_registry.json`. Test discovery lives in `tests/test_helpers/`.
- **Automated Performance Tracking:** E2E tests automatically capture and aggregate NPU performance metrics (FPS, Latency).

5.2.2 Test Classification & Prerequisites

Test Strategy Levels

Quick Start Tests

The framework employs a layered approach to isolate issues effectively

- **Unit Tests (`pytest unit/`):** Verifies core logic using mocks. Fast and ideal for CI/CD pipelines without NPU hardware
- **CLI Tests (`-m cli`):** Ensures command-line arguments correctly trigger intended input modes (image/video/camera)

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Advanced Test Selection

- **E2E (End-to-End) Tests (`-m e2e`):** High-fidelity tests using real `.dxnn` models and NPU hardware to capture actual performance
  Video E2E tests run every stream-capable model except the W6 face detectors (`VIDEO_TOO_SLOW_MODELS`) and the anomaly-detection and super-resolution examples (listed in `IMAGE_ONLY_TASKS` in `tests/test_helpers/constants.py`). Super-resolution's stream path runs on a 6-frame low-resolution clip (`test_super_resolution_stream_e2e`).

Feature Tests (selection)

File	Marker	Scope
`test_verify.py`	`verify`	`DXAPP_VERIFY` output of every downloaded model, sync and async (`*_cpp_postprocess` included): one non-empty record, per-frame file
`test_sync_async_parity.py`	`verify`	Async output equals sync frame by frame (120-frame video clip; `yolov5-s_640x640` , `yolopv2_384x640` , `superpoint_480x640` and one model per postprocessor family)
`test_signal_handling.py`	`signal_handling`	SIGINT graceful shutdown; SIGTERM in save mode finalizes the video and prints the summary

Environment Setup

Configuration Reference

- `pytest.ini` : Defines the fixed custom markers, log formats, and global settings. `conftest.py` also registers one marker per model family and variant directory under `src/python_example/` at run time.
- `conftest.py` : Contains shared fixtures, mock infrastructure, and setup/teardown utilities.

Interpreter for the examples (`DXAPP_TEST_PYTHON`)

ⓒ Copyright 2026 DEEPX All Rights Reserved.

The mocks in `conftest.py` exist only inside the pytest process. Most tests launch the real example scripts as subprocesses, and those import `dx_engine`, `dx_postprocess` and `cv2` for real. The interpreter for those subprocesses is chosen in this order:

1. `$DXAPP_TEST_PYTHON`, when set.
2. The dx-runtime venv, `dx-runtime/venv-dx-runtime/bin/python3`, when it exists.
3. The interpreter running pytest.

The pytest header prints the choice (`example python (DXAPP_TEST_PYTHON): ...`). The simplest setup is to run pytest itself from that venv:

```
VENV=../venv-dx-runtime                  # from dx_app/
$VENV/bin/python -m pip install -r requirements.txt -r tests/python_example/requirements.txt
$VENV/bin/python -m pytest tests/python_example
# another interpreter for the examples:
DXAPP_TEST_PYTHON=/path/to/python3 $VENV/bin/python -m pytest tests/python_example
```

Behind a TLS-intercepting proxy, pass the system CA bundle to pip (`--cert /etc/ssl/certs/ca-certificates.crt`); never disable verification.

Without `DISPLAY`/`WAYLAND_DISPLAY`, both suites set `QT_QPA_PLATFORM=offscreen` for the examples they start, so OpenCV windows do not abort on a headless machine. When an example subprocess times out, it gets SIGTERM first, then SIGKILL if it is still alive (`tests/test_helpers/proc.py`).

Coverage Scope

Variant scripts are discovered from `src/python_example/<task>/<family>/<variant>/` by `tests/test_helpers/utils.py`. Adding a new source directory does not automatically guarantee full test coverage until registry metadata and the discovery helpers agree.

5.2.3 Test Infrastructure

Project Structure & Reference

The test files sit next to each other. They are not split into per-task directories.

```
tests/python_example/
├── README.md
├── conftest.py
├── pytest.ini
```


© Copyright 2026 DEEPX All Rights Reserved.

```
├── requirements.txt
├── test_cli_basic.py
├── test_cli_help.py
├── test_classification_parity.py
├── test_dump_tensors.py
├── test_e2e.py
├── test_e2e_camera_rtsp.py
├── test_multi_loop.py
├── test_preopt_contract.py
├── test_save_mode.py
├── test_signal_handling.py
├── test_sync_async_parity.py
├── test_verify.py
├── test_visualization.py
└── unit/                # Processor, layout, and registry unit tests
```

Shared Module (`tests/test_helpers/`)

All test files import shared constants and utilities from `tests/test_helpers/` :

- `constants.py` : `TASK_IMAGE_MAP` , `MODEL_IMAGE_OVERRIDE` , path constants
- `utils.py` : `setup_environment()` , `discover_python_scripts()` , `strip_variant_suffix()`
- `proc.py` : subprocess runs with a timeout (SIGTERM first, then SIGKILL)
- `pytest_support.py` : shared pytest options (`--loop` , `--camera-index` , `--rtsp-url` , `--stream-duration`)
- `verify.py` : readers and comparers for `DXAPP_VERIFY` output

Asset Management

Assets required for testing are automatically managed by the setup scripts in the project root:

- **Models**: Run `./setup_sample_models.sh` to fetch models for E2E validation.
- **Data**: Run `./setup.sh` to ensure sample images and videos are placed in the expected directories.

5.2.4 Test Execution Guide

Execution Methods

Method A. Quick Start Guide

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Navigate to the `tests/python_example/` directory to execute tests.

```
# 1. Install testing dependencies
pip install -r requirements.txt

# 2. Run all available tests (Unit + CLI + E2E)
pytest

# 3. Skip the e2e-marked tests. The save_mode, dump_tensors, multi_loop, signal_handling,
#    verify and visualization tests still run on the NPU for every downloaded model;
#    for a hardware-free run use: pytest unit/ and pytest -m cli
pytest -m "not e2e"

# Specific test levels
pytest unit/           # Unit tests only
pytest -m cli          # CLI tests only
pytest -m e2e          # End-to-end tests only
```

Method B. Manual Execution (Advanced Selection)

Filter tests by model family or variant to optimize development time.

```
# By model family (substring of the test id, e.g. yolov7_640x640_sync)
pytest -k yolov7
pytest -k scrfd
pytest -k efficientnet

# By variant (marker)
pytest -m yolov7_640x640

# Combinations
pytest -m e2e -k "yolov7 or yolov8"
pytest -m cli -k yolov7
```

Result Analysis

E2E Hardware Benchmarking

The E2E suite functions as an **automated benchmarking tool** for hardware performance.

- **Requirements:** Requires installed `dx_engine, dx_postprocess`, and relevant model assets. In practice, prepare assets with `./setup.sh` or `./setup_sample_models.sh` before E2E execution.
- **Visual Output:** `E2E_DISPLAY=1 pytest -m e2e` drops `--no-display`. Image tests (sync and async) open a window and wait for q/ESC on every result; the video test skips async scripts in this mode.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
# Run E2E with display
E2E_DISPLAY=1 pytest -m e2e
```

> **NOTE**
>
> With `E2E_DISPLAY=1` the video E2E test runs sync scripts only, and every image test waits for a key press.

- **Performance Reporting:** After E2E tests finish, a performance summary is printed to the console. Detailed logs, including FPS and Latency per model variant, are automatically saved to `tests/python_example/performance_reports/performance_report_{e2e|e2e_short}_YYYYMMDD_HHMMSS.csv`. The label is `e2e_short` when the run selects that marker, and `e2e` otherwise.

5.2.5 Repository Checks

The Python examples are covered by the same `bash scripts/ci_checks.sh` that checks the C++ tree without an NPU. See *Repository Checks* in DX-APP C++ Example Tests.

© Copyright 2026 DEEPX All Rights Reserved.

6. DX-APP Multi-Model Graph

The multi-model graph runs **several models over one input** in a single application: a detector, a segmenter and a pose model side by side, a detector that feeds a re-identification model with each person it finds, or a super-resolution model whose output image is handed to a detector. Decision logic such as zones, line crossing, counting and alerts can be added around the models.

The whole pipeline is described in a **node-graph JSON file**: which models run, how they are connected, which regions are cropped and handed on. Changing a model or a connection is a text edit. There is no C++ to edit and nothing to rebuild.

```
./bin/multi_model_graph_async \
    --graph src/cpp_example/multi_model_graph/fanout_od_seg_pose.json \
    --output result.png
```

This command runs a detector, a semantic segmenter and a pose model over one image and draws all three results on one picture.

6.1 Overview

6.1.1 Components

Component	Description
`bin/multi_model_graph_sync`	Runs the graph one stage at a time, in topological order. The reference: slower, and the first thing to try when a result looks wrong
`bin/multi_model_graph_async`	Submits every node whose inputs are ready before waiting on any of them, and keeps several frames in flight. Use it for throughput
`dx_graph` (Python module)	Runs the same graphs from Python, with the same engine, reports and pictures as the CLI
`src/cpp_example/multi_model_graph/*.json`	22 sample graphs (see *Sample Graphs*)
`run_multi_model_demo.sh`	

© Copyright 2026 DEEPX All Rights Reserved.

Component	Description
	Validates every sample graph and runs the demo scenarios
`docs/graph_models.md`	The models that can run in a graph, with what each one produces and consumes

Both binaries give the same result for the same input: their `--report` files are byte-identical (without `--report-timing`, which adds measured times).

6.1.2 Supported Models and Platforms

- **Models**: 492 of the 497 registered models can run in a graph. A model is named by its variant, the `.dxnn` file stem (for example `yolov8-n_640x640`). `--list-models` and `docs/graph_models.md` list them.
- **Inputs**: images, video files, cameras (`camera:<N>`) and RTSP streams (`rtsp://...`), one or several at once.
- **Platforms**: Linux x86_64 is built and tested. aarch64 is compile-checked only. Windows is not built or tested in this release (see *Graph engine on Windows* in DX-APP Installation and Build).
- **DX-RT**: 3.5.0 or later for the DX Model Zoo 2_5_0 files (`.dxnn` container v9). The graph checks the container before it loads a model, and on an older runtime it stops with a `MODEL_LOAD` error (`--check` exits 1 on it too) that says `<file>: .dxnn container v9 needs DX-RT >= 3.5.0, but this runtime is 3.4.1. Use the v8 file (dxnn/2_4_0) or upgrade DX-RT.`

6.2 Build and Run

6.2.1 Build

The graph binaries are built by the **full** build, or by `./build.sh --target multi_model_graph_sync multi_model_graph_async`. `dx_graph` is built by the full build only. The default `./build.sh` (`--minimal`) builds none of them.

```
# multi_model_graph_sync / multi_model_graph_async into bin/
./build.sh --all

# Also build dx_graph for a specific Python interpreter
/path/to/python -m pip install pybind11 numpy
./build.sh --all --python_exec /path/to/python
```


© Copyright 2026 DEEPX All Rights Reserved.

`dx_graph` is installed as `bin/python/dx_graph`. It needs pybind11, numpy and that Python's development headers; without them the build skips it and prints `dx_graph Python module skipped: <reason>`.

6.2.2 Run Your First Graph

Step 1. Check the graph without the NPU

`--check` validates a graph and lists each node, its model and whether the `.dxnn` file is present. It does not open the NPU.

```
./bin/multi_model_graph_sync --check src/cpp_example/multi_model_graph/
fanout_od_seg_pose.json
```

```
OK: graph "od-seg-pose" is valid (4 nodes, 3 edges, 3 models)

node      model                    produces   consumes    artifact
cam       (source)                 frame      -           sample/img/sample_people.jpg
od        yolov8-n_640x640         boxes      full_frame  yolov8-n_640x640.dxnn  [present, v9]
seg       bisenetv2_1024x2048      labelmap   full_frame  bisenetv2_1024x2048.dxnn  [present,
v9]
pose      yolov8-s-pose_640x640    keypoints  full_frame  yolov8-s-pose_640x640.dxnn  [present,
v9]
```

Step 2. Download missing models

A model that is not downloaded yet is listed as `[MISSING]`, followed by a `download_models.py` command for the missing files. That command names the files by their Model Zoo names and adds `--output <model-dir>`; the variant names work as well:

```
python3 scripts/download_models.py --models yolov8-n_640x640 bisenetv2_1024x2048 yolov8-s-
pose_640x640
```

Step 3. Run

```
# One image, result saved as an image
./bin/multi_model_graph_async --graph src/cpp_example/multi_model_graph/
fanout_od_seg_pose.json \
    --output result.png

# A video instead of the graph's own input, result saved as a video
./bin/multi_model_graph_async --graph src/cpp_example/multi_model_graph/
fanout_od_seg_pose.json \
    --input assets/videos/dance-group.mov --output result.mp4

# A camera, shown in a window (q or ESC stops)
```


© Copyright 2026 DEEPX All Rights Reserved.

```
./bin/multi_model_graph_async --graph src/cpp_example/multi_model_graph/
fanout_od_seg_pose.json \
    --input camera:0 --display
```

> **Windows:** video output supports `.mp4` and `.avi` only. `.mkv` needs OpenCV's FFmpeg video backend, which the Windows build does not include, and fails with exit code 1.

Each frame prints one line with what every node found, and the run ends with a summary. The first command prints:

```
graph:  od-seg-pose (4 nodes, 3 edges)
input:  Image file: sample/img/sample_people.jpg
frame 0: od boxes=4 seg labelmap pose keypoints=2
output: result.png
1 frame, 0 failed
```

6.2.3 Demo Script

`run_multi_model_demo.sh` checks every sample graph and runs the demo scenarios.

```
bash run_multi_model_demo.sh           # interactive menu
bash run_multi_model_demo.sh --check    # check every sample graph, no NPU
bash run_multi_model_demo.sh --all      # --check, then each scenario
bash run_multi_model_demo.sh --scenario hand_cascade
```

Option	Description
`--scenario <id or number>`	Run one scenario: `hand_cascade` (0), `logistics_volume` (1) or `dms` (2)
`--async`	Use `multi_model_graph_async` instead of `multi_model_graph_sync`
`--check`	Validate every `src/cpp_example/multi_model_graph/*.json`. Does not use the NPU
`--all`	`--check`, then each scenario with `multi_model_graph_sync`
`--list`	Print the scenarios and the sample graphs, then exit
`--output-dir <dir>`	Reports and saved frames (default: `reports/dev/multi_model_demo`)

© Copyright 2026 DEEPX All Rights Reserved.

Option	Description
`--models-dir <dir>`	`.dxnn` directory (default: `assets/models`)

6.3 Sample Graphs

The sample graphs are in `src/cpp_example/multi_model_graph/`. Each one is a starting point for your own graph.

Model pipelines

File	What it shows	Models
`fanout_od_seg_pose.json`	One frame, three models in parallel	`yolov8-n_640x640` + `bisenetv2_1024x2048` + `yolov8-s-pose_640x640`
`fanout_od_seg_depth.json`	The same, with depth instead of pose	`yolov8-n_640x640` + `bisenetv2_1024x2048` + `fastdepth_224x224`
`cascade_od_reid_track.json`	Detector, then a crop per person, then an embedding, with tracking	`yolov8-n_640x640` → `casvit-t_224x224`
`cascade_od_attr.json`	Detector, then a crop per person, then attributes	`yolov8-n_640x640` → `deepmar_resnet50_224x224`
`cascade_obb_cls.json`	Oriented detector, then an un-rotated crop, then a classifier	`yolo26-l-obb_1024x1024` → `resnet50_224x224`
`handoff_denoise_od.json`	Denoise, then detect on the denoised image	`dncnn-color_512x512` → `yolov5-n_640x640`
`handoff_sr_od_cls.json`	Super-resolve ×2, detect on the large image, classify each person (*)	`realesrgan-x2_192x192` → `yolov8-n_640x640` → `resnet50_224x224`
`chain_sr_od_pose.json`	Super-resolve ×2, detect, then top-down pose per person (*)	`realesrgan-x2_192x192` → `yolov8-n_640x640` → `vitpose-s_256x192`

© Copyright 2026 DEEPX All Rights Reserved.

File	What it shows	Models
chain_zerodce_od_pose_emb.json	Low-light enhancement, detect, then pose and an embedding per person	zerodce_400x600 → yolov8-n_640x640 → vitpose-s_256x192 + casvit-t_224x224
hand_cascade.json	Palm detector, then a crop per palm, then 21 hand landmarks	mediapipe-hand-detector_192x192 → mediapipe-hands-lite_224x224
dms_headpose.json	Face detector, then head pose (pitch, yaw, roll) on a cpu node	scrfd-500m_640x640 → headpose
logistics_volume.json	Detection, instance mask and depth in parallel, then a relative package-volume estimate on a cpu node	yolo26-n_640x640 + yolo26-n-seg_640x640 + yolo26-depth-n_768x768 → volume
multistream_od_two_sources.json	Two sources, each its own stream, through one shared tracked detector and one re-id model	yolov8-n_640x640 → casvit-t_224x224

() Not part of the DX-M1 sample runs in this release: with realesrgan-x2_192x192, these graphs are at the limit of DX-M1's device memory. If a run stops with MODEL_LOAD, see Device Memory*.

Decision graphs (logic nodes)

File	What it decides
zone_intrusion.json	People in a restricted zone raise an intrusion event (cooldown 10 s)
line_counting.json	Tracked people crossing a line are counted per direction, with a cross event each
ppe_zone_check.json	People in a zone are cropped for attributes; one without a hat raises no_hat
loitering_dwell.json	How long each tracked person stays in a zone; a dwell event after 5 s

© Copyright 2026 DEEPX All Rights Reserved.

File	What it decides
`zone_people_count.json`	People counted in two zones
`gallery_known_person.json`	Each person's re-id embedding is named after the nearest gallery entry, or `unknown`
`driver_attention.json`	The driver's head pose raises `looking_away` (yaw beyond ±30°) or `head_down` (pitch below −20°)
`multistream_people_count.json`	People counted per stream; a `crowded` event in a stream with 5 or more
`logistics_volume_alert.json`	The packages' summed volume estimate raises `load_high` above 100 (relative: tune per camera)

The decision graphs ship with a sample image. Their scenes are videos: run them with, for example, `--input assets/videos/person-pair-hallway.mp4 --loop --display`. `multistream_people_count.json` has two sources: give it one `--input <source_id>=<uri>` per source (see *Several Sources*).

Changing a model is a text edit. To run a different detector in `cascade_od_reid_track.json`, change one string and download the new model. The same binary runs it:

```
# in cascade_od_reid_track.json: "model": "yolov8-n_640x640"  ->  "model": "yolo26-s_640x640"
python3 scripts/download_models.py --models yolo26-s_640x640
./bin/multi_model_graph_async --graph src/cpp_example/multi_model_graph/
cascade_od_reid_track.json \
    --output swapped.png
```

Any model whose `produces` and `consumes` fit the slot can be used. `--check` tells you before the run whether it fits.

6.4 Writing a Graph

6.4.1 Graph File

© Copyright 2026 DEEPX All Rights Reserved.

A graph is a JSON file with a `version`, a `name`, a list of `nodes` and a list of `edges`. This is `cascade_od_reid_track.json`: a detector finds people, each person is cropped, and a re-identification model computes an embedding per crop.

```
{
  "version": 1,
  "name": "person-reid",
  "nodes": [
    { "id": "cam",  "type": "source", "uri": "sample/img/sample_people.jpg" },
    { "id": "od",   "model": "yolov8-n_640x640",
      "params": { "score_threshold": 0.35 },
      "track":  { "algo": "iou", "iou": 0.3, "max_age": 30 } },
    { "id": "reid", "model": "casvit-t_224x224" }
  ],
  "edges": [
    { "from": "cam", "to": "od" },
    { "from": "od",  "to": "reid",
      "roi": { "classes": ["person"], "pad": 0.05, "max": 16 } }
  ]
}
```

Key	Meaning
`version`	Always `1` in this release
`name`	Free text, shown in messages and in `--report`
`nodes[].id`	The node's name, used by edges and in every message
`nodes[].params`	Overrides the model's `config.json` for this node only (see *Parameters*)
`nodes[].track`	Assigns a track id to each box the node outputs (see *Tracking*)
`edges[].roi`	Makes the edge a crop edge: the consumer runs once per box (see *ROI Options*)

The files are strict JSON: comments are not accepted.

6.4.2 Nodes

Kind	Written as	Meaning
source		

© Copyright 2026 DEEPX All Rights Reserved.

Kind	Written as	Meaning
	`{ "id": "cam", "type": "source", "uri": "..." }`	Where frames come from: an image, a video file, `camera:0` or an `rtsp://` URL. A relative `uri` is read from the current directory when it exists there, else from the repository root. `--input` replaces it
model	`{ "id": "od", "model": "yolov8-n_640x640" }`	One registered model, named by its variant. A model node has no `"type"`
cpu	`{ "id": "hp", "type": "cpu", "op": "headpose" }`	A fixed computation on the CPU after its inputs: `headpose` (pitch, yaw, roll from five face landmarks) or `volume` (a relative package-volume estimate from boxes, instance masks and depth)
logic	`{ "id": "zone", "type": "logic", "op": "zone", "params": { ... } }`	A decision around the models: `filter`, `zone`, `line`, `count`, `dwell`, `condition`, `gallery` or `user` (see *Decision Logic*)

6.4.3 Edges

Kind	Written as	The consumer receives
frame	`{ "from": "cam", "to": "od" }`	The whole frame: the source frame, or the image the producer made
ROI	`{ "from": "od", "to": "reid", "roi": { } }`	One cropped region per box the producer found. The consumer runs once per crop
result	`{ "from": "face", "to": "hp", "carry": "result" }`	The producer's result itself. The consumer must be a cpu or a logic node

© Copyright 2026 DEEPX All Rights Reserved.

Every model `produces` one result shape (`boxes` , `obboxes` , `instances` , `keypoints` , `labelmap` , `densemap` , `image` , `scores` , `vector`) and `consumes` a `frame` , a `roi` (crop) or `either` . An edge is accepted only when both ends fit:

- A frame edge needs a consumer that consumes `frame` or `either` . From a model node it is allowed only when the producer's output is an `image` (denoising, low-light enhancement, super-resolution): the consumer then runs on that image (*image hand-off*). Its results carry `origin.inv_align` , which maps them back to the source frame; the drawing applies it.
- An ROI edge needs a producer of `boxes` , `obboxes` or `instances` , and a consumer that consumes `roi` or `either` .
- A node cannot receive both a full frame and ROI crops. Split it into two nodes.
- Two edges cannot share the same `from` and `to` , and the graph must have no cycle.
- Two sources feeding one node are two streams, not two inputs (see *Several Sources*).

A graph that breaks a rule is rejected by `--check` with the node or edge and the fix (see *Checking and Troubleshooting*).

6.4.4 ROI Options

All options are optional; `"roi": {}` crops every box unchanged.

Key	Meaning	Default
`classes`	Keep only these class names, for example `["person"]`	all classes
`min_score`	Confidence floor, 0 to 1 (`0.35` , not `35`)	the producer's threshold
`min_area`	Pixel-area floor: checked on the detected box, and again on the crop after `pad` and clipping to the frame	`0`
`pad`	Grow each box by this fraction of its size on every side, 0 to 1 (`0.05` is 5 %)	`0`
`max`	Keep at most this many boxes, highest score first	no limit
`align`		none

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Key	Meaning	Default
	`"face5"`: align the crop from five facial landmarks (face detectors only)	

The options are applied in this order: class, score and area filter, then the `max` cut, then tracking, then `pad` and clipping, then alignment (or un-rotation of an oriented box), then the crop. Every second-stage result is drawn in source-frame coordinates, including the inverse of an `align` or of an OBB un-rotation. In `--report`, a result keeps the coordinates of the crop or image it ran on, and its `origin.inv_align` (a 2×3 affine) and `origin.src_box` map it back to the source frame. Logic nodes always work in source-frame pixels.

6.4.5 Tracking

`"track"` on a detector node gives every box it outputs a `track_id` that follows the object from frame to frame. It is declared on the producer, so every consumer of its boxes sees the same ids.

Key	Meaning	Default
`algo`	The tracker. `"iou"` is the only one	`"iou"`
`iou`	Overlap a box needs to continue a track, greater than 0 and at most 1	`0.3`
`max_age`	Frames an unmatched track survives (0 or more)	`30`

`"track": {}` uses the defaults. Tracking is applied in frame order, so in `multi_model_graph_async` a tracked node adds some latency; throughput is unaffected.

6.4.6 Parameters

A node's `"params"` overrides the model's `config.json` for that node only, so one model can appear twice in a graph with two thresholds:

```
factory defaults  <  <task>/<family>/<variant>/config.json  <  node "params"
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

Values are numbers (`score_threshold` , `nms_threshold` , ...), or strings and string arrays for the keys a model reads as text, such as `class_names` . `class_names` renames a model's classes for one node, which keeps an ROI filter readable:

```
{ "id": "od", "model": "yolov8-n_640x640", "params": { "class_names": ["pedestrian"] } }
```

6.4.7 Device Memory

Every model node opens its own engine on the NPU, even two nodes that name the same model. A graph whose engines do not fit in the NPU's memory fails while loading, with `MODEL_LOAD` on the first node that does not fit. Two settings under `"engine"` lower what a graph needs:

```
{ "id": "od",  "model": "yolov8-n_640x640", "engine": { "share": true, "buffers": 2 } },
{ "id": "od2", "model": "yolov8-n_640x640", "engine": { "share": true, "buffers": 2 } }
```

- `"share": true` : every node with it and the same model uses one engine. The nodes must have the same `params` and `buffers` . Results do not change.
- `"buffers": N` (1 to 100): how many inputs DX-RT keeps in flight for the model (DX-RT's default is 6). Most of a model's device memory is this buffer pool, so 2 buffers need about a third. Fewer buffers can lower async throughput; results do not change.

6.5 Decision Logic

Logic nodes are the decisions around the models: which boxes matter, where they are, how many, for how long, and when to raise an event. They run on the CPU, once per frame, and read their inputs on `"carry": "result"` edges.

Op	Reads	Main params	Outputs	Events
`filter`	boxes	`classes` , `min_score` , `min_area`	the boxes it keeps	-
`zone`	boxes	`polygon` (required), `mode` (`inside` / `outside`), `anchor` (`bottom` / `center`)	the boxes in the selected region	`enter` , `exit` (tracked boxes only)
`line`	tracked boxes			`cross`

© Copyright 2026 DEEPX All Rights Reserved.

Op	Reads	Main params	Outputs	Events
		`line` (required), `direction` (`both` / `in` / `out`)	counts `in` , `out` , `total`	
`count`	1 to 8 box inputs	`classes`	a total and a count per class	-
`dwell`	tracked boxes	`threshold_s` (default 10), `grace_s` (default 1)	seconds per track	`dwell`
`condition`	boxes, records, or per-crop scores or vectors	`value` , `comparator` , `threshold` , `reduce` , `cooldown_s` , `event`	`value` , `active`	its `event`
`gallery`	embeddings	`gallery` (required), `threshold` (default 0.5), `events` (`known` / `all` / `none` , default `known`)	a name per crop, or `unknown`	`match` ; `unknown` too with `"events": "all"`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Op	Reads	Main params	Outputs	Events
`user`	any one input	`name`, `input`, `output`	its input, unchanged	-

- **Coordinates** are fractions of the frame, 0 to 1, so one graph fits any resolution: `"polygon": [x0, y0, x1, y1, ...]` (at least 3 points), `"line": [x0, y0, x1, y1]`.
- **Track ids** come from a detector's `"track"`. `line` and `dwell` need them, and `zone` raises `enter` / `exit` only for tracked boxes; `--check` names the detector to add `"track"` to.
- **`condition`** compares one value (a box count, or a number from its input) with `threshold`, and raises its event when the comparison becomes true. With `cooldown_s` greater than 0 it raises it again at most every `cooldown_s` seconds while the comparison holds.
- **`gallery`** compares embeddings with a gallery file built once from a folder of images: `python3 scripts/build_gallery_database.py --variant <model> --images <folder> --out <file>.bin`.
- **`user`** passes its input through in a run. In an exported project (see *Standalone Project*) it becomes a C++ function you write.
- `--list-ops` lists every logic and cpu op with its inputs, its output and whether it needs track ids; `--list-ops --json` adds each op's events and parameters (types, defaults, ranges).

`zone_intrusion.json` raises an `intrusion` event when a person stands in the lower-right part of the frame:

```
{
 "version": 1,
 "name": "zone-intrusion",
 "nodes": [
  { "id": "cam",    "type": "source", "uri": "sample/img/sample_people.jpg" },
  { "id": "od",     "model": "yolov8-n_640x640",
    "params": { "score_threshold": 0.35 },
    "track":  { "algo": "iou", "iou": 0.3, "max_age": 30 } },
  { "id": "person", "type": "logic", "op": "filter", "params": { "classes": ["person"] } },
  { "id": "zone",   "type": "logic", "op": "zone",
    "params": { "polygon": [0.45, 0.35, 1.0, 0.35, 1.0, 1.0, 0.45, 1.0] } },
  { "id": "alarm",  "type": "logic", "op": "condition",
    "params": { "value": "count", "comparator": ">", "threshold": 0,
                "cooldown_s": 10, "event": "intrusion" } }
 ],
 "edges": [
  { "from": "cam",    "to": "od" },
  { "from": "od",     "to": "person", "carry": "result" },
  { "from": "person", "to": "zone",   "carry": "result" },
  { "from": "zone",   "to": "alarm",  "carry": "result" }
 ]
}
```


© Copyright 2026 DEEPX All Rights Reserved.

Events are printed on stdout after the frame's line, written one JSON object per line to `--events <file>`, and included in `--report`:

```
frame 0: od boxes=3 person boxes=2 zone boxes=1 alarm records=1 value=1 active=1
[EVENT] {"frame":0,"time":0,"stream":"cam","node":"zone","event":"enter","track_id":1,"data":{"count":1}}
[EVENT] {"frame":0,"time":0,"stream":"cam","node":"alarm","event":"intrusion","track_id":1,"data":{"value":1,"threshold":0,"track_ids":"1"}}
```

On the picture, zones are tinted and outlined, lines have an arrow toward their `in` side, counts and conditions appear in a panel at the top left, and the last five events are listed in a corner: bottom left, or another corner when that one would cover labels.

> **NOTE**
>
> A line has no hysteresis: a box whose anchor moves back and forth across the line counts a crossing each time. Place a line where people walk through, not where they stand.

6.6 Several Sources

A graph may have several source nodes. **Each source is its own stream**: the models are shared (one engine each, loaded once), but every stream has its own frames, tracks, logic state and output.

```
./bin/multi_model_graph_async \
    --graph src/cpp_example/multi_model_graph/multistream_od_two_sources.json \
    --input cam1=camera:0 --input cam2=assets/videos/dance-group.mov \
    --output run.avi --frames 300
```

- `--input <source_id>=<uri>` replaces one source's `uri`. A graph with several sources needs the `<source_id>=` form.
- `--output run.avi` writes one video per stream: `run_cam1.avi`, `run_cam2.avi`.
- `--display` shows one window with a grid of the streams, each labelled with its source id.
- `--frames N` stops after N frames of each stream.
- With several sources, every frame in `--report` and every event names its `stream`.

© Copyright 2026 DEEPX All Rights Reserved.

6.7 Command-Line Options

Option	Default	Description
`--graph <file>`	-	The graph to run
`--input [<source>=]<uri>`	each source's `uri`	An image, a video, `camera:<N>` or `rtsp://...`. Repeat once per source
`--output <file>`	none	The rendered result. `.mp4`, `.avi` or `.mkv`: a video (on Windows, `.mp4` or `.avi` only; `.mkv` fails with exit code 1). Other extensions: images (`<stem>_<index><ext>` for each frame of a video). A camera or RTSP source with image output needs `--frames N`
`--display`	off	Show each frame in a window; `q` or ESC stops the run. Needs a desktop session
`--draw <id,...>`	every node	Draw only these nodes; every node still runs and reports
`--report <file>`	none	Every frame's results as JSON
`--report-timing`	off	With `--report`: add the time per node to each frame
`--events <file>`	none	Every event of the logic nodes, one JSON object per line
`--output-layers <dir>`	none	Per frame, the clean frame and each node's drawing as separate transparent PNGs, with `layers.json`. A video, camera or RTSP source needs `--frames N`

© Copyright 2026 DEEPX All Rights Reserved.

Option	Default	Description
`--model-dir <dir>`	`assets/models`	Where the `.dxnn` files are
`--frames <N>`	all	Stop after N frames (of each stream)
`--loop`	off	Restart a video at its end, until Ctrl-C, `q`/ESC or `--frames N`
`--max-inflight <N>`	`16`	Frames in flight at once (`multi_model_graph_async` only)
`--max-jobs-per-stage <N>`	`0` (no limit)	Jobs one stage may have outstanding across frames (`multi_model_graph_async` only)
`--stall-timeout-ms <N>`	`0` (wait forever)	Fail when work is outstanding and nothing completes for N ms (`multi_model_graph_async` only)
`--check [<file>]`	-	Validate the graph and list its models without opening the NPU, then exit
`--list-models`	-	List the models; filter with `--consumes <roi, frame or either>` and `--produces <shape>`
`--list-ops`	-	List the logic and cpu ops
`--json`	off	With `--check`, `--list-models` or `--list-ops`: print JSON instead of a table
`--help`	-	Show usage

Exit codes: `0` success, `1` the graph or its models cannot run, or a frame failed, `2` a usage error.

Ctrl-C (and SIGTERM) stops gracefully: the frames in flight finish and `--report` and `--output` are finalized. A second Ctrl-C more than 200 ms later ends the process at once. If a run

© Copyright 2026 DEEPX All Rights Reserved.

may have to be killed, write `.avi` rather than `.mp4`, which cannot be played without its closing index (`.mkv` also survives a kill, but is not available on Windows).

`DXAPP_PROGRESS=1` prints a `[PROGRESS] frames=N fps=F avg_fps=A node_ms=<id>:<ms>,...` line about once a second.

6.8 Checking and Troubleshooting

Validate before running. `--check` parses and validates the graph, lists each node with its shapes and its `.dxnn` file, and names any file that is missing, without opening the NPU. A missing file does not make `--check` fail (a graph can be checked before its models are downloaded), but a file that cannot be loaded does: a file that is not a `.dxnn`, or a v9 file on DX-RT older than 3.5.0. Whether all the models fit in the NPU's memory shows only when they load.

Find a model for a slot. `--list-models` answers which model can go where:

```
./bin/multi_model_graph_async --list-models --consumes roi --produces scores
```

```
model                        task                produces   consumes   input     published file     ready
-----------------------------------------------------------------------------------------------------------
faceattr_resnetv1-18_218x178     face_attribute       scores     either     178x218    yes       v9
yes
alexnet_224x224                image_classification  scores     either     224x224    yes       v9       yes
...
```

(Columns narrowed to fit the page.) `file` shows whether the model is downloaded (`v8`, `v9`, `missing`), and `ready` says whether the graph engine can run it.

Error messages name where the problem is, what it is, and how to fix it:

```
ERROR [CODE] <where>: <what>
  -> <how>
```

Code	When
`GRAPH_SCHEMA`	Malformed JSON, a wrong type, a missing or unknown key, or a value out of range
`GRAPH_EDGE`	

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Code	When
	The shapes of an edge do not fit (for example `"seg" produces labelmap, not boxes` on an ROI edge), or a logic node's inputs are wrong
`GRAPH_ALIGN`	`"align": "face5"` from a producer without facial landmarks
`GRAPH_CYCLE` , `GRAPH_ORPHAN`	The graph has a cycle, or a node no source reaches
`GRAPH_VERSION` , `GRAPH_RESERVED`	An unsupported `version` , or something reserved for a later release (the `"prompt"` key, `"type": "fuse"`)
`MODEL_UNKNOWN`	The model name is not registered; the message suggests the closest name
`MODEL_NOT_READY`	The model is registered but cannot run in a graph (the anomaly-detection models in this release)
`MODEL_MISSING`	The `.dxnn` file is not in `--model-dir` ; the message gives the `download_models.py` command
`MODEL_LOAD`	The file is present but cannot be loaded: device memory (see *Device Memory*), device state, or a v9 file on DX-RT older than 3.5.0

A run that finds nothing says why. When an ROI edge produces no crops although its producer found boxes (for example `"classes": ["persons"]` when the detector emits `person`), the run prints a warning with the filter and the classes the producer actually emitted.

6.9 Python: dx_graph

`dx_graph` runs the same graph files from Python. It links the same engine as the CLI, so the reports, the rendered pictures and the error messages are the CLI's own.

```
import dx_graph

with dx_graph.Graph("src/cpp_example/multi_model_graph/cascade_od_reid_track.json") as g:
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
    for report in g.stream(source="my_video.mp4"):   # like --input; pipelined like the async CLI
        d = report.to_dict()                         # this frame's --report entry
        canvas = g.render(report)                    # this frame's --output image (numpy, BGR)
```

```
PYTHONPATH=bin/python python3 my_script.py
```

- A graph with several sources: `g.stream(sources={"cam1": a, "cam2": b})`; each report's `.stream` names its source.
- A model the device cannot load raises `dx_graph.GraphError` with `.code == "MODEL_LOAD"`.
- The executor options are the constructor's `max_frames_in_flight`, `max_jobs_per_stage` and `stall_timeout_ms`.
- Graphs are written in JSON, as for the CLI. Python cannot build or edit a graph, or add its own stages.

The full API is described in `src/bindings/python/dx_graph/README.md`.

6.10 Standalone Project

`scripts/export_graph_project.py` writes a graph as a CMake project that builds and runs wherever the DEEPX runtime is installed, with only the code of the graph's models. It gives the same events and the same picture as `multi_model_graph_async`.

```
python3 scripts/export_graph_project.py --graph my_graph.json --out ~/my_graph
cd ~/my_graph && ./setup.sh && ./run.sh      # results/result.jpg, results/events.jsonl
```

Option	Description
`--name <name>`	Project and program name (default: from the graph's `name`)
`--no-models`	Leave the `.dxnn` files out; the project's `setup.sh` downloads them
`--model-dir <dir>`	Where the `.dxnn` files are (default: `assets/models`)
`--dry-run`	Write nothing; with `--json`, print what would be exported and its size

© Copyright 2026 DEEPX All Rights Reserved.

A `user` logic node becomes a C++ class in `user_logic/<name>.cpp` with a `Run()` stub that you edit and rebuild (`cmake --build build`).

6.11 Limitations in This Release

Item	Status
Anomaly detection	The four anomaly-detection models are registered but cannot run in a graph (`MODEL_NOT_READY`)
3D / LiDAR models	Rejected in every graph: every source is a camera frame
Text-conditioned models (`"prompt"`) and model fusion (`"type": "fuse"`)	Reserved, rejected with `GRAPH_RESERVED`
CLIP prompt scores	Not a graph node: a CLIP image model outputs its image embedding
SuperPoint tracks	Not in a graph; the `descriptors` output is available
DnCNN image hand-off	The denoised image stays at 512×512, the model's resolution
`handoff_sr_od_cls.json`, `chain_sr_od_pose.json`	Not part of the DX-M1 sample runs (device-memory limit)
Windows, aarch64	Not built or tested on Windows; aarch64 is compile-checked only

ⓒ Copyright 2026 DEEPX All Rights Reserved.

7. DX-APP Post-processing

7.1 DX-APP C++ Post-processing

7.1.1 Overview

This directory serves as the centralized repository for **optimized C++ post-processing modules**. These libraries provide the critical decoding logic required to transform raw NPU tensor outputs into actionable, human-readable data.

Key Features

- **Performance Optimization:** Computational bottlenecks like Non-Maximum Suppression (NMS), segmentation mask resizing, and keypoint extraction are implemented in high-performance C++.
- **Python Bindings:** These libraries are consumed by the `pybind11` bindings (`dx_postprocess`), which enable the `*_cpp_postprocess.py` variants of the Python examples.
- **Modularity:** Each model family (YOLO, DeepLab, etc.) is isolated into its own module, allowing for targeted updates and lightweight linking.

7.1.2 Architecture & Implementation

At a high level, these libraries manage the transition from **DX-RT Tensors** to **Structured Data**.

Core Processing Steps

- **Step 1.** Tensor Decoding: Converting raw logits and offsets into coordinates (bounding boxes), confidence scores, and class IDs.
- **Step 2.** Filtering & NMS: Applying Non-Maximum Suppression to remove redundant overlapping detections and filtering results based on user-defined confidence thresholds.
- **Step 3.** Specialized Extraction
  : **Pose:** Extracting **x, y** coordinates and visibility scores for keypoint skeletons.
  : **Segmentation:** Decoding segmentation logits into per-pixel masks or class color maps.
- **Step 4.** Hardware Adaptation: Finalizing and scaling coordinates for models that use the **PPU** to handle the initial stages of post-processing.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Implementation Examples

The principles above are applied across various model families as follows

- `yolov5_postprocess` : Decodes YOLOv5 detection heads, runs NMS, and returns standard bounding boxes, scores, and class IDs.
- `yolov5pose_postprocess` : In addition to boxes and scores, it extracts per-person keypoints and their corresponding confidence levels for skeletal mapping.
- `deeplabv3_postprocess` : Applies argmax to the segmentation logits and returns a per-pixel class-ID mask at the model's output resolution (resizing and coloring are left to the caller).
- `*_ppu_postprocess` : Designed for models compiled with **PPU support**. These modules receive data that has already been partially processed by the NPU, adapting and finalizing the outputs for the host CPU.

Example-layer fast segmentation

The C++ example runtime also provides `FastSegmentationPostprocessor` in `src/cpp_example/common/processors/segmentation_postprocessor.hpp`. It is a generic low-resolution semantic-segmentation fast path that performs argmax before nearest-neighbor resize/crop. This class is used by generated examples through the `fast_segmentation` alias in `scripts/add_model_variant.py`; it is separate from the standalone `src/postprocess/` pybind library modules listed below.

Directory Structure

The libraries are organized by model architecture. Modules appended with `_ppu` are specialized for models where partial post-processing is offloaded to the **DEEPX PPU (Post-Processing Unit)**.

Module Components

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Each subdirectory follows a standardized pattern

- `*_postprocess.h` : Defines the class interface (e.g., `YOLOv5PostProcess`) and the result structures (e.g., `YOLOv5Result`).
- `*_postprocess.cpp` : Implementation of decoding algorithms and mathematical filters.
- `CMakeLists.txt` : present in the modules listed in `src/postprocess/CMakeLists.txt` ; `./build.sh --all --make_so` builds each of them as a standalone shared library (`lib/libdxapp_<module>_postprocess.so`). With the default `--minimal` build (or `--target` / `--category`), `--make_so` builds none of them.

```
src/postprocess/
├── CMakeLists.txt
├── README.md
├── anomaly/
├── attribute/
├── centerpose/
├── classification/
├── clip_text/
├── damoyolo/
├── deeplabv3/
├── depth/
├── dncnn/
├── dope/
├── efficientdet/
├── embedding/
├── espcn/
├── face3d/
├── hand_landmark/
├── mediapipe_hand/
├── nanodet/
├── obb/
├── picodet/
├── ppmatting/
├── preopt/
├── realesrgan/
├── retinaface/
├── rtdetr/
├── scrfd/
├── scrfd_ppu/
├── semantic_seg/
├── sfa3d/
├── ssd/
├── superpoint/
├── ulfgfd/
├── vitpose/
├── yolact/
├── yolopv2/
├── yolov4/
├── yolov5/
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
├── yolov5_ppu/
├── yolov5face/
├── yolov5pose/
├── yolov5pose_ppu/
├── yolov5seg/
├── yolov7/
├── yolov7_ppu/
├── yolov8/
├── yolov8_ppu/
├── yolov8pose/
├── yolov8seg/
├── yolov9/
├── yolov10/
├── yolov11/
├── yolov12/
├── yolov26/
├── yolox/
├── yolox_ppu/
├── yolov3tiny_ppu/
└── zero_dce/
```

7.1.3 Cross-Language Integration

Usage Methods

The SDK is designed so that C++ and Python developers utilize the same underlying engine, ensuring a seamless transition from prototyping to production.

Method A. C++ Usage

Include the header and instantiate the class directly within your inference loop.

```cpp
#include "yolov5_postprocess.h"

YOLOv5PostProcess post_processor(input_w, input_h, obj_threshold, score_threshold,
                                 nms_threshold, is_ort_configured);
auto final_results = post_processor.postprocess(npu_output_tensors);  //
std::vector<YOLOv5Result>
```

Method B. Python Usage

Import the `pybind11` wrapper via the `dx_postprocess` module to access the same C++ performance.

```python
from dx_postprocess import YOLOv5PostProcess
```


© Copyright 2026 DEEPX All Rights Reserved.

```
post_processor = YOLOv5PostProcess(input_w, input_h, obj_threshold, score_threshold,
                                   nms_threshold, is_ort_configured)
final_results = post_processor.postprocess(ie_output)
```

Build Configuration

Compilation Guide

The sources in `src/postprocess/` are compiled into the `dx_postprocess` Python module when `./build.sh` installs it (default build) or when you run `python3 -m pip install .` in `src/bindings/python/dx_postprocess/`.

```
# Execute from the dx_app/ root directory
./build.sh
```

`./build.sh --all --make_so` additionally builds the modules listed in `src/postprocess/CMakeLists.txt` as standalone shared libraries in `lib/`. `--make_so` has no effect on the default `--minimal`, `--target` or `--category` builds. The C++ examples do not link these libraries. They use `src/cpp_example/common/processors/`, except for `sfa3d`, `superpoint`, `dope` and `yolopv2`, whose sources are compiled directly into the C++ examples.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

7.2 DX-APP Python Post-processing

7.2.1 Overview

The `dx_postprocess` module provides high-performance Python bindings for the **DEEPX C++ post-processing library**. By utilizing `pybind11`, it allows Python applications to execute hardware-optimized decoding logic with near-native C++ performance.

Performance Benefits

Integrating C++ post-processing into your Python pipeline offers two primary advantages

- **Stage Acceleration:** The post-processing operation itself (e.g., NMS, coordinate scaling) is significantly faster than native Python implementations.
- **Pipeline-Wide Improvement:** End-to-end FPS gains occur in specific scenarios.
  - **Primary Bottleneck:** When the post-processing stage is the slowest part of the pipeline.
  - **CPU Contention:** In CPU-constrained environments (e.g., embedded boards), reduced CPU usage in the post-processing stage helps other stages (Read, Preprocess) and NPU utilization function more efficiently.

> **NOTE**
>
> While the C++ library is faster, overall pipeline improvement depends on the bottleneck location. If NPU inference or data reading is the limiting factor, end-to-end FPS gains may be minimal, though CPU power consumption will still decrease.

For end-to-end Python example usage, refer to DX-APP Python Example Usage Guide.

7.2.2 Technical Strategy & Support

Technical Strategy

© Copyright 2026 DEEPX All Rights Reserved.

The SDK follows a **Single Source of Truth** strategy to ensure reliability across platforms:

- **Implementation:** C++ post-processing classes are wrapped with `pybind11` for seamless Python integration.
- **Unified Logic:** It directly wraps the C++ source in `src/postprocess/`, so a class used from Python runs the same decoding as that C++ library.
- **Build System:** Utilizes **CMake** with **scikit-build-core** for robust, cross-platform extension building.
- **Source Code Locations:**
  - **Python Bindings:** `src/bindings/python/dx_postprocess/postprocess_pybinding.cpp`
  - **C++ Implementations:** `src/postprocess/`

Supported Models & Tasks

ⓒ Copyright 2026 DEEPX All Rights Reserved.

The library wraps **58 C++ post-processing classes** with pybind11 bindings, ensuring consistent results between C++ and Python implementations.

- **Object Detection:** YOLOv4, YOLOv5, YOLOv7, YOLOv8, YOLOv9, YOLOv10, YOLOv11, YOLOv12, YOLOv26, YOLOX, NanoDet, DAMOYOLO, SSD, CenterPose, EfficientDet, RT-DETR (v1/v2/v3), PP-PicoDet (PP-ShiTu mainbody), YOLO26 pre-optimized (top-k table; detection and pose)
- **Face Detection:** SCRFD, YOLOv5Face, RetinaFace, ULFGFD
- **Face Landmark:** Face3D (3DDFA-v2)
- **Pose Estimation:** YOLOv5Pose, YOLOv8Pose, VitPose, CenterPose
- **Keypoint Detection:** SuperPoint
- **Object Pose Estimation:** DOPE
- **3D Object Detection:** SFA3D
- **Panoptic Driving Perception:** YOLOPv2
- **Hand Detection:** MediaPipe Hand
- **Semantic Segmentation:** DeepLabV3, SemanticSeg (BiSeNet/SegFormer)
- **Instance Segmentation:** YOLOv5Seg, YOLOv8Seg, YOLACT, mask-RT-DETR
- **Image Matting:** PP-Matting (alpha matte at the model resolution; the examples resize it)
- **OBB Detection:** YOLOv26-OBB
- **Classification:** ClassificationPostProcess (all image-classification families, e.g. EfficientNet, ResNet, RegNet)
- **Attribute Recognition:** AttributePostProcess (person attribute, face attribute)
- **Depth Estimation:** DepthPostProcess (FastDepth, SCDepth, Depth Anything V2, YOLO26-depth)
- **Image Denoising:** DnCNN
- **Super Resolution:** ESPCN, RealESRGAN
- **Image Enhancement:** Zero-DCE
- **Embedding:** ArcFace, CLIP text encoder
- **Hand Landmark:** HandLandmark
- **Anomaly Detection:** EfficientAD (teacher + student + autoencoder outputs), AnomalyFeature (PatchCore backbone response). Both return the anomaly map at feature-map resolution; the Python examples resize it to the image with OpenCV.
- **PPU Variants:** YOLOv5-PPU, YOLOv7-PPU, YOLOv8-PPU, SCRFD-PPU, YOLOv5Pose-PPU, YOLOX-PPU, YOLOv3Tiny-PPU. `SCRFD-PPU` and `YOLOv5Pose-PPU` are postprocess classes only; `SCRFD500M_PPU` and `YOLOV5Pose_PPU` are not registered examples.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Generic fast segmentation in Python examples

`FastSegmentationPostprocessor` is available from `common.processors` for Python example-layer semantic segmentation. It is intentionally generic and selected by the `fast_segmentation` generation alias. It is not a direct `dx_postprocess` pybind class; use it from `src/python_example/common/processors/fast_segmentation_postprocessor.py` when the model outputs low-resolution logits or class maps that should be argmaxed before resize.

7.2.3 Installation & Setup

Prerequisites

Before building, ensure the following are installed on your system

- **Python:** 3.8 or higher
- **Compiler:** C++14-compatible (GCC 7.5+, Clang, etc.)
- **Build Tool:** CMake 3.16 or higher

`pybind11` is installed by pip as a build dependency (`pyproject.toml`) and found with `find_package(pybind11)`.

Installation Steps

The module is automatically built during the standard SDK build, but it can also be installed independently from the binding directory.

Method	Command	Recommended For
Default Build (minimal)	`./build.sh`	Standard developer build; builds run_demo C++ targets and installs dx_postprocess
Full Build	`./build.sh --all`	Full SDK deployment (all targets) — requires explicit `--all` flag
Standalone	`cd ./src/bindings/python/dx_postprocess && python3 -m pip install .`	Focused updates to the post-processing logic

© Copyright 2026 DEEPX All Rights Reserved.

Standalone Installation Notes

The standalone installation uses the local `pyproject.toml` and the configured Python packaging backend. Ensure your active Python environment has access to the required build dependencies before running `python3 -m pip install .`.

Verification

To ensure the module is correctly linked to your Python environment, run:

```
python3 -c "import dx_postprocess; print('dx_postprocess successfully installed!')"
```

7.2.4 Usage Guide & Running Examples

The `dx_postprocess` classes expect a list of NumPy arrays directly from the `InferenceEngine`.

Running Examples

```
# Python examples in src/python_example/ utilize this library in their '_cpp_postprocess.py' variants.
# From dx_app/ directory

# Sync (Image Inference)
python src/python_example/object_detection/yolov9/yolov9-s_640x640/yolov9-
s_640x640_sync_cpp_postprocess.py --image sample/img/sample_kitchen.jpg

# Async (Stream Inference)
python src/python_example/object_detection/yolov9/yolov9-s_640x640/yolov9-
s_640x640_async_cpp_postprocess.py --video assets/videos/dance-group.mov
```

Each `*_cpp_postprocess.py` script defines an `on_engine_init(runner)` callback and passes it to `run_entry()`. Once the engine reports the model input size, the callback creates the model's `dx_postprocess` class (for example `YOLOv8PostProcess` or `ClassificationPostProcess`) and sets it as `runner._cpp_postprocessor`, so the C++ class replaces the Python postprocessor. `dx_postprocess` is imported inside the callback, so `--help` works before the module is built.

Direct API Call

```
from dx_postprocess import YOLOv9PostProcess
import numpy as np

# Initialize the optimized C++ post-processor
postprocessor = YOLOv9PostProcess(
input_w=640,
```


© Copyright 2026 DEEPX All Rights Reserved.

```
input_h=640,
score_threshold=0.25,
nms_threshold=0.45,
is_ort_configured=True
)

# Post-process inference output
# ie_output is a list of numpy arrays from InferenceEngine
detections = postprocessor.postprocess(ie_output)

# detections is a numpy array with shape (N, 6)
# Each row: [x1, y1, x2, y2, confidence, class_id]
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

8. DX-APP Architecture and Usage Reference

This chapter is the reference for the whole repository: how it is laid out, how an example is built from the shared runtime layer, the command-line options and environment variables every example shares, and the runtime features built into all of them.

To run a first example, follow DX-APP Quick Start. Requirements and installation are in DX-APP Installation and Build.

8.1 Core Concepts & Architecture

8.1.1 Repository Layout

The project is structured to separate core logic from language-specific implementations.

```
dx_app/
├── src/
│   ├── cpp_example/          # C++ end-to-end examples (organized across 29 tasks)
│   │   ├── multi_model_graph/  # Multi-model graph CLI and sample graph JSON files
│   │   └── common/           # ← Shared C++ runtime layer
│   │       ├── base/         #   Abstract interfaces (task factories such as IDetectionFactory,
IPreprocessor, IPostprocessor, IVisualizer, IInputSource)
│   │       ├── processors/   #   Shared processors
│   │       ├── runner/       #   Task-specific sync/async runner pairs
│   │       ├── inputs/       #   Image/Video/Camera/RTSP input sources
│   │       ├── visualizers/  #   Task-specific visualizers
│   │       ├── config/       #   ModelConfig loader
│   │       ├── utility/      #   Labels, preprocessing, profiling, run_dir, interrupt handling,
verify_serialize
│   │       ├── graph/        #   Multi-model graph engine
│   │       ├── registry/     #   Model registry used by the graph engine
│   │       ├── trackers/     #   IoU tracker
│   │       └── third_party/  #   Header-only third-party libraries (nlohmann_json)
│   ├── python_example/       # Python end-to-end examples (organized across 29 tasks)
│   │   └── common/           # ← Shared Python runtime layer
│   │       ├── base/         #   Abstract interfaces (task factories such as IDetectionFactory,
IPreprocessor, IPostprocessor, IVisualizer, IInputSource)
│   │       ├── processors/   #   Shared processors
│   │       ├── runner/       #   SyncRunner, AsyncRunner, run_dir, verify_serialize, sr_tiling, args
│   │       ├── inputs/       #   Image/Video/Camera/RTSP input sources
│   │       ├── visualizers/  #   Task-specific visualizers
│   │       ├── config/       #   ModelConfig loader
│   │       ├── trackers/     #   IoU tracker
```


© Copyright 2026 DEEPX All Rights Reserved.

```
│   │       └── utility/       #   Labels, preprocessing, profiling
│   ├── postprocess/          # C++ post-processing (consumed by pybind11 bindings)
│   ├── utility/              # Shared support code used by build flow
│   └── bindings/
│       └── python/
│           ├── dx_postprocess/ # pybind11 bindings wrapping src/postprocess/
│           └── dx_graph/       # pybind11 module running multi-model graphs
├── config/
│   ├── model_registry.json     # Model registry — single source of truth
│   ├── test_models.conf        # Test model configuration
│   └── README.md               # Config directory documentation
├── scripts/                  # Developer tools, validation, and helper scripts
├── tests/                    # pytest-based test suites
│   ├── test_helpers/           #   Shared test constants & utilities
│   ├── cpp_example/            #   C++ tests (CLI, E2E, visualization, features)
│   └── python_example/         #   Python tests (unit, CLI, E2E, visualization)
├── assets/                   # Downloaded models/videos (via setup.sh)
├── build.sh                  # Top-level build script
├── run_tc.sh                 # Unified test runner for example tests
├── install.sh                # Dependency and OpenCV installer
└── docs/                     # Detailed documentation
```

For contributor-oriented layout details, refer to DX-APP Example Source Structure.

> **User vs Contributor Guidance**
>
> This chapter is primarily a user-facing reference. If you are extending examples, onboarding new models, or maintaining the repository structure, use the contributor-oriented documents in *DX-APP Developer Guide*.

8.1.2 Pipeline Architecture

C++ Application Templates (`src/cpp_example/`)

These templates provide high-performance, production-ready references for building applications using the DX-RT C++ API.

The C++ tree is organized by **task** → **model family** → **variant**, with a shared `common/` layer providing base interfaces, shared processors, task-specific runner pairs, visualizers, and input abstraction. Each model directory delegates to `common/` via the factory pattern. For details, refer to DX-APP C++ Usage Guide and DX-APP Example Source Structure.

Pipeline Architecture

© Copyright 2026 DEEPX All Rights Reserved.

Each template follows a self-contained pipeline designed for modularity

- **Step 1. Input:** Image, Video, Camera, or RTSP stream (via `common/inputs/`).
- **Step 2. Pre-process:** Resizing and normalization (via the preprocessors in `common/processors/`, which use the letterbox helpers in `common/utility/`).
- **Step 3. Inference:** Execution on the NPU via **DX-RT**.
- **Step 4. Post-process:** Call to shared C++ processors in `common/processors/` (e.g., NMS, box scaling).
- **Step 5. Output:** Result rendering via `common/visualizers/` (Display) or storage (Save).

8.1.3 Post-processing Libraries (`src/postprocess/`)

These libraries transform raw NPU output tensors into structured, actionable data. They are **consumed by the pybind11 bindings** (`src/bindings/python/dx_postprocess/`) to enable `*_cpp_postprocess.py` variants in Python.

> **NOTE**
>
> The C++ examples under `src/cpp_example/` do **not** link the `src/postprocess/` libraries; they use their own shared processors in `src/cpp_example/common/processors/`. The exceptions are four modules (`sfa3d`, `superpoint`, `dope`, `yolopv2`), whose sources are compiled directly into the C++ examples and are also exposed through the pybind11 bindings.

Module Structure

The library is organized into model-specific subdirectories (e.g., `yolov5/, yolov8/, deeplabv3/`), each containing

- `*_postprocess.h`: Defines the post-processing class (e.g., `YOLOv5PostProcess`) and standard result structures (e.g., `YOLOv5Result`).
- `*_postprocess.cpp`: Contains the optimized implementation for decoding, coordinate scaling, and filtering.
- `CMakeLists.txt`: Present in the modules listed in `src/postprocess/CMakeLists.txt`; `./build.sh --all --make_so` builds those modules as standalone shared libraries (`lib/libdxapp_<module>_postprocess.so`).

Functional Responsibilities

© Copyright 2026 DEEPX All Rights Reserved.

The libraries handle the heavy computational load required after the inference stage

- **Tensor Decoding:** Converting raw NPU buffer outputs into human-readable results such as bounding boxes, confidence scores, and class IDs.
- **Advanced Geometry:** Extracting keypoints for pose estimation or skeletons.
- **Mask Generation:** Processing multi-dimensional tensors into segmentation masks.
- **Filtering & Optimization:** Applying algorithms like **Non-Maximum Suppression (NMS)** and threshold-based filtering to remove redundant detections.

Cross-Language Integration

- **For Python Developers:** The `*_cpp_postprocess.py` variants use these C++ libraries via the `dx_postprocess` pybind11 module, achieving near-native performance.
- **For C++ Developers:** The C++ examples use their own shared processors in `src/cpp_example/common/processors/`, which are compiled and linked directly.

8.2 Usage Reference

8.2.1 CLI Reference

All C++ and Python examples share a consistent set of command-line arguments.

Common Arguments

Flag	C++	Python	Description
`-m` / `--model`	`-m` / `--model_path`	`-m` / `--model`	Path to `.dxnn` model file. Optional: if omitted, the example's default model is used and auto-downloaded when missing
`-i` / `--image`	`-i` / `--image_path`	`-i` / `--image`	Input image file or directory
`-v` / `--video`	`-v` / `--video_path`	`-v` / `--video`	Input video file
`-c` / `--camera`	`-c` / `--camera_index`	`-c` / `--camera`	Camera device index
`-r` / `--rtsp`	`-r` / `--rtsp_url`	`-r` / `--rtsp`	RTSP stream URL

© Copyright 2026 DEEPX All Rights Reserved.

Flag	C++	Python	Description
`-l` / `--loop`	`-l` / `--loop` (default: auto)	`-l` / `--loop` (default: 1; bare `--loop` = 2)	Inference repeat count
`--no-display`	`--no-display`	`--no-display`	Disable visualization window
`-s` / `--save`	`-s` / `--save`	`-s` / `--save`	Save rendered output to run directory
`--save-dir`	`--save-dir`	`--save-dir`	Base output directory (default: `artifacts/cpp_example` under the current directory for C++, `artifacts/python_example/` under the dx_app root for Python)
`--dump-tensors`	`--dump-tensors`	`--dump-tensors`	Dump raw input/output tensors to files
`--show-log`	`--show-log`	`--show-log`	Enable verbose log output (default: quiet)
`--drop-frames`	`--drop-frames`	—	C++ only: preview drops stale frames instead of pacing the pipeline (not available in the classification sync runner or the embedding runners)
`--config`	`--config`	`--config`	Model config JSON path. Python: auto-detected if omitted (`config.json` next to the model, then next to the script). C++: read only when given; otherwise the factory's

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Flag	C++	Python	Description
			built-in defaults are used
`-h` / `--help`	`-h` / `--help`	`-h` / `--help`	Show usage

- **Input Source Rule:** `--image`, `--video`, `--camera`, and `--rtsp` form a mutually exclusive group. If none is specified, a **default sample image** is automatically selected based on the task type. Image-only examples (embedding, re-ID, attribute recognition) accept only `--image`.

Environment Variables

Variable	Description
`DXAPP_SAVE_IMAGE`	When set to a file path, saves the visualization output to that path (no `--save` required)
`DXAPP_VERIFY`	When set to `1`, dumps post-processing results to `logs/verify/{model}.json` (last frame) and `{model}.frames.jsonl` (every frame, in input order) for numerical verification
`DXAPP_VERIFY_DIR`	Directory for `DXAPP_VERIFY` output (default `logs/verify`)
`DXAPP_PROGRESS`	When set to `1`, prints `[PROGRESS] frames=N fps=F avg_fps=A read_ms=R pre_ms=P infer_ms=I post_ms=Q` at most once a second and once at exit, for front ends such as DX AI Studio: `fps` is the last second's rate, `avg_fps` counts from the first frame (start-up and model load excluded), `*_ms` are the stage averages over the same second (the final line: run averages). The multi_model_graph binaries print `node_ms=<id>:<ms>,...` (time per graph node) instead of the `*_ms` stage fields

8.2.2 C++ Templates

© Copyright 2026 DEEPX All Rights Reserved.

To help developers optimize for specific hardware targets, templates are provided in two execution patterns

- **Synchronous (`xx_sync.cpp`):**

  Logic: A single-threaded, sequential loop (**Input** → **Inference** → **Output**).

  Use Case: Best for single-image processing and simplified debugging.

- **Asynchronous (`xx_async.cpp`):**

  Logic: Uses multi-threading and the `RunAsync()` API to overlap stages. While the NPU performs inference on Frame **N**, the CPU simultaneously handles pre-processing for Frame **N+1** and post-processing for Frame **N-1**.

  Use Case: Essential for maximizing **FPS** on live video streams and ensuring high NPU utilization.

8.2.3 Python Integration (Bindings & Examples)

DX-APP provides a unified environment that combines the rapid development of Python with the high performance of native C++.

High-Performance C++ Bindings (`dx_postprocess`)

To eliminate post-inference bottlenecks, DX-APP provides optimized C++ logic exposed via `pybind11` .

- **Key Capabilities:** Handles CPU-intensive tasks such as NMS (Non-Maximum Suppression), tensor decoding, and mask generation at native speeds.
- **Scope:** Used by the `*_cpp_postprocess.py` variants. The C++ examples use their own shared processors under `src/cpp_example/common/processors/` .
- **Installation:** Automatically compiled during `./build.sh` .
  : **Manual install:** `cd src/bindings/python/dx_postprocess && python3 -m pip install .`

For detailed usage examples and API references, please refer to the documentation in **Section. DX-APP Python Post-processing**

Application Examples (`src/python_example/`)

These templates utilize `dx_engine` (for inference) and `dx_postprocess` (for acceleration). Users can choose from four variants depending on their performance requirements.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

The Python tree is organized by **task** → **model family** → **variant**, with a shared `common/` layer providing base interfaces, shared processors, generic sync/async runners, visualizers, and input abstraction — the same factory-based architecture as the C++ side. For structure and contributor-facing rules, refer to DX-APP Python Usage Guide and DX-APP Example Source Structure.

Task-Based Structure

Templates are categorized by task across multiple task directories. All examples share the `common/` runtime layer for processors, runners, and visualizers. Representative tasks:

- **Image Classification:** EfficientNet, AlexNet, ResNet, MobileNet, etc.
- **Object Detection:** YOLOv5/v7/v8/v9/v10/v11/v12/v13, YOLOX, NanoDet, DAMOYOLO, SSD (PPU variants under `yolo_ppu/`)
- **Face Detection:** SCRFD, YOLOv5Face, YOLOv7Face, RetinaFace
- **Pose Estimation:** YOLOv8-Pose
- **Segmentation:** BiSeNet, DeepLabV3+, SegFormer, YOLOv8Seg
- **Depth Estimation, Face Recognition, Oriented Object Detection, Image Denoising, Low-Light Enhancement, Super Resolution, Hand Landmark, Person Attribute, Person Re-ID**

Functional Variants

Variant	Post-processing	Threading Model	Recommendation
`*_sync.py`	Pure Python	Synchronous	Learning & Logic Debugging
`*_async.py`	Pure Python	Asynchronous	Basic performance optimization
`*_sync_cpp_postprocess.py`	C++ Binding	Synchronous	Accelerating heavy CPU tasks
`*_async_cpp_postprocess.py`	C++ Binding	Asynchronous	Maximum FPS (Recommended)

8.3 Advanced Features

DX-APP includes several production-oriented features built into all templates.

© Copyright 2026 DEEPX All Rights Reserved.

Auto-Download

When an example (C++ or Python) runs without `-m`, the runner uses its default model from `config/model_registry.json` and, if the file is missing, downloads it via `setup_sample_models.sh`. If the download fails, an error message with manual download instructions is displayed. An explicit `-m` path that does not exist is an error (no download). Missing input files, including videos, are not downloaded; run `./setup.sh` or `./setup_sample_videos.sh` first.

Default Input Fallback

If no input source (`--image`, `--video`, `--camera`, `--rtsp`) is provided, the runner automatically selects a **default sample image** appropriate for the task type. A log message indicates which default was applied.

Signal Handling

C++ runners and the `multi_model_graph_*` CLI register SIGINT/SIGTERM handlers for graceful shutdown. Pressing Ctrl+C during inference cleanly exits, releasing all resources: a runner prints `"Interrupted by user"`, and the graph CLI finishes the frames in flight in every stream and finalizes `--report` / `--output` / `--events`. A later Ctrl+C, more than 200 ms after the first, terminates the process at once (for a shutdown that is itself stuck); repeats within 200 ms count as the same request. SIGTERM is always graceful, however often it arrives. Python runners behave the same way (`common/runner/interrupts.py`); see the Python usage guide.

Run Directory (`--save` / `--save-dir`)

When `--save` is enabled, a timestamped run directory is created:

```
artifacts/cpp_example/
  {model}_sync-image-{name}-{YYYYMMDD-HHMMSS}/
    run_info.txt        # Metadata (script, model, input paths)
    output.jpg          # Saved visualization (image mode)
    output.mp4          # Saved visualization (video mode; output.avi (MJPG) if no mp4 encoder)
    dump_tensors/       # (if --dump-tensors) raw tensor files
```

Video, camera and RTSP runs are named `{model}_sync-stream-{name}-...`, and a directory input `...-image-dir-...`. The Python runners use the same pattern under `artifacts/python_example/`, prefixed with the script name; they save images as `{image}_result.jpg` and dump tensors to `tensors/` as `.npy` files.

Numerical Verification (`DXAPP_VERIFY`)

© Copyright 2026 DEEPX All Rights Reserved.

A complete pipeline for serializing inference results for inspection and debugging:

- (1) Set `DXAPP_VERIFY=1` before running any example
- (2) Post-processing results are serialized to `logs/verify/{model}.json` (last frame) and `{model}.frames.jsonl` (every frame, in input order)
- (3) Supports every result type (Detection, Classification, Pose, Segmentation, Panoptic, 3D, SuperPoint keypoints, …)

Tensor Dump (`--dump-tensors`)

Dumps raw input/output tensors for debugging. On exception, tensors are auto-dumped (C++: sync runners, `.bin` files; Python: `.npy` files plus a `reason.txt`).

Model Config (`--config`)

Runtime parameters (score threshold, NMS threshold, top-k) live in `<task>/<family>/<variant>/config.json` , next to that variant's entry. The Python examples load it automatically. A C++ example reads it only when you pass `--config <path>` ; otherwise it uses the defaults compiled into its factory. A single-model extract may keep `config.json` beside the entry script.

Version Compatibility

Every example in this release, C++ and Python, needs **DX-RT 3.5.0 or later**: the DX Model Zoo 2_5_0 files that `./setup.sh` downloads are `.dxnn` container v9, and DX-RT 3.5.0 is the first runtime that loads them. Check the runtime before you run an example:

```
dxcli --version   # the first line must be DXRT v3.5.0 or later
```

The runners also check versions when they start, but the C++ and Python checks differ:

Language	What the runner checks	What an older DX-RT shows
C++	DX-RT ≥ 3.5.0 and model format ≥ v7 (`minversionforRTandCompiler` in `common/utility/common_util.hpp`), once the engine is created	`DXRT library version is too low. (required: >= 3.5.0, current: <version>)` , or DX-RT's own error on the `.dxnn` if loading fails first
Python	DX-RT ≥ 3.0.0 only, and a warning for a model format older than v7	DX-RT 3.0.0 to 3.4.x pass the check, and the example then fails while loading the model,

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Language	What the runner checks	What an older DX-RT shows
		with DX-RT's own error (for example `Model file format version 9 is not supported`)

> ⚠ **Python examples on DX-RT 3.0.0 to 3.4.x**
>
> The Python runners do not stop on DX-RT 3.0.0 to 3.4.x, and the error you see when the model loads comes from DX-RT, not from DX-APP. If a Python example fails while loading a model, check `dxcli --version` first and upgrade DX-RT to 3.5.0 or later. `import dx_engine` fails with `dx_engine and DX-RT versions do not match` when the `dx_engine` package and the runtime differ in major.minor version.

The multi-model graph checks the `.dxnn` container itself before it loads a model and names the required DX-RT version (see DX-APP Multi-Model Graph). The requirements are listed in *Software Requirements on Linux* in DX-APP Installation and Build.

Headless Mode

C++ and Python runners detect the absence of `DISPLAY` / `WAYLAND_DISPLAY` and skip the preview window automatically. Use `--no-display` for explicit headless operation.

© Copyright 2026 DEEPX All Rights Reserved.

9. DX-APP Developer Guide

9.1 DX-APP DX Tool Guide

This guide is intended for contributors and maintainers working on the DX-APP example repository.

`./scripts/dx_tool.sh` is the unified developer entry point for repetitive example maintenance tasks such as model onboarding, package extraction, example discovery, validation, execution, and benchmarking.

9.1.1 Overview

The tool provides both:

- **Interactive mode** for guided workflows
- **Command mode** for repeatable developer operations and automation

Primary script:

- `./scripts/dx_tool.sh`

Related helper scripts:

- `./scripts/add_model.sh`
- `./scripts/extract_model_package.sh`
- `./scripts/extract_sln_package.bat`
- `./scripts/validate_models.sh`
- `./scripts/run_examples.sh`
- `./scripts/bench_models.sh`

Key data files:

- `./config/model_registry.json` — model registry, single source of truth

When to Use `dx_tool.sh`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Use `dx_tool.sh` when you need to:

- add a new example model to the repository structure
- inspect existing models by task or keyword
- export an example as a standalone package for deployment
- validate example assets and generated code structure
- run a filtered subset of examples
- benchmark examples across C++ and Python variants

For end-user installation, setup, and basic inference execution, refer to the installation and usage documents instead of this guide.

9.1.2 Quick Start for Contributors

Step 1. Full Developer Sequence

If you are adding or refactoring an example, follow this standard sequence to ensure everything is built and tested correctly

```
# 1. Prepare assets & Build
./setup.sh
./build.sh --clean --all

# 2. Inspect & Validate repository state
./scripts/dx_tool.sh list
./scripts/dx_tool.sh validate

# 3. Run and Verify (C++ and/or Python)
./scripts/dx_tool.sh run --lang both
./run_tc.sh --cpp --cli
./run_tc.sh --python
```

This sequence prepares assets, rebuilds the repository, inspects available models, runs registry-driven validation, and performs an example-oriented test pass.

Step 2. Execution Modes

Interactive mode

Run the tool with no subcommand to enter the guided menu.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
./scripts/dx_tool.sh
```

This mode is useful when:

- you are exploring the repository for the first time
- you do not remember the exact subcommand
- you want menu-based task selection

> **NOTE**
>
> `dx_tool.sh run` with no arguments delegates to `scripts/run_examples.sh` interactive mode, which provides a 6-stage guided menu (language, category, model filter, sync/async, input type, display/save options) with a configuration summary before execution. Each test also displays its performance table.

Command mode

Run a subcommand directly when you already know the intended task.

```
./scripts/dx_tool.sh list
./scripts/dx_tool.sh search yolov9
./scripts/dx_tool.sh validate
```

`validate` prompts for the language and scope; for scripts, call `./scripts/validate_models.sh` directly.

This mode is better for:

- repeatable contributor workflows
- shell history reuse
- scripting and CI-friendly operations

Step 3. Command Summary

Command	Purpose
`add`	Create a new model/example skeleton
`extract`	Export an example as a standalone package
`list`	List example models by task (C++/Python)

© Copyright 2026 DEEPX All Rights Reserved.

Command	Purpose
`search`	Search models by keyword
`info`	Show model details
`delete`	Delete example directories (`<task>/<family>/<variant>` , or a bare family name = the whole family); registry and `test_models.conf` rows are left in place
`new-task`	Generate skeleton `common/` files (postprocessor, visualizer, factory interface, sync/async runner) for a new task type
`validate`	Run registry-driven validation via `validate_models.sh` : build and run (NPU inference) every supported registry model, generating any that has no example yet
`run`	Execute examples with filters
`bench`	Benchmark examples with filters
`help`	Show usage help

9.1.3 Core Capabilities & Reference

Model Management

Model Registry

`config/model_registry.json` is a JSON array that serves as the single source of truth for all model metadata. Each entry contains:

Field	Purpose
`model_name`	Registry key used by `./setup.sh --models` (e.g., `yolov9s`)
`variant`	Example directory and `.dxnn` stem (e.g., `yolov9-s_640x640`)

© Copyright 2026 DEEPX All Rights Reserved.

Field	Purpose
`family`	Family directory (e.g., `yolov9`)
`task`	Task directory (e.g., `object_detection`)
`dxnn_file`	Compiled model filename (e.g., `yolov9-s_640x640.dxnn`)
`add_model_task`	Legacy task name passed to `add_model.sh` by `validate_models.sh` (e.g., `object_detection`, `classification`, `ppu`)
`postprocessor`	Which shared processor to use (e.g., `yolov8`)
`input_width`, `input_height`	Model input dimensions
`config`	Extra parameters (thresholds, num_classes, etc.)
`supported`	Whether the model is part of the standard validation flow
`alias_of`, `published`, `image_only`, ...	Alias, publication and test metadata

The `add` command copies a reference example picked from this registry and rewrites its factory files, `config.json`, and entry-point scripts for the new variant.

Adding Examples

- Use `add` when you are onboarding a new model into an existing task category.
- Use `new-task` when you are introducing a new task-level grouping that does not yet exist in the current repository layout.

In both cases, validate the result. `add_model.sh` registers the new variant in `model_registry.json` and `test_models.conf` automatically (unless `--no-register` is given). For a new task, check whether the test helpers need the new task as well.

Validation & Verification

Basic validation

© Copyright 2026 DEEPX All Rights Reserved.

```
./scripts/dx_tool.sh validate
# or directly:
./scripts/validate_models.sh --lang py
```

This
builds and runs (NPU inference) the example of every supported registry row whose `.dxnn` is present; a row without an example is generated first. With `--skip-verify` only missing examples are generated.

`validate_models.sh` options

Option	Purpose
`--lang <lang>`	Language filter: `cpp`, `py` or `both`
`--skip-verify`	Code generation only (no inference)
`--no-video`	Image-only mode
`--list`	Print commands without executing
`--clean`	Legacy cleanup for the pre-v3.3.0 flat layout. **Do not run it on the current tree**: it deletes every `<task>/<family>/` directory whose name is not a registry `model_name`, including committed examples
`--start-from <model_name>`	Resume from a registry `model_name`
`<task_filter>`	Filter by registry `add_model_task` (e.g., `object_detection`, `classification`, `pose`, `obb`, `ppu`); task directory names such as `image_classification` are rejected

Execution & Benchmarking

- **Selective Run**: The `run` command abstracts the complexity of finding specific binaries. It uses filters to execute subsets of the 497 registered models instantly.
- **Performance Benchmarking**: The `bench` command runs models in a loop, calculating average latency and throughput (FPS) while minimizing system noise.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

9.1.4 Common Workflows

Step 1. New Model/Task Integration

Step 1-1. Discover existing examples

```
./scripts/dx_tool.sh list
./scripts/dx_tool.sh search yolov8
./scripts/dx_tool.sh info yolov9-s_640x640
```

Use this first to avoid creating duplicate or inconsistent example names.

Step 1-2. Add a new model example

```
./scripts/dx_tool.sh add
```

The add flow is intended for contributors creating a new example under the `<task>/<family>/<variant>/` layout. Typical information includes:

- target language (C++, Python, or both)
- task type
- postprocessor (reference)
- variant name (the `.dxnn` stem)
- family directory (empty = the reference's family)
- generation mode (sync + async, or sync only)

For YOLO-family model onboarding and postprocessor selection details, refer to DX-APP YOLO Customizing Guide.

You can also invoke `add_model.sh` directly for non-interactive usage:

```
# Create from a postprocessor family (the reference is the alphabetically first registered variant that uses it, preferring the same task)
./scripts/add_model.sh yolo-custom_640x640 object_detection --postprocessor yolov8

# Copy from an existing variant (useful for same-family variants)
./scripts/add_model.sh yolov7-custom_1280x1280 object_detection --base-model yolov7_640x640 --input-size 1280x1280

# Open a new family for the variant
./scripts/add_model.sh my-det_640x640 object_detection --family my_det --postprocessor yolov5

# Generate and verify in one step
```


© Copyright 2026 DEEPX All Rights Reserved.

```
./scripts/add_model.sh yolo-custom_640x640 object_detection --postprocessor yolov8 --verify --model assets/models/yolo-custom_640x640.dxnn
```

The first argument is the variant: the `.dxnn` stem, which is also the directory name. The example lands in `src/{cpp,python}_example/<task>/<family>/<variant>/` with the C++ factory scoped in `dxapp::v_<variant>`, `config.json` names the variant, and `config/model_registry.json` and `config/test_models.conf` gain the variant unless `--no-register` is given (a variant that is already registered takes its family, model file and input size from its row). The generator is `scripts/add_model_variant.py`; `add_model.sh` adds `--verify` (with `--model`, `--image`, `--video`, `--no-video`) and `--auto-add` around it.

Common `--postprocessor` values include:

Task	Common values
Object detection	`yolov5`, `yolov7`, `yolov8`, `yolov9`, `yolov10`, `yolov11`, `yolov12`, `yolov13`, `yolov26`, `yolox`, `damoyolo`, `nanodet`, `ssd`
Semantic segmentation	`deeplabv3`, `bisenetv1`, `bisenetv2`, `segformer`, `fast_segmentation`
Instance segmentation	`yolov5seg`, `yolov8seg`, `yolov26seg`
Pose estimation	`yolov5pose`, `yolov8pose`, `yolov26pose`
Face detection	`scrfd`, `yolov5face`, `yolov7face`

Use `fast_segmentation` for generic semantic-segmentation models that output low-resolution logits or class maps and should use the `FastSegmentationPostprocessor` fast path.

Key `add_model.sh` options

Option	Description
`--postprocessor <type>`	Select the post-processing family to use as template
`--base-model <variant>`	Copy from a specific existing variant instead of the default reference
`--family <name>`	Family directory (default: the registry row's family, else the reference's)

© Copyright 2026 DEEPX All Rights Reserved.

Option	Description
`--input-size <WxH>`	Model input size (default: the registry row's, else the reference's)
`--dxnn <file>`	Model file name (default: `<variant>.dxnn`)
`--no-register`	Do not add the variant to `model_registry.json` / `test_models.conf`
`--lang <lang>`	Target language: `cpp`, `py` or `both` (default: `both`)
`--sync-only`	Generate only the sync entries
`--verify`	Build and run inference verification after generation
`--model <path>`	`.dxnn` model file for `--verify`
`--no-video`	Skip video verification (image only)
`--auto-add`	Generate the example of every registry row that has none

After generation, review the resulting files under `src/cpp_example/` and/or `src/python_example/`.

Step 1-3. Verify the integration

After generation, verify that the new example is correctly integrated and runs without issues.

```
# Build and run every supported registry model (long; interactive)
./scripts/dx_tool.sh validate

# Test the new example (e.g., for C++)
./scripts/dx_tool.sh run --lang cpp --filter <variant>
```

NOTE

If you used the `--verify` flag with `add_model.sh` in the previous step, this manual verification might be redundant but is still recommended for visual confirmation.

© Copyright 2026 DEEPX All Rights Reserved.

Step 2. Packaging & Distribution

Step 2-1. Extract a standalone package

```
./scripts/dx_tool.sh extract
```

Use this when you need a standalone copy of one example (its model directory plus the `common/` files it depends on) for deployment outside the repository.

The extracted package carries only the `common/` files the model actually depends on: only a fraction of the shared files, instead of every postprocessor, runner, and visualizer for all 29 tasks.

To get the complete shared framework instead, answer `n` at the `Prune unused common/ files? [Y/n]` prompt, or pass `--no-prune`:

```
./scripts/extract_model_package.sh object_detection/yolov7 --output-dir out_yolov7 --no-prune
```

Pruned and full packages are equivalent: they compile identically and produce byte-identical inference output. Note that the Python visualizers use an unseeded color palette, so rendered colors differ between runs either way.

Step 2-2. Extract a Visual Studio solution package on Windows

Use `extract_sln_package.bat` when you want to extract a single C++ example into a Visual Studio/CMake package that can be opened or built outside the full DX-APP solution.

```
.\scripts\extract_sln_package.bat image_classification/resnet/resnet50_224x224 --output-dir out_res
net50
```

The output is created under:

```
out_resnet50\sln\image_classification\resnet50_224x224\
```

If CMake and the Visual Studio 2022 generator are available, the extractor also configures the package immediately and generates a solution file:

```
out_resnet50\sln\image_classification\resnet50_224x224\build\dxapp_resnet50_224x224_sln_packa
ge.sln
```

The package includes the selected model sources, shared C++ example helpers, `CMakeLists.txt`, `build.bat`, and generated dependency defaults. To build it directly:

© Copyright 2026 DEEPX All Rights Reserved.

```
cd .\out_resnet50\sln\image_classification\resnet50_224x224
.\build.bat
```

OpenCV and DXRT paths are configured through CMake. The extractor writes the dependency defaults it can detect at extraction time to:

```
cmake\dxapp_package_deps.cmake
cmake\dxapp_package_deps.bat
```

In the usual local developer environment, users should not need to edit Visual Studio property pages manually. If the package is moved to another PC or dependency paths change, set one of these variables before running `build.bat`, or edit `cmake\dxapp_package_deps.cmake`:

Variable	Purpose
`DEEPX_SDK_DIR`	DEEPX SDK (DXRT) root
`DXRT_INSTALLED_DIR`	DXRT install root used for `include`, `lib`, and `bin`
`OpenCV_DIR`	OpenCV CMake package directory
`VCPKG_INSTALLED_DIR`	vcpkg installed tree used for runtime DLL lookup

To create only the package skeleton without configuring CMake or generating `.sln` files:

```
.\scripts\extract_sln_package.bat image_classification/resnet/resnet50_224x224 --output-dir out_res
net50 --no-generate-sln
```

Step 3. Repository Maintenance

Step 3-1. Validate repository consistency

```
./scripts/dx_tool.sh validate
```

Run validation after adding or restructuring examples. It builds and runs every supported registry model, so it catches examples that no longer build or run, and it generates any registered variant that has no example yet.

Step 3-2. Run examples selectively

© Copyright 2026 DEEPX All Rights Reserved.

```
# Interactive — guided category/model selection with performance output
./scripts/dx_tool.sh run
scripts/run_examples.sh

# Non-interactive
./scripts/dx_tool.sh run --lang cpp
./scripts/dx_tool.sh run --lang py
./scripts/dx_tool.sh run --lang both
```

Use `run` to execute filtered example sets without manually locating every command.
In interactive mode, selecting a category shows all available models in that category.

Step 3-3. Benchmark examples

```
./scripts/dx_tool.sh bench --lang cpp
./scripts/dx_tool.sh bench --lang py
```

Use `bench` when you want comparable runtime results across example variants.

Notes for Automation

For CI or scripted usage:

- prefer direct subcommands over interactive mode
- keep `dx_tool.sh` for contributor automation, not end-user setup
- use `setup.sh`, `build.sh`, and `run_tc.sh` for deterministic pipeline steps

9.1.5 Supplementary Information

Script Relationships

`dx_tool.sh` is a **developer workflow tool**, not a replacement for every top-level script.

- `setup.sh`: prepares shared assets such as models and videos
- `build.sh`: builds C++ binaries and Python bindings
- `run_tc.sh`: runs the repository test suites
- `scripts/dx_tool.sh`: helps contributors add, inspect, validate, run, and benchmark example content

ⓒ Copyright 2026 DEEPX All Rights Reserved.

A typical contributor workflow is:

- (1) prepare assets with `./setup.sh`
- (2) build binaries with `./build.sh`
- (3) inspect or add examples with `./scripts/dx_tool.sh`
- (4) run validations and tests with `./scripts/dx_tool.sh validate` and `./run_tc.sh`

FAQ & Troubleshooting

The exact workflow depends on the contributor task, but the following command patterns are the most commonly used.

Inspect the current repository state

```
./scripts/dx_tool.sh list
./scripts/dx_tool.sh search yolov8
./scripts/dx_tool.sh info yolov9-s_640x640
```

Validate after adding or refactoring examples

```
./scripts/dx_tool.sh validate
```

Run language-specific example subsets

```
# Interactive — 6-stage guided menu
./scripts/dx_tool.sh run
scripts/run_examples.sh

# Non-interactive — pass options directly
./scripts/dx_tool.sh run --lang cpp
./scripts/dx_tool.sh run --lang py
./scripts/dx_tool.sh run --lang both
```

Benchmark language-specific example subsets

```
./scripts/dx_tool.sh bench --lang cpp
./scripts/dx_tool.sh bench --lang py
```

See Also

- `scripts/dx_tool.sh`
- `scripts/add_model.sh`

© Copyright 2026 DEEPX All Rights Reserved.

- `scripts/validate_models.sh`
- `scripts/run_examples.sh`
- `scripts/bench_models.sh`
- `config/model_registry.json`
- DX-APP Example Source Structure

ⓒ Copyright 2026 DEEPX All Rights Reserved.

9.2 DX-APP Example Source Structure

This guide is intended for contributors and maintainers who need to understand or extend the refactored DX-APP example layout.

9.2.1 Overview

The `src/` directory is the implementation root for DX-APP examples and shared runtime logic.

Its structure is designed to:

- separate C++ and Python example implementations clearly
- organize examples by **AI task** and then by **model family**
- share post-processing logic across languages
- make onboarding of new models predictable
- support testing, validation, and benchmarking workflows

9.2.2 Directory Hierarchy & Organization

Top-Level Layout Strategy

```
src/
├── cpp_example/          # C++ examples organized by task/model
│   └── common/           # ← C++ shared runtime layer
├── python_example/       # Python examples organized by task/model
│   └── common/           # ← Python shared runtime layer
├── postprocess/          # C++ post-processing (consumed by pybind11 bindings)
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
├── bindings/          # pybind11 bridge exposing src/postprocess/ to Python
└── utility/          # Shared utility code used by build flow
```

- **src/cpp_example/** : end-to-end C++ example applications — each model directory contains thin entry-point files that delegate to the shared `common/` layer
- **src/cpp_example/common/** : shared C++ runtime layer — base interfaces (4 hpp), processors, runners, visualizers, input sources (5 hpp), config, and utilities
- **src/python_example/** : end-to-end Python example applications — same factory-based delegation pattern as C++
- **src/python_example/common/** : shared Python runtime layer — base interfaces (4 py), processors, runners, visualizers, input sources (5 py), config, and utilities
- **src/postprocess/** : C++ post-processing libraries consumed by the **pybind11 bindings** (`src/bindings/`). This enables `*_cpp_postprocess` variants to use C++ decode logic from Python
- **src/bindings/** : pybind11 bridge exposing `src/postprocess/` to Python as the `dx_postprocess` package
- **src/utility/** : common support code shared across the build flow

Both `cpp_example/common/` and `python_example/common/` share the same core modules: `base/` , `config/` , `processors/` , `runner/` , `inputs/` , `visualizers/` , `utility/` . They are language-specific implementations of the same architectural pattern.

Task-First Organization

Both `cpp_example/` and `python_example/` follow the same task-first structure.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Representative task directories include:

- `image_classification/`
- `object_detection/` — PPU models are a family here, for example `yolo_ppu/`
- `face_detection/`
- `pose_estimation/`
- `semantic_segmentation/`
- `instance_segmentation/`
- `depth_estimation/`
- `face_recognition/`
- `image_denoising/`
- `low_light_enhancement/`
- `super_resolution/`
- `oriented_object_detection/`
- `hand_landmark/`
- `person_attribute/`
- `person_reid/`
- `face_landmark/`

This makes it easy to answer both questions below:

- what task does this example belong to?
- where should a new model example be added?

Family and Variant Subdirectories

Within each task directory, examples are split by model family, then by variant. The variant directory name is the `.dxnn` stem.

Examples:

```
src/cpp_example/object_detection/yolov9/yolov9-s_640x640/
src/cpp_example/object_detection/yolov8/yolov8-n_640x640/
src/python_example/object_detection/yolov9/yolov9-s_640x640/
src/python_example/object_detection/ssd/ssd-mobilenetv2-lite_300x300/
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

This gives each variant its own workspace for:

- config files
- source files
- factory helpers
- model-specific wiring

9.2.3 Language-Specific Structure & Variants

Common Runtime Layer (common/)

Shared Runtime Layer (common/)

Both src/cpp_example/common/ and src/python_example/common/ implement the same shared runtime architecture. Each provides base interfaces, configuration loading, input sources, processors, runners, visualizers, and utilities — so individual model directories only need thin entry points and a factory.

C++ Common Structure

```
src/cpp_example/common/
├── base/                       # Abstract interfaces (.hpp)
│   ├── i_factory.hpp            #   IFactory pattern: task-specific I<Task>Factory interfaces (e.g.,
IDetectionFactory) that create the preprocessor, postprocessor and visualizer
│   ├── i_processor.hpp          #   IPreprocessor / IPostprocessor — image → input tensor, model
outputs → results
│   ├── i_visualizer.hpp         #   IVisualizer — draw(frame, results, ctx) → frame
│   └── i_input_source.hpp       #   IInputSource — image/video/camera/RTSP abstraction
├── config/
│   └── model_config.hpp         # ModelConfig — loads config.json
├── processors/                 # Shared processors
│   ├── yolo_detection_postprocessor.hpp     # YOLOv5/v7/v8 and the later YOLO detectors
│   ├── scrfd_face_postprocessor.hpp
│   ├── nanodet_postprocessor.hpp
│   ├── damoyolo_postprocessor.hpp
│   ├── ssd_postprocessor.hpp
│   ├── segmentation_postprocessor.hpp       # DeepLabV3 plus FastSegmentationPostprocessor
│   ├── anchorless_instance_seg_postprocessor.hpp
│   ├── depth_postprocessor.hpp
│   └── ...
├── runner/                     # Task-specific runner pairs
│   ├── sync_detection_runner.hpp  #   Sequential detection loop
│   ├── async_detection_runner.hpp #   Pipelined detection loop
│   ├── sync_classification_runner.hpp
│   ├── async_classification_runner.hpp
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
│   ├── sync_segmentation_runner.hpp
│   ├── async_segmentation_runner.hpp
│   └── ...                    #   one sync + one async runner per task type
├── inputs/                     # 5 input source headers
│   ├── camera_source.hpp
│   ├── image_source.hpp
│   ├── video_source.hpp
│   ├── rtsp_source.hpp
│   └── input_factory.hpp
├── visualizers/                # Task-specific visualizers
│   ├── detection_visualizer.hpp
│   ├── classification_visualizer.hpp
│   ├── segmentation_visualizer.hpp
│   ├── face_visualizer.hpp
│   ├── pose_visualizer.hpp
│   └── ...
├── utility/                    # Utility headers
│   ├── common_util.hpp
│   ├── labels.hpp
│   ├── preprocessing.hpp
│   ├── profiling.hpp
│   ├── run_dir.hpp
│   ├── safe_queue.hpp
│   ├── verify_serialize.hpp
│   ├── visualization.hpp
│   └── ...                    #   colorspace, frame_reorder, kitti_calib,
│                              #   lidar_util, sfa3d_geometry, sr_tiling
├── graph/                      # Multi-model graph engine
├── registry/                   # Concrete model registry used by the graph engine
├── trackers/                   # IoU tracker (stable instance colors across frames)
├── test/                       # Unit test for common/
└── third_party/                # Third-party header-only libraries
    └── nlohmann_json.hpp       #   JSON for Modern C++
```

C++ runners are **task-specific**: each task type has a dedicated sync/async runner pair (e.g., `sync_detection_runner`, `async_detection_runner`). This provides optimal performance for each task's specific data flow.

Python Common Structure

```
src/python_example/common/
├── base/                # Abstract interfaces (.py)
│   ├── i_factory.py     #   task-specific I<Task>Factory interfaces (e.g., IDetectionFactory) —
create the preprocessor, postprocessor and visualizer
│   ├── i_processor.py   #   IPreprocessor / IPostprocessor — image → input tensor, model outputs
→ results
│   ├── i_visualizer.py  #   IVisualizer — visualize(frame, results) → frame
│   └── i_input_source.py #   IInputSource — image/video/camera/RTSP abstraction
├── config/
│   ├── config_schema.py   # Config schema validation for config.json values
```


© Copyright 2026 DEEPX All Rights Reserved.

```
│   └── model_config.py    # ModelConfig — loads config.json (input size, labels, thresholds)
├── processors/            # Shared processors
│   ├── yolo_postprocessor.py           # YOLOv5/v7/v8/v9/v10/v11/v12/YOLOX
│   ├── fast_yolo_postprocessor.py      # opt-in exact fast path (YOLOv5/v7, byte-identical)
│   ├── face_postprocessor.py           # SCRFD, YOLOv5Face, YOLOv7Face
│   ├── segmentation_postprocessor.py   # BiSeNet, DeepLabV3+, SegFormer
│   ├── fast_segmentation_postprocessor.py # opt-in fast path (SegFormer / low-res seg)
│   ├── instance_seg_postprocessor.py   # YOLOv5Seg, YOLOv8Seg, YOLOv26Seg
│   ├── fast_instance_seg_postprocessor.py # opt-in fast mask path (YOLOv8-seg)
│   ├── yolact_postprocessor.py         # YOLACT instance segmentation
│   ├── fast_yolact_postprocessor.py    # opt-in fast mask path (YOLACT)
│   ├── obb_postprocessor.py            # YOLOv26OBB
│   ├── pose_postprocessor.py           # YOLOv5-Pose, YOLOv8-Pose
│   ├── ppu_postprocessor.py            # PPU variants (YOLOv3/v4/v5/v7/v8/v9/v10/v11/v12, YOLOX)
│   ├── classification_postprocessor.py # EfficientNet, AlexNet, etc.
│   ├── efficientdet_postprocessor.py   # EfficientDet
│   ├── fast_efficientdet_postprocessor.py # opt-in exact fast path (EfficientDet, byte-identical)
│   ├── depth_postprocessor.py          # FastDepth, SCDepth
│   ├── nanodet_postprocessor.py        # NanoDet
│   ├── ssd_postprocessor.py            # SSD MobileNet
│   ├── damoyolo_postprocessor.py       # DAMOYOLO
│   ├── embedding_postprocessor.py      # GenericEmbeddingPostprocessor (base of CLIP / ArcFace)
│   ├── restoration_postprocessor.py    # DnCNN, Real-ESRGAN
│   ├── nms_utils.py                    # Shared NMS / box utilities
│   ├── letterbox_preprocessor.py       # Shared letterbox preprocessing
│   └── ...
├── runner/                # Generic runner files
│   ├── sync_runner.py     # SyncRunner — sequential Pre→Infer→Post→Display loop
│   ├── async_runner.py    # AsyncRunner — pipelined multi-thread runner
│   ├── args.py            # Unified CLI argument parser (--model, --image, --video, etc.)
│   ├── entry.py           # run_entry() used by every entry script
│   ├── interrupts.py      # Graceful stop on SIGINT/SIGTERM
│   ├── run_dir.py         # `--save` run-directory and run_info helper
│   ├── sr_tiling.py       # Tiled super-resolution helper
│   └── verify_serialize.py # Serialize results to JSON for numerical verification
├── inputs/                # 5 input source files
│   ├── image_source.py
│   ├── video_source.py
│   ├── camera_source.py
│   ├── rtsp_source.py
│   └── input_factory.py
├── visualizers/           # Task-specific visualizers
│   ├── detection_visualizer.py
│   ├── classification_visualizer.py
│   ├── segmentation_visualizer.py
│   ├── face_visualizer.py
│   ├── pose_visualizer.py
│   ├── instance_seg_visualizer.py
│   ├── obb_visualizer.py
│   └── ...
├── trackers/              # IoU tracker (stable instance colors across frames)
├── variant_config.py      # Builds processors from the variant config.json
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
├── variants.py            # Variant lookup in config/model_registry.json
└── utility/                   # Utility files
    ├── common_util.py     # General utilities
    ├── labels.py          # COCO / ImageNet label constants
    ├── preprocessing.py   # Shared resize/normalize/letterbox
    ├── profiling.py       # Stage-wise latency profiler
    ├── safe_queue.py      # Thread-safe queue for async pipeline
    ├── skeleton.py        # Pose skeleton definitions
    ├── visualization.py   # Drawing helpers (boxes, text, masks)
    └── ...                # colorspace, kitti_calib, lidar_input, sfa3d_geometry, video_io
```

Python runners are **generic**: `SyncRunner` and `AsyncRunner` work for all task types via the factory pattern. This provides simplicity and uniform usage across all models.

Key Difference: C++ vs Python Runners

Aspect	**C++ (`cpp_example/common/runner/`)**	**Python (`python_example/common/runner/`)**
Runner set	One sync + one async runner per task type	`SyncRunner` + `AsyncRunner` plus shared helpers
Dispatch	Task-specific runner per category	Generic runner for all tasks
Example	`sync_detection_runner.hpp`	`sync_runner.py`

How Model Directories Connect to `common/`

C++ variant directory

The factory and the sync/async entries live in `<task>/<family>/<variant>/`. The directory name is the `.dxnn` stem.

```
src/cpp_example/object_detection/yolov8/yolov8-n_640x640/
├── factory/
│   └── yolov8-n_640x640_factory.hpp
├── yolov8-n_640x640_sync.cpp        # Entry → sync_detection_runner
├── yolov8-n_640x640_async.cpp       # Entry → async_detection_runner
└── config.json
```

Python variant directory

ⓒ Copyright 2026 DEEPX All Rights Reserved.

The same folder holds the factory, `config.json`, and one thin script per execution mode. Each script fixes that variant.

```
src/python_example/object_detection/yolov8/yolov8-n_640x640/
├── factory/
│   └── yolov8-n_640x640_factory.py
├── config.json
├── yolov8-n_640x640_sync.py
├── yolov8-n_640x640_async.py
├── yolov8-n_640x640_sync_cpp_postprocess.py
└── yolov8-n_640x640_async_cpp_postprocess.py
```

The factory imports shared components from `common/`:

```cpp
// C++ factory example
#include "common/processors/yolo_detection_postprocessor.hpp"
#include "common/visualizers/detection_visualizer.hpp"
```

```python
# Python factory example (config-driven: processors come from the variant's config.json)
from common.base import IDetectionFactory
from common.variant_config import build_processor, default_variant, load_variant_config
```

The entry-point script/program delegates to the runner:

```python
# Python
from common.runner import SyncRunner
from common.runner.entry import run_entry

run_entry(_script, factory_cls=Yolov9Factory, runner_cls=SyncRunner,
          description="yolov9-s_640x640 sync inference",
          fixed_variant="yolov9-s_640x640")
```

Adding a variant means a new `<task>/<family>/<variant>/` directory with `config.json`, a factory, and that variant's entry files.

Entry Naming

C++

```
src/cpp_example/<task>/<family>/<variant>/
├── factory/<variant>_factory.hpp
├── <variant>_sync.cpp
```


© Copyright 2026 DEEPX All Rights Reserved.

```
  ├── <variant>_async.cpp
  └── config.json
```

- `<variant>_sync.cpp` : sequential execution for that variant
- `<variant>_async.cpp` : pipelined execution for that variant
- `bin/<variant>_sync` and `bin/<variant>_async` : built executables

Python

```
src/python_example/<task>/<family>/<variant>/
├── factory/<variant>_factory.py
├── config.json
├── <variant>_sync.py
├── <variant>_async.py
├── <variant>_sync_cpp_postprocess.py
└── <variant>_async_cpp_postprocess.py
```

- `*_sync.py` : synchronous path
- `*_async.py` : asynchronous path
- `*_sync_cpp_postprocess.py` : synchronous path using shared C++ post-processing bindings
- `*_async_cpp_postprocess.py` : asynchronous path using shared C++ post-processing bindings

9.2.4 Cross-Language Integration & Frameworks

Shared Post-Processing Layer

C++ and Python Shared Runtime (`common/`)

Both `cpp_example/` and `python_example/` contain a `common/` directory with the same core modules:

Module	Contents	Role
`common/base/`	4 interface headers (.hpp)	`IFactory` , `IPreprocessor` / `IPostprocessor` , `IVisualizer` , `IInputSource`
`common/config/`	`model_config.hpp`	Loads `config.json` (input size, labels, thresholds)
`common/processors/`	Postprocessor / preprocessor headers	Shared decode logic for all model families

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Module	Contents	Role
`common/runner/`	Runner headers	A dedicated sync/async runner pair per task type
`common/inputs/`	5 source headers	Image, Video, Camera, RTSP input abstraction
`common/visualizers/`	Visualizer headers	Task-specific result rendering
`common/utility/`	Utility headers	Labels, preprocessing, profiling, run_dir, colorspace, verify_serialize

This is the core architectural pattern of DX-APP: model directories are thin wrappers (factory + entry points) that delegate all heavy logic to their respective `common/` layer.

Pybind11 Post-Processing Bridge (`src/postprocess/` + `src/bindings/`)

`src/postprocess/` contains C++ post-processing implementations that are **not** linked by the C++ examples, except for four modules (`sfa3d` , `superpoint` , `dope` , `yolopv2`) whose sources are compiled into them. They are consumed by the pybind11 bindings under `src/bindings/python/dx_postprocess/` .

This bridge enables the `*_cpp_postprocess.py` Python variants to use C++ decode logic for higher performance, while the pure Python variants (`*_sync.py` , `*_async.py`) use `python_example/common/processors/` natively.

Variant	Post-processing source
`*_sync.cpp` / `*_async.cpp`	`cpp_example/common/processors/`
`*_sync.py` / `*_async.py`	`python_example/common/processors/`
`*_sync_cpp_postprocess.py` / `*_async_cpp_postprocess.py`	`src/postprocess/` via pybind11

Numerical Verification (`DXAPP_VERIFY`)

ⓒ Copyright 2026 DEEPX All Rights Reserved.

DX-APP can serialize post-processing results to JSON for inspection and debugging:

- **`DXAPP_VERIFY=1`** : serializes post-processing results to `logs/verify/{model}.json` (last frame) and `{model}.frames.jsonl` (every frame, in input order)
- **`common/runner/verify_serialize.py`** : serializes postprocess results to JSON for comparison

This helps catch regressions such as broken post-processing, incorrect model configurations, or NPU output changes.

9.2.5 Management & Extension

Model Registry & Automation

`config/model_registry.json`

The model registry is a JSON array that serves as the single source of truth for model metadata:

```
{
  "model_name": "yolov9s",
  "dxnn_file": "yolov9-s_640x640.dxnn",
  "variant": "yolov9-s_640x640",
  "family": "yolov9",
  "task": "object_detection",
  "add_model_task": "object_detection",
  "postprocessor": "yolov8",
  "input_width": 640,
  "input_height": 640,
  "config": { "score_threshold": 0.25, "nms_threshold": 0.45 },
  "supported": true
}
```

`model_name` is the registry key used by `./setup.sh --models` . The example directory is `<task>/<family>/<variant>` , here `object_detection/yolov9/yolov9-s_640x640` .

`scripts/add_model.sh`

© Copyright 2026 DEEPX All Rights Reserved.

The `add_model.sh` script copies a reference example picked from the registry and rewrites, for the new variant:

- `config.json` (variant, model file, input size, registry thresholds; processor classes from the reference)
- Python `factory/<variant>_factory.py` and `factory/__init__.py` (plus `custom_ops.py` when the reference has one)
- the entry-point scripts the reference has (sync/async × python/cpp_postprocess)
- C++ `factory/<variant>_factory.hpp` (namespace `dxapp::v_<variant>`), `<variant>_sync.cpp` and `<variant>_async.cpp` under `src/cpp_example/<task>/<family>/<variant>/`
- a `config/model_registry.json` row and a `config/test_models.conf` line (unless `--no-register`)

This enables onboarding a new model with zero manual code in most cases; see the DX Tool Guide for the options.

Contributor Rules

Adding New Examples

When adding a new example, keep the following rules:

- (1) choose the correct task directory first
- (2) create `<task>/<family>/<variant>/` for that model
- (3) follow existing variant naming conventions
- (4) place shared decode logic in `src/cpp_example/common/processors/` (C++) or `src/python_example/common/processors/` (Python) when appropriate
- (5) add pybind11 bindings in `src/postprocess/` + `src/bindings/` only if the new flow needs `*_cpp_postprocess.py` variants
- (6) register the model in `config/model_registry.json` if it should be part of the standard validation flow
- (7) update validation/test registration where required

Contributor checklist

© Copyright 2026 DEEPX All Rights Reserved.

When a new example is intended to become part of the maintained repository flow, confirm all of the following:

- source files are placed under the correct `<task>/<family>/<variant>/` directory
- naming follows the current variant convention
- required shared post-processing logic exists or is added under `src/cpp_example/common/processors/` (C++) or `src/python_example/common/processors/` (Python)
- model assets can be prepared through the standard setup flow
- validation passes
- test registration is updated if automated coverage is required
- user-facing or contributor-facing docs are updated when the structure meaningfully changes

---

9.2.6 Supplementary Information

Connection to Testing

The source tree and the test tree are related, but not identical.

Shared Test Infrastructure (`tests/test_helpers/`)

`tests/test_helpers/` provides shared constants and utilities used by both C++ and Python test suites:

- `constants.py` — paths, sample-image maps, timeout values
- `utils.py` — executable/script and model discovery, environment setup (full module list: see DX-APP C++ Example Tests)

C++ tests

C++ tests are organized as follows. For the full list, see DX-APP C++ Example Tests.

Category	File	Description
CLI Help	`test_cli_help.py`	Validates `--help` output for all executables
CLI Basic	`test_cli_basic.py`	Validates invalid-argument and no-argument handling
E2E	`test_e2e.py`	End-to-end inference with image/video inputs

© Copyright 2026 DEEPX All Rights Reserved.

Category	File	Description
Visualization	`test_visualization.py`	Validates output image generation with `--save`
Feature: Save Mode	`test_save_mode.py`	Tests `--save` / `--save-dir` behavior
Feature: Dump Tensors	`test_dump_tensors.py`	Tests `--dump-tensors` output
Feature: Verify	`test_verify.py`	Tests `DXAPP_VERIFY` environment variable
Feature: Multi-Loop	`test_multi_loop.py`	Tests `--loop` repeated execution
Feature: Signal	`test_signal_handling.py`	Tests SIGINT graceful shutdown

The tests discover the executables in `bin/`.

Python tests

- Python example tests discover variant scripts through `tests/test_helpers/utils.py` and `config/model_registry.json`
- `tests/python_example/test_visualization.py` validates output image generation across task types
- `tests/python_example/unit/` holds processor and layout checks
- adding a new source directory alone does **not** automatically guarantee full test coverage

Relevant files include:

- `tests/test_helpers/utils.py`
- `tests/test_helpers/constants.py`
- `tests/python_example/test_visualization.py`
- `tests/python_example/unit/`
- `config/test_models.conf`

Model validation

- `scripts/validate_models.sh` runs registry-driven validation across all supported models
- `config/model_registry.json` is the primary reference for which models are validated

© Copyright 2026 DEEPX All Rights Reserved.

This means source layout, test coverage, and registry entries must be updated together when onboarding new examples.

Developer Resources

Recommended Onboarding

When you add or refactor a model example:

- (1) create or update the source layout under `src/`
- (2) prepare the required model assets
- (3) build the repository
- (4) validate the example structure
- (5) run relevant tests

Typical related commands:

```
./setup.sh
./build.sh --clean --all
./scripts/dx_tool.sh validate
./scripts/validate_models.sh --lang py
./run_tc.sh --cpp --cli
./run_tc.sh --python
```

If the example is intended to be part of the standard repository workflow, also review the relevant documentation pages so that user and contributor guidance stays aligned with the current structure.

See Also

- `src/cpp_example/`
- `src/cpp_example/common/`
- `src/python_example/`
- `src/python_example/common/`
- `src/postprocess/` (pybind11 source)
- `src/bindings/python/dx_postprocess/`
- `config/model_registry.json`
- `scripts/validate_models.sh`
- `tests/cpp_example/`

© Copyright 2026 DEEPX All Rights Reserved.

- `tests/python_example/`
- DX Tool Guide

ⓒ Copyright 2026 DEEPX All Rights Reserved.

9.3 DX-APP YOLO Customizing Guide

This guide explains how to add or adapt YOLO-family models in DX-APP. It is intended for contributors who need to onboard a new YOLO model, select the correct postprocessor, tune thresholds, or adjust factory code after model conversion.

DX-APP supports YOLO examples through a common factory pattern. Follow these workflow steps:

- **Step 1**: Choose the Right YOLO Family
- **Step 2**: Generate a New YOLO Example
- **Step 3**: Tune `config.json`
- **Step 4**: Review the Factory Code
- **Step 5**: Understand Output Shape Compatibility
- **Step 6**: Build and Run
- **Step 7**: Verify Before Registry Integration
- **Step 8**: Update Registry and Test Metadata

9.3.1 [Step 1] Choose the Right YOLO Family

Start by identifying the output format of the compiled `.dxnn` model. The model name alone is helpful, but the output tensor layout is the final source of truth.

Model type	Typical output	add_model.sh --postprocessor	C++ postprocessor	Python postprocessor
YOLOv3/v4/v5-style anchor-based detection	box + objectness + class scores	`yolov5`	`YOLOv5Postprocessor` (the registered `yolov4-*` C++ examples use `EfficientDetPostprocessor`)	`YOLOv5Postprocessor`
YOLOv7 anchor-based detection	YOLOv7 anchors/strides	`yolov7`	`YOLOv7Postprocessor`	`YOLOv5Postprocessor` with model config

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Model type	Typical output	add_model.sh --postprocessor	C++ postprocessor	Python postprocessor
YOLOX	anchor-free YOLOX-style detection	`yolox`	`YOLOXPostprocessor`	`YOLOXPostprocessor`
YOLOv8/v9/v10/v11/v12/v13/v26 detection	anchor-free / DFL-style detection, no objectness field	`yolov8`, `yolov9`, `yolov10`, `yolov11`, `yolov12`, `yolov13`, `yolov26`	`YOLOv8Postprocessor` (v8, v9, v11, v12, v13; `yolo11-n_640x640` uses `YOLOv11Postprocessor`), `YOLOv10Postprocessor` (v10), `YOLOv26Postprocessor` (v26)	`YOLOv8Postprocessor` family
YOLO segmentation	boxes + masks/ prototypes	`yolov8seg`, `yolov26seg`, `yolov5seg`	task-specific segmentation postprocessor	task-specific segmentation postprocessor
YOLO pose	boxes + keypoints	`yolov5pose`, `yolov8pose`, `yolov26pose`	task-specific pose postprocessor	task-specific pose postprocessor
YOLO face detection	face boxes and optional landmarks	`yolov5face`, `yolov7face`	face-specific postprocessor	face-specific postprocessor
YOLO OBB	oriented boxes	`yolov26obb`	OBB postprocessor	OBB postprocessor
PPU YOLO variants	hardware-assisted postprocess output	`yolov5_ppu`, `yolov7_ppu`, `yolov8_ppu`, etc.	PPU postprocessor	PPU postprocessor

ⓒ Copyright 2026 DEEPX All Rights Reserved.

> **NOTE**
>
> If the model is a new detection-only YOLO variant and its output is compatible with the YOLOv8+ anchor-free/DFL family, start with `--postprocessor yolov8`. If it has explicit objectness and anchor-grid decoding, start with `--postprocessor yolov5` or `--postprocessor yolov7`.
>
> The YOLOv12 and YOLOv13 examples do not use a postprocessor of their own (the C++ `YOLOv12Postprocessor` class exists but is unused): their outputs follow the YOLOv8 layout, so `--postprocessor yolov12` and `--postprocessor yolov13` select `YOLOv8Postprocessor`, as the registered `yolov12-*` and `yolov13-*` examples do.

9.3.2 [Step 2] Generate a New YOLO Example

Use `scripts/add_model.sh` to create the C++ and Python example trees from the closest reference model.

```
# YOLOv8-style detection model
./scripts/add_model.sh yolo-custom_640x640 object_detection --postprocessor yolov8 --lang both

# YOLOv7-style anchor-based model
./scripts/add_model.sh yolov7-custom_1280x1280 object_detection --base-model yolov7_640x640 --input-size 1280x1280 --lang both

# YOLOX-style model
./scripts/add_model.sh custom-yolox_640x640 object_detection --postprocessor yolox --lang both
```

An example lives in the family/variant layout, which is where `add_model.sh` writes it. `<family>` groups related models. `<variant>` is the `.dxnn` stem, and that folder holds the factory, `config.json`, and the entry files.

```
src/cpp_example/object_detection/<family>/<variant>/
├── factory/<variant>_factory.hpp
├── <variant>_sync.cpp
├── <variant>_async.cpp
└── config.json

src/python_example/object_detection/<family>/<variant>/
├── factory/<variant>_factory.py
├── config.json
├── <variant>_sync.py
├── <variant>_async.py
```


© Copyright 2026 DEEPX All Rights Reserved.

```
├── <variant>_sync_cpp_postprocess.py
└── <variant>_async_cpp_postprocess.py
```

After generation, review both language trees. Do not assume a generated factory is final; the selected template only gives a safe starting point.

`add_model.sh` adds the variant to `config/model_registry.json` and `config/test_models.conf`; pass `--no-register` for local experiments.

9.3.3 [Step 3] Tune `config.json`

Most YOLO customization is done through the `config` object of each variant's `config.json`. The Python scripts load the `config.json` next to the script automatically. The C++ executables read it only when it is passed with `--config` (for example `./bin/<variant>_sync -m assets/models/<variant>.dxnn -i sample/img/sample_kitchen.jpg --config src/cpp_example/object_detection/<family>/<variant>/config.json`); without `--config` they use the defaults built into the factory. Put the keys inside `config`: C++ lets the members of `config` override top-level keys, while Python lets top-level keys override `config`. Keep C++ and Python configs aligned unless a language-specific path intentionally differs.

Common keys:

Key	Meaning	Typical use
`obj_threshold`	objectness threshold	YOLOv5/v7/YOLOX-style models with objectness
`score_threshold`	class confidence threshold (Python reads it as `conf_threshold`; `conf_threshold` itself is ignored by C++)	all detection models
`nms_threshold`	IoU threshold for NMS	all detection models
`num_classes`	number of classes	custom datasets that are not COCO-80
`class_names`	display labels (C++ only; the Python `DetectionVisualizer` labels come from `custom_labels` in the variant's `visualizer.kwargs`)	custom dataset labels

© Copyright 2026 DEEPX All Rights Reserved.

Key	Meaning	Typical use
`strides`	output strides of the Python PPU decoders (`YOLOv5PPUPostprocessor`, `YOLOv7PPUPostprocessor`); the Python YOLOv5/YOLOX/YOLOv8 decoders derive or fix their strides	PPU variants with non-standard heads

Example for a custom 3-class YOLOv5-style model:

```
{
  "config": {
    "obj_threshold": 0.25,
    "score_threshold": 0.30,
    "nms_threshold": 0.45,
    "num_classes": 3,
    "class_names": ["person", "vehicle", "animal"]
  }
}
```

NOTE

Custom anchors are not read from `config.json`: C++ uses the postprocessor presets, and Python variants that need them set them in the variant's `custom_ops.py` (e.g., `yolov7-w6_1280x1280_nodecode`).

Example for a custom YOLOv8-style model:

```
{
  "config": {
    "score_threshold": 0.30,
    "nms_threshold": 0.45,
    "num_classes": 3,
    "class_names": ["person", "vehicle", "animal"]
  }
}
```


© Copyright 2026 DEEPX All Rights Reserved.

> **Warning**
>
> `num_classes` must match the compiled model output. If the model was compiled with a different class count, changing only `config.json` will not fix the tensor shape.

9.3.4 [Step 4] Review the Factory Code

Factories assemble the preprocessor, postprocessor, visualizer, and model metadata. For YOLO detection models they normally use `LetterboxPreprocessor` / `DetectionPreprocessor`, a YOLO postprocessor, and `DetectionVisualizer`.

C++ YOLOv8-style factory pattern:

```
PostprocessorPtr<DetectionResult> createPostprocessor(
    int input_width, int input_height, bool is_ort_configured = false) override {
    return std::make_unique<YOLOv8Postprocessor>(
        input_width, input_height,
        score_threshold_, nms_threshold_,
        is_ort_configured,
        num_classes_,
        class_names_
    );
}

void loadConfig(const dxapp::ModelConfig& config) override {
    score_threshold_ = config.get<float>("score_threshold", score_threshold_);
    nms_threshold_ = config.get<float>("nms_threshold", nms_threshold_);
    class_names_ = config.get_string_list("class_names");
    num_classes_ = config.get<int>("num_classes", num_classes_);
}
```

Python YOLOv8-style pattern: the postprocessor class is set in the variant's `config.json`, and the family factory builds it through `_build("postprocessor", ...)`:

```
"postprocessor": {
  "class": "YOLOv8Postprocessor",
  "args": [{"token": "input_width"}, {"token": "input_height"}, {"token": "config"}]
}
```


© Copyright 2026 DEEPX All Rights Reserved.

Check these points after generation:

- `get_model_name()` returns the new model name.
- `get_task_type()` remains `object_detection` for detection models.
- C++ and Python use the same postprocessor family.
- Thresholds and class metadata are loaded from `config.json` (C++: only with `--config`).
- Specialized tasks use the correct task-specific visualizer and runner.

9.3.5 [Step 5] Understand Output Shape Compatibility

YOLO postprocessors support common DX-APP output patterns, but a converted model may still differ.

Typical YOLOv5/v7-style outputs:

- single decoded tensor: `[1, N, 5 + num_classes]`
- multi-scale NPU tensors: `[1, C, H, W]`, where `C = anchors * (5 + num_classes)`
- objectness is present and combined with class scores

Typical YOLOv8+ detection outputs:

- anchor-free detection tensors
- no separate objectness field
- postprocessor uses class score and NMS thresholds

If verification fails due to unexpected tensor shape:

1. run with `--show-log` if the example supports verbose logs,
2. enable tensor dumping with `--dump-tensors`,
3. inspect output tensor names, dimensions, and value ranges,
4. compare with the selected reference model,
5. adjust the factory/config only if the output family is compatible,
6. implement a focused postprocessor variant only when the tensor contract is genuinely different.

9.3.6 [Step 6] Build and Run

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Build the generated targets:

```
./build.sh --type debug --target <variant>_sync
./build.sh --type debug --target <variant>_async
```

Run C++ examples:

```
./bin/<variant>_sync  -m assets/models/<variant>.dxnn -i sample/img/sample_kitchen.jpg --no-display -l 1 --config src/cpp_example/object_detection/<family>/<variant>/config.json
./bin/<variant>_async -m assets/models/<variant>.dxnn -v assets/videos/dance-group.mov --no-display -l 1 --config src/cpp_example/object_detection/<family>/<variant>/config.json
```

Run Python examples:

```
python src/python_example/object_detection/<family>/<variant>/<variant>_sync.py \
  --model assets/models/<variant>.dxnn \
  --image sample/img/sample_kitchen.jpg \
  --no-display \
  --loop 1

python src/python_example/object_detection/<family>/<variant>/<variant>_async.py \
  --model assets/models/<variant>.dxnn \
  --video assets/videos/dance-group.mov \
  --no-display \
  --loop 1
```

Use saved output for visual checks:

```
./bin/<variant>_sync -m assets/models/<variant>.dxnn -i sample/img/sample_kitchen.jpg --save --config src/cpp_example/object_detection/<family>/<variant>/config.json
```

9.3.7 [Step 7] Verify Before Registry Integration

Before registering a new YOLO model as a maintained example, verify it in both languages.

Recommended sequence:

```
# Registry-driven validation (interactive; all registered models)
./scripts/dx_tool.sh validate

# Targeted build
./build.sh --type debug --target <variant>_sync
./build.sh --type debug --target <variant>_async

# CLI smoke tests
```


© Copyright 2026 DEEPX All Rights Reserved.

```
./scripts/dx_tool.sh run --lang cpp --filter <variant>
./scripts/dx_tool.sh run --lang py --filter <variant>

# Broader checks when assets are available
./run_tc.sh --cpp --cli
./run_tc.sh --python
```

For numerical validation, add or update rules only after the visual and CLI checks are stable. Avoid checking in model metadata that points to missing `.dxnn` assets.

9.3.8 [Step 8] Update Registry and Test Metadata

When the model is intended to ship as a first-class example, review the registry and `test_models.conf` rows that `add_model.sh` added, and update the rest together:

- `config/model_registry.json`
- `config/test_models.conf`
- C++ example directory
- Python example directory
- verification rules if numerical checks are required
- documentation or release notes if this model is user-facing

A registry entry should identify:

- model name,
- `.dxnn` file name,
- task/category,
- postprocessor family,
- input size,
- support status,
- any required config metadata.

Do not add registry entries for local experiments or project-specific artifacts unless the release explicitly includes them.

9.3.9 Troubleshooting

ⓒ Copyright 2026 DEEPX All Rights Reserved.

No detections appear

- Lower `score_threshold` temporarily.
- Check whether the selected postprocessor family matches the output tensor format.
- Confirm `num_classes` matches the model output.
- Confirm preprocessing uses letterbox behavior expected by the model.

Boxes are shifted or scaled incorrectly

- Check model input width/height.
- Check whether the model expects letterbox preprocessing or direct resize.
- Compare C++ and Python outputs on the same image.
- Verify box scaling uses the `PreprocessContext` from the active preprocessor.

NMS removes too many boxes

- Increase or decrease `nms_threshold` depending on overlap behavior.
- Lower `score_threshold` to check whether candidate boxes exist before NMS.

C++ and Python results differ

- Ensure both configs contain the same thresholds, class count, and labels, and that the C++ run passes its `config.json` with `--config`.
- Ensure both factories use the same postprocessor family.
- Run one image with display disabled and compare saved output or serialized verification output.
- Check whether one path uses a C++ binding variant while the other uses pure Python postprocess.

Build target is missing

- Confirm the family directory is `<family>/` and the variant directory is the `.dxnn` stem.
- Run `./build.sh --target list` to confirm the target exists.
- Check that CMake discovered the generated example directory.

9.3.10 Release Checklist

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Before merging a YOLO customization:

- [ ] The selected postprocessor family matches the model output tensor contract.
- [ ] C++ and Python factories use matching preprocessing/postprocessing behavior.
- [ ] `config.json` has correct thresholds and `num_classes`.
- [ ] The model runs on at least one image input.
- [ ] Async execution runs on image or video input as appropriate for the task.
- [ ] Saved visualization looks correct.
- [ ] `dx_tool.sh validate` passes.
- [ ] Targeted build passes for sync and async targets.
- [ ] Registry/test metadata is updated only for release-supported models.
- [ ] Documentation or release notes are updated for user-facing changes.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

9.4 DX-APP Agent-Driven Development Guide

9.4.1 Overview

DX-APP supports DEEPX agent-driven development (dx-agent-dev) for building standalone inference applications on DEEPX NPU accelerators. Instead of manually writing boilerplate, you describe what you want in natural language and a network of specialized agents generates production-ready inference code, validates it, and reports results.

This guide covers the agent architecture, available skills, the validation framework, and troubleshooting for dx_app standalone inference development.

9.4.2 Agent Architecture

Six agents collaborate to build, validate, and manage dx_app inference applications.

Agent	Description	Routes To
`dx-app-builder`	Master router — classifies request type and dispatches to specialist agent	`dx-python-builder` `dx-cpp-builder` `dx-benchmark-builder` `dx-model-manager`
`dx-python-builder`	Builds Python inference apps in 4 variants: `sync`, `async`, `sync_cpp_postprocess`, `async_cpp_postprocess` (Sub-agent — invoked by dx-app-builder)	—
`dx-cpp-builder`	Builds C++ inference apps using the `InferenceEngine` API (Sub-agent — invoked by dx-app-builder)	—
`dx-benchmark-builder`	Benchmarks and profiles inference performance on target hardware (Sub-agent — invoked by dx-app-builder)	`dx-python-builder` (rebuild as a faster variant)

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Agent	Description	Routes To
dx-model-manager	Downloads, registers, and manages .dxnn compiled models (Sub-agent — invoked by dx-app-builder)	—
dx-validator	Validates generated app code and .deepx/ framework integrity	—

Routing Flow

```
User Request
    │
    ▼
dx-app-builder  (classifies intent)
    │
    ├──► dx-python-builder    (Python inference app)
    ├──► dx-cpp-builder       (C++ inference app)
    ├──► dx-benchmark-builder (performance profiling)
    └──► dx-model-manager     (model operations)
            │
            ▼
      validate_app.py  (run by the builder after generation)
```

9.4.3 Skills

Skills encapsulate reusable workflows that agents invoke during code generation.

General SWE Process

Skill	Description
dx-agent-brainstorm	Brainstorm and plan before any code generation
dx-agent-tdd	Test-driven development — validate each file immediately after creation
dx-agent-verify	Verify before claiming completion — evidence before assertions

© Copyright 2026 DEEPX All Rights Reserved.

Skill	Description
dx-swe-writing-plans	Write structured implementation plans before touching code
dx-swe-executing-plans	Execute a written implementation plan with review checkpoints
dx-swe-debugging	Systematic debugging — diagnose before proposing fixes
dx-swe-parallel-agents	Dispatch 2+ independent tasks to parallel agents
dx-swe-subagent-dev	Execute implementation plans with independent sub-agents
dx-swe-receiving-review	Receive and process code review feedback with technical rigor
dx-swe-requesting-review	Request code review to verify work meets requirements
dx-skill-router	Route tasks to the appropriate skill automatically

DEEPX Build

Skill	Description
dx-agent-app-build-python	Build a Python inference app in any of the 4 variants using the IFactory pattern
dx-agent-app-build-cpp	Build a C++ inference app with the `InferenceEngine` runtime API
dx-agent-app-build-async	Build an async high-performance app with pipelined pre/infer/post stages
dx-agent-app-model-management	Download `.dxnn` models from the registry and configure model paths
dx-agent-app-validate	Run the validation levels (static, config, component, NPU smoke, output accuracy,

© Copyright 2026 DEEPX All Rights Reserved.

Skill	Description
	cross-validation, integration) against generated code

Harness

Skill	Description
`dx-harness-writing-skills`	Create and edit skill files
`dx-harness-validate`	Validate this repo's `.deepx/` integrity

9.4.4 Supported AI Tools

dx_app agent-driven development works with five AI coding tools. Each auto-loads the knowledge base through its own configuration.

Tool	Config Files	Agents Available
Claude Code	`CLAUDE.md` 6 agents in `.claude/agents/` 18 skills in `.claude/skills/`	`@dx-app-builder`, `@dx-validator` (+ 4 sub-agents via handoff)

GitHub Copilot	`.github/copilot-instructions.md`
	6 agents in `.github/agents/`
	18 skills in `.github/skills/`
	4 instructions in `.github/instructions/`
		`@dx-app-builder`, `@dx-validator`
		(`dx-python-builder`, `dx-cpp-builder`,
		`dx-benchmark-builder`, `dx-model-manager`
		are sub-agents reached through `@dx-app-builder`)

Cursor	`.cursor/rules/dx-app.mdc` (always) 6 agent rules + 18 skill rules `python-example.mdc`, `cpp-example.mdc`, `tests.mdc` (28 total)	Free-form with auto-applied rules
OpenCode		`@dx-app-builder` or `/dx-agent-app-build-python`

© Copyright 2026 DEEPX All Rights Reserved.

Tool	Config Files	Agents Available
	`AGENTS.md`, `opencode.json` 6 agents in `.opencode/agents/` 18 skills in `.deepx/skills/`	
Codex CLI	`AGENTS.md` `.codex/skills/dx-codex-identity/SKILL.md`	Free-form with auto-loaded instructions

Copilot File-Specific Instructions

When editing files matching these globs, Copilot automatically injects context-specific instructions:

Glob Pattern	Injected Instruction	Content
`src/python_example/**`	`python-example.instructions.md`	IFactory pattern, `parse_common_args()`, 4-variant naming, no relative imports
`src/cpp_example/**`	`cpp-example.instructions.md`	C++14 standard, RAII patterns, InferenceEngine API
`src/postprocess/**`	`postprocess.instructions.md`	Postprocessing conventions, pybind bindings
`tests/**`	`tests.instructions.md`	pytest markers (`slow`, `smoke`), validation references

OpenCode Skills (Slash Commands)

General SWE Process

Slash Command	Description
`/dx-agent-brainstorm`	Brainstorm and plan before code generation
`/dx-agent-tdd`	Test-driven development with incremental validation
`/dx-agent-verify`	

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Slash Command	Description
	Verify completion with evidence before assertions
/dx-swe-writing-plans	Write structured implementation plans
/dx-swe-executing-plans	Execute implementation plans with review checkpoints
/dx-swe-debugging	Systematic debugging — diagnose before fixing
/dx-swe-parallel-agents	Dispatch independent tasks to parallel agents
/dx-swe-subagent-dev	Execute plans with independent sub-agents
/dx-swe-receiving-review	Receive code review feedback with technical rigor
/dx-swe-requesting-review	Request code review to verify work
/dx-skill-router	Route tasks to the appropriate skill

DEEPX Build

Slash Command	Description
/dx-agent-app-build-python	Step-by-step Python app generation with IFactory
/dx-agent-app-build-cpp	C++ app with InferenceEngine
/dx-agent-app-build-async	Async high-performance app
/dx-agent-app-model-management	Model download and registry
/dx-agent-app-validate	Run the validation levels (static, config, component, NPU smoke, output accuracy, cross-validation, integration)

Harness

Slash Command	Description
/dx-harness-writing-skills	Create and edit skill files

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Slash Command	Description
/dx-harness-validate	Validate this repo's .deepx/ integrity

Platform File Loading Reference

Each AI coding agent auto-loads different configuration files at the dx_app level.

Auto-Loaded Files

File	Auto-loaded by	Loading
.github/copilot-instructions.md	Copilot Chat/CLI	Auto
CLAUDE.md	Claude Code	Auto
AGENTS.md + opencode.json	OpenCode	Auto
AGENTS.md + .codex/skills/dx-codex-identity/SKILL.md	Codex CLI	Auto
.cursor/rules/dx-app.mdc	Cursor	Auto
.cursor/rules/python-example.mdc	Cursor	Auto (glob)
.cursor/rules/cpp-example.mdc	Cursor	Auto (glob)
.cursor/rules/tests.mdc	Cursor	Auto (glob)

Agent Files (Manual @mention)

Agent	Copilot (@mention)	OpenCode (@mention)	Claude Code (.claude/agents/)
dx-app-builder	.github/agents/dx-app-builder.agent.md	.opencode/agents/dx-app-builder.md	.claude/agents/dx-app-builder.md
dx-python-builder	.github/agents/dx-python-builder.agent.md	.opencode/agents/dx-python-builder.md	.claude/agents/dx-python-builder.md
dx-cpp-builder	.github/agents/dx-cpp-builder.agent.md	.opencode/agents/dx-cpp-builder.md	.claude/agents/dx-cpp-builder.md

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Agent	Copilot (@mention)	OpenCode (@mention)	Claude Code (.claude/agents/)
dx-benchmark-builder	.github/agents/dx-benchmark-builder.agent.md	.opencode/agents/dx-benchmark-builder.md	.claude/agents/dx-benchmark-builder.md
dx-model-manager	.github/agents/dx-model-manager.agent.md	.opencode/agents/dx-model-manager.md	.claude/agents/dx-model-manager.md
dx-validator	.github/agents/dx-validator.agent.md	.opencode/agents/dx-validator.md	.claude/agents/dx-validator.md

Copilot File-Specific Instructions

File	Scope
.github/instructions/python-example.instructions.md	Python example files
.github/instructions/cpp-example.instructions.md	C++ example files
.github/instructions/postprocess.instructions.md	Postprocess files
.github/instructions/tests.instructions.md	Test files

Skill Files (All Platforms — /slash-command)

Skill	File
/dx-agent-brainstorm	.deepx/skills/dx-agent-brainstorm/SKILL.md
/dx-agent-app-build-async	.deepx/skills/dx-agent-app-build-async/SKILL.md
/dx-agent-app-build-cpp	.deepx/skills/dx-agent-app-build-cpp/SKILL.md
/dx-agent-app-build-python	.deepx/skills/dx-agent-app-build-python/SKILL.md
/dx-swe-parallel-agents	.deepx/skills/dx-swe-parallel-agents/SKILL.md
/dx-swe-executing-plans	.deepx/skills/dx-swe-executing-plans/SKILL.md
/dx-agent-app-model-management	.deepx/skills/dx-agent-app-model-management/SKILL.md

© Copyright 2026 DEEPX All Rights Reserved.

Skill	File
/dx-swe-receiving-review	.deepx/skills/dx-swe-receiving-review/SKILL.md
/dx-swe-requesting-review	.deepx/skills/dx-swe-requesting-review/SKILL.md
/dx-skill-router	.deepx/skills/dx-skill-router/SKILL.md
/dx-swe-subagent-dev	.deepx/skills/dx-swe-subagent-dev/SKILL.md
/dx-swe-debugging	.deepx/skills/dx-swe-debugging/SKILL.md
/dx-agent-tdd	.deepx/skills/dx-agent-tdd/SKILL.md
/dx-agent-app-validate	.deepx/skills/dx-agent-app-validate/SKILL.md
/dx-agent-verify	.deepx/skills/dx-agent-verify/SKILL.md
/dx-swe-writing-plans	.deepx/skills/dx-swe-writing-plans/SKILL.md
/dx-harness-writing-skills	.deepx/skills/dx-harness-writing-skills/SKILL.md
/dx-harness-validate	.deepx/skills/dx-harness-validate/SKILL.md

Shared Knowledge Base (.deepx/)

The .deepx/ directory is the canonical source of truth for all agent knowledge. Platform-specific files (.github/agents|skills|instructions/ , .github/copilot-instructions.md , .claude/ , .cursor/rules/ , .opencode/agents/ , CLAUDE.md , AGENTS.md and their -KO twins) are generated from .deepx/ by dx-agent-gen generate , which ships with dx-all-suite. It is not auto-loaded as a whole (OpenCode's opencode.json loads only .deepx/memory/common_pitfalls.md and .deepx/instructions/coding-standards.md); agents and skills read specific files as needed during task execution.

Directory	Files	Description
.deepx/agents/	6 files (dx-app-builder.md , dx-python-builder.md , dx-cpp-builder.md , dx-benchmark-builder.md , dx-model-manager.md , dx-validator.md)	Authoritative agent definitions
.deepx/skills/	18 directories	Detailed skill workflows (one SKILL.md per directory)

© Copyright 2026 DEEPX All Rights Reserved.

Directory	Files	Description
`.deepx/templates/`	6 template files	Instruction templates for the platform files (processed by `dx-agent-gen`)
`.deepx/knowledge/`	1 file	Domain-specific reference material
`.deepx/toolsets/`	6 files	API references and schema documentation
`.deepx/instructions/`	6 files	Coding standards and workflow rules
`.deepx/memory/`	5 files	Persistent knowledge — pitfalls and session memory
`.deepx/contextual-rules/`	4 files	Scope-aware coding rules (C++ examples, Python examples, postprocess, tests)
`.deepx/prompts/`	4 files	Reusable prompt templates
`.deepx/scripts/`	3 files	Validation and harness scripts
`.deepx/docs/`	2 files	CI sub-repo gate guide (EN/KO)

9.4.5 User Scenarios

Scenario 1: Build a Python Detection App

Prompt:

```
"Build a yolo26n person detection app using Python"
```

Tool	How to Use
Claude Code	Type the prompt directly. `CLAUDE.md` runs `/dx-skill-router` first and hands app builds to the `dx-app-builder` agent (`.claude/agents/`), which asks the 3 mandatory questions

© Copyright 2026 DEEPX All Rights Reserved.

Tool	How to Use
	(language/variant, AI task, model), loads the `dx-agent-app-build-python` skill, generates files in `dx-agent-dev/<session_id>/` (or `src/...` if explicitly requested), and validates.
GitHub Copilot	`@dx-app-builder` followed by the prompt. Routes to `dx-python-builder`, generates the requested variants and runs `validate_app.py`.
Cursor	Type the prompt directly. `dx-app.mdc` (always loaded) provides context. `python-example.mdc` activates for `src/python_example/` files.
OpenCode	`@dx-app-builder` followed by the prompt, or `/dx-agent-app-build-python` skill directly.

Scenario 2: Build a C++ App

Prompt:

```
"Build a C++ inference app for yolo26n using InferenceEngine"
```

Tool	How to Use
Claude Code	Type the prompt directly. `CLAUDE.md` hands the build to the `dx-app-builder` agent, which loads the `dx-agent-app-build-cpp` skill.
GitHub Copilot	`@dx-app-builder` followed by the prompt. Routes to `dx-cpp-builder`.
Cursor	Type the prompt directly. `cpp-example.mdc` activates for `src/cpp_example/` files, injecting C++14 and RAII conventions.
OpenCode	`@dx-app-builder` followed by the prompt, or `/dx-agent-app-build-cpp` skill directly.

Scenario 3: Download and Register a Model

© Copyright 2026 DEEPX All Rights Reserved.

Prompt:

> "Download yolo26n model for DX-M1"

Tool	How to Use
Claude Code	`@dx-app-builder` followed by the prompt. Routes to `dx-model-manager`.
GitHub Copilot	`@dx-app-builder` followed by the prompt. Routes to `dx-model-manager`.
Cursor	Type the prompt directly.
OpenCode	`@dx-app-builder` followed by the prompt, or `/dx-agent-app-model-management` skill.

Scenario 4: Validate Generated Code

Prompt:

> "Validate the detection app I just created"

Tool	How to Use
Claude Code	`@dx-validator` followed by the prompt.
GitHub Copilot	`@dx-validator` followed by the prompt.
Cursor	Type the prompt directly.
OpenCode	`@dx-validator` followed by the prompt, or run manually: `python .deepx/scripts/validate_app.py dx-agent-dev/<session_id>/`

Scenario 5: Build a Pose Estimation App

Prompt:

ⓒ Copyright 2026 DEEPX All Rights Reserved.

"Build a pose estimation app with yolo26n-pose"

Tool	How to Use
Claude Code	Type the prompt directly. `CLAUDE.md` hands the build to the `dx-app-builder` agent, which loads the `dx-agent-app-build-python` skill with the `pose_estimation` task type. Generates keypoint visualization and skeleton drawing logic.
GitHub Copilot	`@dx-app-builder` followed by the prompt. Routes to `dx-python-builder` with pose-specific postprocessing.
Cursor	Type the prompt directly. `dx-app.mdc` (always loaded) provides context; `python-example.mdc` also applies when you place the app under `src/python_example/pose_estimation/`.
OpenCode	`@dx-app-builder` followed by the prompt, or `/dx-agent-app-build-python` skill directly.

Scenario 6: Build an Instance Segmentation App

Prompt:

"Build an instance segmentation app with yolo26n-seg"

Tool	How to Use
Claude Code	Type the prompt directly. `CLAUDE.md` hands the build to the `dx-app-builder` agent, which loads the `dx-agent-app-build-python` skill with the `instance_segmentation` task type. Generates mask overlay visualization.
GitHub Copilot	`@dx-app-builder` followed by the prompt. Routes to `dx-python-builder` with segmentation-specific postprocessing.
Cursor	

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Tool	How to Use
	Type the prompt directly. `dx-app.mdc` (always loaded) provides context; `python-example.mdc` also applies when you place the app under `src/python_example/instance_segmentation/`.
OpenCode	`@dx-app-builder` followed by the prompt, or `/dx-agent-app-build-python` skill directly.

Scenario 7: Build a Classification App

Prompt:

```
"Build an image classification app with EfficientNet-Lite0"
```

Tool	How to Use
Claude Code	Type the prompt directly. `CLAUDE.md` hands the build to the `dx-app-builder` agent, which loads the `dx-agent-app-build-python` skill with the `image_classification` task. Generates top-K label prediction logic.
GitHub Copilot	`@dx-app-builder` followed by the prompt. Routes to `dx-python-builder` with classification postprocessing (softmax + top-K).
Cursor	Type the prompt directly. `dx-app.mdc` (always loaded) provides context; `python-example.mdc` also applies when you place the app under `src/python_example/image_classification/`.
OpenCode	`@dx-app-builder` followed by the prompt, or `/dx-agent-app-build-python` skill directly.

Scenario 8: Build an Async High-Performance App

Prompt:

ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
"Build an async high-performance detection app with yolo26n"
```

Tool	How to Use
Claude Code	Type the prompt directly. `CLAUDE.md` hands the build to the `dx-app-builder` agent, which loads the `dx-agent-app-build-async` skill. Generates pipelined pre/infer/post stages with queue-based parallelism.
GitHub Copilot	`@dx-app-builder` followed by the prompt. Routes to `dx-python-builder` with async variant focus.
Cursor	Type the prompt directly. `dx-app.mdc` (always loaded) provides context; `python-example.mdc` also applies when you place the app under `src/python_example/`.
OpenCode	`@dx-app-builder` followed by the prompt, or `/dx-agent-app-build-async` skill directly.

9.4.6 Quick Start

Request a person detection app in natural language:

```
@dx-app-builder "Build a Python person detection app using yolo26n"
```

The agent will:

1. **Ask the 3 mandatory questions** — language/variant (Python `sync` / `async` / `cpp_postprocess` or C++), AI task (`detection`, `classification`, `segmentation`, etc.), model
2. **Present a build plan** — list of files to generate, model to download, config to write
3. **Route to `dx-python-builder`** — the specialist agent takes over
4. **Generate files** in `dx-agent-dev/<session_id>/` (or `src/` if explicitly requested)
5. **Validate and report** — the builder runs `validate_app.py`, runs the sync example once if an NPU is present, and prints a completion report

Mandatory Questions (HARD-GATE)

© Copyright 2026 DEEPX All Rights Reserved.

When using `@dx-app-builder`, the agent enforces 3 mandatory questions before generating any code:

1. **Language/variant**: Python (sync / async / cpp_postprocess / async_cpp_postprocess) or C++?
2. **AI task**: detection, classification, segmentation, pose, etc.
3. **Model**: Specific model name (e.g., `yolo26n`) or auto-recommend

These questions are **non-skippable** — even if your prompt provides enough context, the agent will confirm each decision explicitly before proceeding.

9.4.7 What Gets Created

By default, agent-generated code is placed in the `dx-agent-dev/` isolation directory to prevent conflicts with existing source code.

Default Output (dx-agent-dev/)

```
dx-agent-dev/<session_id>/
├── README.md              # Session metadata and run instructions
├── session.json           # Machine-readable session config
├── setup.sh               # Environment setup script (mandatory)
├── run.sh                 # App launch script (mandatory)
├── session.log            # Agent session log (mandatory)
├── __init__.py
├── config.json
├── factory/
│   ├── __init__.py
│   └── {model}_factory.py
├── {model}_sync.py
├── {model}_async.py                # if requested
├── {model}_sync_cpp_postprocess.py   # if requested
└── {model}_async_cpp_postprocess.py  # if requested
```

`{model}_sync.py` is always generated; the other three variants are generated when you ask for them.

Session ID format: `YYYYMMDD-HHMMSS_<agent>_<coding_model>_<target_model>_<task>` where `<agent>` is `claude`, `codex`, `copilot`, `cursor`, or `opencode` (e.g., `20260403-143022_claude_sonnet46_yolo26n_detection`). The timestamp is the system local time.

Production Output (src/)

© Copyright 2026 DEEPX All Rights Reserved.

When you explicitly request production placement, files are written directly to `src/python_example/{task}/{family}/{variant}/` — the standard source tree (`{variant}` is the `.dxnn` stem).

File Descriptions

File	Purpose
`config.json`	Inference thresholds read by the factory (e.g. `score_threshold`, `nms_threshold`, `top_k`); the model path comes from `--model` or `model_registry.json`
`{model}_factory.py`	Implements `IFactory`: `create_preprocessor`, `create_postprocessor`, `create_visualizer`, `get_model_name`, `get_task_type`
`{model}_sync.py`	Synchronous single-threaded inference entry point
`{model}_async.py`	Asynchronous pipelined inference entry point
`{model}_sync_cpp_postprocess.py`	Sync inference with C++ post-processing via pybind
`{model}_async_cpp_postprocess.py`	Async inference with C++ post-processing via pybind

NOTE

`setup.sh`, `run.sh`, and `session.log` are mandatory artifacts in every session output directory

9.4.8 Validation Levels

The `dx-agent-app-validate` skill defines these levels. Levels 1–3 run without an NPU; Levels 4–6 need the NPU and the model file.

Level	What It Checks	Requires Hardware
1 — Static	Syntax, imports, IFactory methods,	No

© Copyright 2026 DEEPX All Rights Reserved.

Level	What It Checks	Requires Hardware
	`parse_common_args()`, no hard-coded paths	
2 — Config	`config.json` is valid JSON and has the task's keys; the model has a `model_registry.json` entry	No
3 — Component	Factory creates the preprocessor, postprocessor and visualizer; OpenCV has a GUI backend; the app runs when copied outside the suite	No
4 — Smoke	Single-frame inference on the NPU	Yes
5 — Output Accuracy	Detections > 0, valid boxes, scores and class IDs, postprocessor matches the model family	Yes
5.5 — Cross-Validation	Comparison with a precompiled reference model or an existing verified example (skipped if neither exists)	Yes
6 — Integration	Full pipeline on video with `--save`	Yes

9.4.9 Validation Commands

```
# Static checks (11 checks: required files, syntax, relative imports, logging, hard-coded paths,
IFactory, parse_common_args, unused imports, README, app.yaml, unreachable code)
python .deepx/scripts/validate_app.py dx-agent-dev/<session_id>/

# Add the 3 smoke tests (--help, factory import, NPU status) — run after the session's setup.sh
python .deepx/scripts/validate_app.py dx-agent-dev/<session_id>/ --smoke-test

# Framework integrity — verify .deepx/ directory structure
python .deepx/scripts/validate_framework.py
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

> **NOTE**
>
> `validate_app.py` checks a session output directory; source-tree variant directories have no `__init__.py` and fail the required-files check. Other options: `--strict`, `--json`, `--verbose`.

9.4.10 Knowledge Base Structure

Agent knowledge lives in the `.deepx/` directory at the dx_app project root.

Directory	Count	Contents
`agents/`	6	Agent definitions and routing rules
`skills/`	18	Skill workflow directories (one `SKILL.md` each)
`templates/`	6	Instruction templates for the platform files (processed by `dx-agent-gen`)
`knowledge/`	1	Domain-specific reference material
`toolsets/`	6	API references (InferenceEngine, common framework / IFactory, `.dxnn` model format, model registry, pybind postprocess, OCR app)
`memory/`	5	Persistent knowledge (index, common pitfalls, model zoo, performance patterns, platform API)
`contextual-rules/`	4	Scope-aware rules for C++ examples, Python examples, postprocess, and tests
`prompts/`	4	Reusable prompt templates (new C++ app, Python

© Copyright 2026 DEEPX All Rights Reserved.

Directory	Count	Contents
		detection/segmentation, orchestrated build)
scripts/	3	validate_app.py , validate_framework.py , harness_bootstrap.sh
docs/	2	CI sub-repo gate guide (EN/KO)

NOTE

Platform files (.github/agents\|skills\|instructions/ , .github/copilot-instructions.md , .claude/ , .cursor/rules/ , .opencode/agents/ , CLAUDE.md , AGENTS.md and their -KO twins) are generated by dx-agent-gen generate , which ships with dx-all-suite. Do not edit them directly. Edit .deepx/ source files and re-run the generator. In a standalone dx_app checkout, run bash .deepx/scripts/harness_bootstrap.sh --check before editing .deepx/ .

Agents read from these directories at task start. Memory files are updated when new patterns or fixes are discovered during development.

9.4.11 Session Sentinels

Agents output fixed markers at the start and end of each task for automated testing:

Marker	When
[DX-AGENT-DEV: START]	First line of the agent's response
[DX-AGENT-DEV: DONE (output-dir: <relative_path>)]	Last line after all work is complete. <relative_path> is the session output directory relative to the project root. If no files were generated, omit the (output-dir: ...) part.

Sub-agents invoked via handoff do not output sentinels — only the top-level agent does.

© Copyright 2026 DEEPX All Rights Reserved.

Rules:

1. **Required** — Output `[DX-AGENT-DEV: START]` as the absolute first line of the first response, before any other text, tool call, or reasoning. It cannot be skipped even if the user says "just proceed" — automated tests fail without it.
2. Output `[DX-AGENT-DEV: DONE (output-dir: <path>)]` as the very last line after all work, validation, and file generation is complete.
3. If you are a sub-agent invoked via handoff/routing, do NOT output sentinels — only the top-level agent outputs them.
4. If the user sends multiple prompts in a session, output START/DONE for each prompt.
5. The `output-dir` in DONE must be the relative path from the project root to the session output directory.
6. **Never output DONE after only producing planning artifacts** (specs, plans, design documents). DONE means all deliverables are produced — implementation code, scripts, configs, and validation results.
7. Right after START and right before DONE the agent prints the DEEPX banner; the banner never moves the sentinel lines.
8. Before DONE the agent checks that every mandatory file exists in the session directory.

9.4.12 Troubleshooting

- **Problem:** Agent writes relative imports (`from .factory import ...`)
  Cause: Default LLM behavior
  Solution: Use absolute imports only.
  Example: `from common.processors import YOLOv8Postprocessor`

- **Problem:** Factory class missing methods
  Cause: Incomplete `IFactory` implementation
  Solution: Implement all 5 required methods:
  `create_preprocessor` , `create_postprocessor` , `create_visualizer` , `get_model_name` , `get_task_type`

- **Problem:** Model not found at runtime
  Cause: `.dxnn` file path not registered
  Solution: Query `model_registry.json` via `dx-model-manager` to download and register the

ⓒ Copyright 2026 DEEPX All Rights Reserved.

model.

- **Problem:** NPU not available / device error
  Cause: Accelerator not detected by driver
  Solution: Run `dxcli -s` to check device status and verify the DEEPX kernel module is loaded (`dxrt-cli -s` is the same tool).

- **Problem:** `validate_app.py` fails immediately
  Cause: The target directory has no `__init__.py`, `config.json` or `factory/`, or (with `--smoke-test`) `common` cannot be imported
  Solution: Run it from the dx_app root on a session directory (`dx-agent-dev/<session_id>/`), after the session's `setup.sh` has copied `./common` into it.

9.4.13 Further Reading

- DX-APP Architecture and Usage Reference
- DX-APP Python Example Usage Guide
- DX-APP C++ Example Usage Guide
- DX-APP Example Source Structure

© Copyright 2026 DEEPX All Rights Reserved.

10. Appendix

10.1 Third-Party License Notice

10.1.1 Third-Party Sample Model Licenses

Sample models included in **DX-APP** (distributed as `.dxnn` files) are provided for **evaluation and development purposes only** and are **not licensed for commercial deployment**.

These models are compiled from various third-party open-source projects, each subject to its own license terms (e.g., AGPL-3.0, GPL-3.0, non-commercial research licenses, etc.). Most of these licenses impose significant restrictions on commercial use, including but not limited to:

- Full source code disclosure obligations (Copyleft).
- Outright prohibition of commercial deployment.
- Mandatory attribution requirements.

For commercial use, users must either:

- **Obtain appropriate licenses** directly from the original model providers/authors, or
- **Use their own** commercially licensed or proprietary models.

10.1.2 Third-Party Sample Dataset Licenses

Sample images and videos (the "Datasets") included in **DX-APP** are either sourced from publicly available academic or open-source datasets, or **synthetically generated by AI**. They are provided strictly for **evaluation and demonstration purposes only**.

DEEPX does not claim ownership or grant any commercial rights to the third-party Datasets. Many publicly available datasets (e.g., ImageNet) are restricted to **non-commercial research and educational use** by their original creators. AI-generated

© Copyright 2026 DEEPX All Rights Reserved.

sample images contain no third-party content and are not subject to an external dataset license.

- **User Responsibility:** Users are solely responsible for identifying the original source of any sample data and complying with its specific license terms.
- **Redistribution:** Users must not further redistribute these sample images or use them in any commercial product or service without explicit permission from the original copyright holders.

10.1.3 General Disclaimer

DEEPX provides these third-party models and datasets "as-is" for the user's convenience. DEEPX is not responsible for any legal issues or damages arising from the unauthorized or commercial use of these resources. By using these samples, the user agrees to assume all legal and financial liability related to license compliance.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

11. RELEASE_NOTES

11.1 DX-APP v3.3.0 / 2026-10-23

11.1.1 1. Changed

- **Example Layout**: Reorganized C++ and Python examples into task/family/variant directories. The variant name is the .dxnn stem, and each C++ factory is namespaced `dxapp::v_<variant>`
- **Task Renames**:
  - `classification` → `image_classification`
  - `embedding` → split into `face_recognition`, `image_retrieval`, `visual_place_recognition`, `zero_shot_image_classification`
  - `attribute_recognition` → split into `person_attribute`, `face_attribute`
  - `reid` → split into `image_classification`, `person_reid` (new task)
  - `instance_segmentation` (FastSAM) → `zero_shot_instance_segmentation`
  - `obb_detection` → `oriented_object_detection`
  - `image_enhancement` → `low_light_enhancement`
  - `face_alignment` → `face_landmark`
- **PPU Models**: Moved PPU models under `object_detection/yolo_ppu`
- **Executable and Script Names**: Executables, build targets, and Python scripts are now named after the variant (e.g. `yolov9s_sync` → `yolov9-s_640x640_sync`). Update any scripts that call the old names
- **Classification Sample**: Replaced the `efficientnet-lite0_256x256` sample with `efficientnet-lite0_224x224`
- **Model Zoo Manifest**: Migrated the model-zoo manifest to the `2_5_0` URL hierarchy (`models.ver`, `modelzoo_manifest.json`)
- **DX-RT 3.5.0 Required**: Model Zoo 2_5_0 `.dxnn` files are container format v9, which needs DX-RT 3.5.0 or later; the C++ examples check the version at startup
- **`add_model.sh`**: Takes a variant (the `.dxnn` stem) as its first argument; adds `--family`, `--input-size`, `--dxnn`, and `--no-register`. `config/model_registry.json` rows gain `variant`, `family`, `task` (plus `task_legacy`, `image_only`, `zoo_canonical`, `alias_of`, `published`)

© Copyright 2026 DEEPX All Rights Reserved.

- **Model Package Pruning**: `scripts/extract_model_package.sh` now prunes `common/` to the files the extracted model actually depends on instead of copying the whole framework; use `--no-prune` for the previous full copy
- **Ubuntu 18.04 (EOL)**: Dropped Ubuntu 18.04 support; removed it from the Makefile, CI matrix, and installation guide

11.1.2 2. Fixed

- Fixed async runners calling the shared postprocessor from several dxrt callback threads at once. DXAPP_VERIFY records follow input order
- Fixed retrieval, re-id, and `--check` gallery paths that depended on the working directory
- Fixed SuperPoint keypoints being drawn at the wrong vertical position in the C++ example
- Fixed SuperPoint tracker settings (nn_thresh, max_pixel_dist, track_max_length) being ignored
- Fixed all 998 `*_cpp_postprocess.py` Python examples running the Python postprocessor instead of `dx_postprocess`: the layout regeneration had dropped their `on_engine_init`, so they behaved exactly like the plain sync/async scripts. Each installs its C++ class again, and `run_entry()` forwards `on_engine_init` to the runner
- Fixed `dx_postprocess` results that differed from the Python and C++ examples: `ClassificationPostProcess` applied softmax to outputs that already were a distribution and broke top-1 ties differently; `VitPosePostProcess` returned heatmap cells instead of model-input pixels (keypoints at 1/4 of their position); DAMO-YOLO and NanoDet NMS suppressed per class instead of class-agnostically; CenterPose ranked its heads and snapped keypoints differently
- Fixed Python example postprocessors ignoring the variant's `score_threshold` (the nested config block reached them without the `conf_threshold` alias, so they ran at their 0.3 default), and `YOLOv7PPUPostprocessor` decoding with the YOLOv5 anchors (boxes ~0.82x the C++ size)
- Fixed `dx_postprocess` rejecting numpy int64 arrays (spelled `"l"` on LP64 platforms), such as CLIP token ids
- Fixed 3D object detection C++ runners printing errors and logs without the `[DXAPP]` prefix
- Fixed `scripts/bench_models.sh` for the v3.3.0 layout: it read the 4-column `config/test_models.conf` as 3 columns and keyed models by family, so every row came out NO_MODEL; it now benchmarks each of the 497 variants

11.1.3 3. Added

- Multi-model graph engine and CLI `multi_model_graph_sync` / `multi_model_graph_async` (`--check` , `--list-models` , `--report`)

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- Registered 147 DX Model Zoo models, including RT-DETR, YOLOv12, YOLOv13, pre-optimized YOLO, anomaly detection, image matting, CLIP retrieval, person re-id, and visual place recognition
- Sample galleries for re-id, image retrieval, and visual place recognition
- `dx_postprocess` classes for the eight postprocessors that had none (50 → 58 pybind classes): `EfficientADPostProcess`, `AnomalyFeaturePostProcess` (PatchCore), `RTDETRPostProcess` (RT-DETR v1/v2/v3), `MaskRTDETRPostProcess`, `PreoptTopKPostProcess` (YOLO26 pre-optimized detection and pose), `PicoDetPostProcess` (PP-ShiTu mainbody), `PPMattingPostProcess` and `CLIPTextPostProcess`. Anomaly maps, masks and alpha mattes come back at the model's resolution and the examples resize them, since the module has no OpenCV

11.2 DX-APP v3.2.2 / 2026-08-14

11.2.1 1. Changed

- Moved SuperPoint point tracking out of the keypoint detection post-process into the visualizer
- Switched the depth-estimation demo from Depth-Anything-V2 to YOLO26-Depth-S.
- Use a new low-resolution source video(lowres-drone-city-road.mp4) for the super-resolution demo

11.2.2 2. Fixed

- Fixed the Windows all-build not running in parallel by copying shared files once instead of duplicating the copy per target
- Route C++ Async mode to the _async binary (For Windows)
- Fixed grammar errors in installation and build documentation

11.2.3 3. Added

- YOLO26-Depth examples for all five model sizes (n / s / m / l / x, 768x768): C++ sync and async, plus Python sync, async, sync_cpp_postprocess and async_cpp_postprocess
- Registered the 5 yolo26-depth models in `config/model_registry.json`, `scripts/modelzoo_manifest.json`
- Use low-resolution sample video for super-resolution demos (bump sample video archive to v3.2.2)

© Copyright 2026 DEEPX All Rights Reserved.

11.3 DX-APP v3.2.1 / 2026-08-03

11.3.1 1. Changed

- Updated all example commands in README and docs to the new model filename convention (e.g. YoloV9S.dxnn -> yolov9-s_640x640.dxnn)
- Super-resolution now saves both a side-by-side comparison (sr_input_output.jpg) and the standalone upscaled output.
- Replace the super-resolution sample image with genuinely low-resolution inputs (`sample_superresolution.png` removed)

11.3.2 2. Fixed

- Fixed RealESRGAN discolored output vs. C++: corrected color rounding and RGB->BGR channel order, and stopped routing 3-channel SR models through the luminance-only tiled path
- Drain the output queue and wait for all submitted frames before stopping the display thread in the C++ async runners, fixing the lost tail of a saved video
- Include `DXAPP_SAVE_IMAGE` in the render gate of the sync/async pose runners, fixing headless runs that wrote no image when only the env variable was set
- Feed ESPCN the BT.601 limited-range Y
- Remove tile seams in tiled super-resolution by cutting tiles with a halo of the model's receptive-field radius

11.3.3 3. Added

- `--sr-tile-halo` CLI option (0..4, default 4) to control tile overlap for tiled super-resolution examples

11.4 DX-APP v3.2.0 / 2026-07-22

11.4.1 1. Changed

- Image-only examples (embedding/reid/attribute) now show an input hint and reject stream inputs instead of exposing unused `--video/--camera/--rtsp` options
- Detection-family examples expose explicit `config.json` runtime knobs (score/nms/top_k/obj) with output-preserving defaults

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- Standardized DX-APP user-facing message level tags to `[DXAPP] [INFO]`, `[DXAPP] [WARN]`, and `[DXAPP] [ERROR]`.
- Replaced the OBB detection sample image (DOTA8 dataset → AI-generated satellite image) to remove licensing risk
- Added an AI-generated sample data clause to the Third-Party License document
- Migrated the model-zoo manifest to the `2_4_0` URL hierarchy (`models.ver`, `modelzoo_manifest.json`)
- Generalized float-input preprocessing (FLOAT dtype → normalized float32 buffer) via `runSyncInferenceTyped` / `fillModelInputBuffer`
- Default build mode changed to minimal in build.sh : automatically runs minimal build when no option is specified, with available options printed to console
- build.bat interactive prompt default changed to 1 minimal

11.4.2 2. Fixed

- Super-resolution now preserves input resolution by using dynamic tile padding instead of a fixed 20-tile downscale
- `dx_postprocess` pybind build fixed by adding missing `<cmath>` / `<numeric>` includes
- Invalid `--image` paths now print a sample-image hint and exit with code 1
- Fixed `--save` option not producing output image files in C++ sync runners for face, obb, pose, instance segmentation tasks
- Fixed the `inflight_current` decrement order in the async runner's `completeInflightMetrics`
- DOPE post-process: +0.5 pixel-center offset + normalized `[0,1]` output
- YOLOPv2 post-process: vehicle-only filtering, class-agnostic NMS, letterbox inverse-transform
- YOLOPv2 visualizer: driving-mask type mismatch (`int` → `CV_8UC1`)
- RetinaFace post-process: NHWC feature-map decode support (e.g. `retinaface_mobilenet_v1_736x1280`)
- Depth Anything V2 (vits/vitb/vitl): per-model ImageNet mean/std normalization via `getInputNormalization()`
- Windows x64 DXRT library path (`lib` → `lib/x64`) across postprocess CMakeLists + generate_build_bat.py
- Always link `Threads::Threads` for exported C++ example package
- Underscore-stripped fallback in E2E test model matching (e.g. `YoloV7W6` ↔ `yolov7_w6`)

ⓒ Copyright 2026 DEEPX All Rights Reserved.

11.4.3 3. Added

- DEEPX Agent-Driven Development (dx-agent-dev) — Beta. Generate standalone Python/C++ inference apps from plain language: an AI agent builds the IFactory-based app (preprocess → infer → postprocess → visualize) against the DEEPX model registry and runs it on the DX-M1 NPU.
- Native C++ post-processing for the example model zoo (YOLO families, semantic seg, Face3D, embedding/classification/attribute, restoration, YOLO-PPU), replacing Python fallbacks
- Opt-in `--fast-postprocess` path for object detection and instance segmentation
- YOLO Customizing Guide documentation
- Added Windows standalone Visual Studio solution package extraction workflow for DX-APP C++ examples, including automatic OpenCV/DXRT CMake dependency configuration.
- Added `--demo-models` setup/download option to download only models used by `run_demo.sh` / `run_demo.bat`
- Added DX-APP build selection options for minimal and category-based builds, plus Windows build selection TUI(Text User Interface).
- Added the agent knowledge base (`.deepx/`): specialized agents, app-building/SWE skills, and generated instruction files for Claude Code, GitHub Copilot, Cursor, OpenCode, and Codex CLI
- Support for 69 net-new .dxnn models (86 added / 17 removed) and 5 new AI task categories
- **Model registry**: 280 → 347 models across 22 AI task categories
- **New AI tasks (5)**: 3D Object Detection (SFA3D), Keypoint Detection (SuperPoint), Object Pose Estimation (DOPE), Panoptic Driving Perception (YOLOPv2), Hand Detection (MediaPipe Hand)
- **Post-processors (7)** + pybind bindings (43 → 50 classes): DOPE, MediaPipe Hand, RealESRGAN, SFA3D, SuperPoint, VitPose, YOLOPv2
- **C++ factory interfaces (3)**: `IObjectPoseFactory`, `IKeypointDetectionFactory`, `IPanopticDrivingFactory`
- **Visualizers (3)**: DOPE (solvePnP 3D cuboid), SuperPoint (keypoint-only), YOLOPv2 (lane + drivable-area overlay)
- Per-model Python examples (4 variants: sync / async / sync_cpp_postprocess / async_cpp_postprocess) and C++ examples (sync / async) for all new models
- build.sh: Added --all flag to support full build and install in a single command
- build.bat: Added CLI argument parsing (--minimal, --all, --category, -h/--help) for non-interactive builds

© Copyright 2026 DEEPX All Rights Reserved.

11.5 DX-APP v3.1.1 / 2026-05-06

11.5.1 1. Changed

- None

11.5.2 2. Fixed

- Fixed typo error in document(DX-APP User Manual)

11.5.3 3. Added

- Added License information for third-party models & datasets

---

11.6 DX-APP v3.1.0 / 2026-04-10

11.6.1 1. Changed

- Unified 5-layer architecture and design patterns across Python and C++ implementations
  - **App** Entry point -> `yolov5s_sync.cpp` : `yolov5s_sync.py`
  - **Runner** Pipeline orchestration (Sync/Async) -> `sync_detection_runner.hpp` : `sync_runner.py`
  - **Factory** Per-model component assembly -> `yolov5s_factory.hpp` : `yolov5s_factory.py`
  - **Component** Preprocessor / Postprocessor / Visualizer -> `processors/*.hpp` : `processors/*.py`
  - **Interface** Abstract contracts -> `i_factory.hpp` , `i_processor.hpp` : `i_factory.py` , `i_processor.py`
- Consolidated cross-language(Python : C++) common modules and 1:1 mapping structure
- Modernized `run_demo.sh` with a 3-stage interactive menu supporting variable AI tasks
- `--model` , `--image` , `--video` arguments are now optional — when omitted, task-appropriate default sample image/video is automatically selected
- `setup_sample_models.sh` migrated to Python-based downloader — supports `--list` , `--dry-run` , `--category` , `--models` and other granular download options
- `setup.sh` now integrates model download options — e.g. `setup.sh --models YoloV7 YoloV8S` to download specific models
- `run_demo.sh` fully redesigned — 18 demo models, 3-stage interactive menu (Task→Mode→Input), unified C++/Python support, `--task` / `--mode` / `--input` CLI arguments for non-interactive usage

© Copyright 2026 DEEPX All Rights Reserved.

- Display windows now show original resolution frames with auto-sized window (~1/4 screen area)

11.6.2 2. Fixed

- None

11.6.3 3. Added

- Supported new Depth Estimation task featuring FastDepth for monocular depth estimation
- Supported new Image Restoration task featuring DnCNN, Zero-DCE, and ESPCN models
- Migrated Full DX-Model Zoo encompassing 280 models across 17 task categories with 560 C++/Python examples (sync+async)
- Added yolov8, v9, v10, v11, v12 PPU models and C++/Python examples
- Implemented https://sdk.deepx.ai manifest-based DX-ModelZoo auto-download system (`scripts/download_models.py`)
- Auto-download for models and videos — automatically invokes `setup_sample_models.sh` when model file is missing, videos via `setup_sample_videos.sh`
- Interactive mode for `scripts/run_examples.sh` — 6-stage menu (Language→Category→Filter→ExecMode→InputType→Options) when run without arguments, with case-insensitive keyword filtering
- `dx_tool.sh run` unified with `run_examples.sh` interactive mode
- Real-time performance table output during example execution
- `--show-log` option for Python examples — controls per-frame detailed log output

11.7 DX-APP v3.0.2 / 2026-02-26

11.7.1 1. Changed

- Copy of dxrt and vcpkg DLLs into the dx-app/bin directory when building with MSVC.

11.7.2 2. Fixed

- Removed experimental filesystem includes and updated float literals in example cpp files for build error on windows
- Refactored apply_argmax to reduce nesting and fix gcovr warnings

© Copyright 2026 DEEPX All Rights Reserved.

11.7.3 3. Added

- Added vcpkg installation script for windows build.

11.8 DX-APP v3.0.1 / 2026-02-09

11.8.1 1. Changed

- None

11.8.2 2. Fixed

- Hardcoded attribute size in YOLO post-processing to dynamically adjust based on model output shape

11.8.3 3. Added

- Added yolov26 cls, yolo26 pose, yolo26 seg, yolo26 obb examples

---

11.9 DX-APP v3.0.0 / 2026-01-16

11.9.1 1. Changed

1.1 Major Project Structure Refactoring

- **Complete overhaul from existing demo applications to example system**
- To improve user understanding, separated the previously integrated example code by Task (classification, object detection, segmentation, face recognition, pose estimation) / Model (EfficientNet, YOLO, YOLO_PPU, SCRFD, ...) / Inference method (sync, async) / Post-processing (pure python, pybind)
  - Complete removal of legacy C++ demo code in `demos/` directory and provision of `run_demo.sh` and `run_demo.bat` based on separated examples
  - Transition to new `src/cpp_example/` and `src/python_example/` structure

1.2 Build System Improvements

- Improved CMake configuration and enhanced shared library support

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- Updated C++17 and Visual Studio 2022(v143) configuration for Windows build
- Adjusted DXRT include and link directories for cross-compilation

1.3 Complete Reconstruction of C++ / Python Example System

- **Support for synchronous and asynchronous execution modes**
- **Support for various input sources**: image, video, camera, RTSP stream
- **Real-time processing mode**: Performance measurement without GUI using `--no-display` option
- **Enhanced performance profiling**:
- Latency measurement for each stage: preprocessing, inference, post-processing
- E2E(End-to-End) FPS calculation and performance report generation
- Automatic generation of timestamp-based performance report files

1.4 Model Support Expansion

- **YOLOv10, YOLOv11, YOLOv12** examples added
- **YOLOv8 Segmentation** (YOLOv8-seg) support
- **DeepLabv3** segmentation model support
- **PPU (Post-Processing Unit)** module integration:
- YOLOv5, YOLOv7 PPU version support
- SCRFD PPU version support
- Both Python and C++ examples provided

1.5 Documentation Improvements

- Newly written example guides and installation guides
- Added detailed usage examples and parameter descriptions for each model

11.9.2 2. Fixed

2.1 Code Quality Improvements

- Added try-catch error handling to all projects
- Improved `std::exception` handling and throw `std::invalid_argument` when layer requirements are not met

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- Removed `using namespace std` usage and improved code clarity with explicit `std::` usage
- Improved parameter handling and frame processing logic
- Enhanced argument validation and error messages

2.2 Input Processing Improvements

- Set `cv2.CAP_PROP_BUFFERSIZE` (buffer size 1) for camera and RTSP speed improvement
- Fixed input_tensor passing to maintain memory reference until asynchronous inference completes

11.9.3 3. Added

3.1 Post-processing Library (dx_postprocess)

- **Pybind11-based Python binding**:
- Provides Python binding for C++ post-processing functions
- Automatically installs to current Python execution environment

3.2 Test Infrastructure Construction

- **Pytest-based integrated test system**:
- Automated testing for all Python examples
- Achieved code coverage of 93.65% or higher
- E2E(End-to-End) test framework
- Includes all model tests for classification, object detection, segmentation, and pose estimation
- Added `.coveragerc` file for code coverage configuration
- Support for display mode and E2E mode testing

3.3 New Examples and Features

- **Classification Models**:
- EfficientNet example integration
- ImageNet classification examples (synchronous/asynchronous)
- **Object Detection Models**:
- YOLO26 (Support new Ultralytics model, which is optimized for edge deployment)

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- YOLOv5, YOLOv7, YOLOv8, YOLOv9, YOLOv10(python only), YOLOv11(python only), YOLOv12(python only)
- YOLOX
- SCRFD (face detection)
- YOLOv5-Face
- YOLOv5-Pose (pose estimation)
- **Segmentation Models**:
- YOLOv8-seg (instance segmentation)
- DeepLabv3 (semantic segmentation)
- **Pose Estimation**:
- YOLOv5-Pose examples
- Added skeleton drawing functionality

11.9.4 4. Removed or Replaced

4.1 Legacy Demo Removal

- Complete removal of `demos/classification/`
- Complete removal of `demos/object_detection/`
- Complete removal of `demos/segmentation/`
- Complete removal of `demos/pose_estimation/`
- Complete removal of `demos/face_recognition/`
- Removal of `demos/denoiser/`
- Removal of `demos/dncnn_yolo/`
- Removal of `demos/object_det_and_seg/`
- Removal of `demos/noiseVideoMaker/`

4.2 Legacy Configuration File Removal

- Complete removal of JSON configuration files in `example/` directory
- Removal of `example/dx_postprocess/` JSON files
- Removal of Debian package related files (`debian/`)
- Removal of Docker build files (`docker/Dockerfile.app.build`)

© Copyright 2026 DEEPX All Rights Reserved.

4.3 Legacy Code Cleanup

- Removal of `demo_utils/` directory
- Removal of duplicate or unused code
- Removal of old YOLOv5 post-processing files
- Removal of RISCV64 architecture support

11.9.5 5. Migration Guide

5.1 Notice for Existing Users

v3.0.0 is a major update that includes **Breaking Changes** compared to v2.x.

- **Demo Code**: The existing `demos/` directory has been completely removed. Please refer to the new examples in `src/cpp_example/` and `src/python_example/` .
- **JSON Configuration Files**: JSON files in the existing `example/` directory have been removed. Python examples are configured directly through command-line arguments.
- **YOLO Post-processing Type Names**: Some have been changed, but aliases are provided for backward compatibility.

5.2 Recommended Upgrade Path

1. Refer to Python example documentation
2. Check example code in `src/python_example/` or `src/cpp_example/` directory
3. Install Python dependencies through `requirements.txt`
4. Use build scripts (`build.sh` or `build.bat`)

11.9.6 6. Known Issues

- When using the PPU model for face detection & pose estimation, `dx-compiler v2.1.0 and v2.2.0` does not currently support converting face and pose models to PPU format. This feature will be added in a future release. The PPU models used in the demo were converted using dx-compiler `v1.0.0(dx_com v1.60.1)` .

11.10 DX-APP v2.1.0 / 2025-11-28

ⓒ Copyright 2026 DEEPX All Rights Reserved.

11.10.1 1. Changed

- Enhanced build script documentation and usage instructions
- Updated cmake configuration in build.bat to use C++17 and v143 to enhance documentation windows build script(visual studio 2022)
- Model package updated from version 2.0.0 to 2.1.0 to support PPU models
- Improved demo script with additional PPU-Demo (1, 4, 6, 8, 11)
- Added CPU-specific PyTorch wheel source (https://download.pytorch.org/whl/cpu) in templates/python/requirements.txt.

11.10.2 2. Fixed

- Fixed Windows MSBuild compilation warnings by replacing implicit type casts with explicit static_cast
- Improved tensor allocation in imagenet classification example
- Updated numBoxes calculation based on post-processing type in LayerReorder
- Rename YOLO post-processing types and add aliasing for backward compatibility
- Added VSCode configuration files for usability
- Fixed errors that occurred when using VAAPI with camera input
- Enhanced yolo application to display final FPS even when forcefully terminated during camera input usage
- Enhanced user input handling for run_demo selection with a countdown timer (20s)

11.10.3 3. Added

- Windows Environment Support DX-APP now fully supports the Windows operating system! In response to user requests, we've expanded compatibility beyond Linux to include Windows, enabling a broader range of development environments to take advantage of DX-APP.
  - **OS**: Windows 10 / 11
  - **DEEPX M1 Driver Version**: v1.7.1 or higher
  - **DEEPX M1 Runtime Lib Version**: v3.1.0 or higher
  - **Python**: Version 3.8 or higher (required for Python module support)
  - **Compiler**: Visual Studio Community 2022 (required for building C++ examples)
- Added automated build script (build.bat) for automatic build and Visual Studio solution generation

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- Three new PPU data types : BBOX (for object detection) / POSE (for pose estimation keypoints) / FACE (for face detection landmarks)
- Enhanced post-processing functions to support PPU inference output format

11.10.4 4. Known Issues

- DeepLabV3 Semantic Segmentation model accuracy may be slightly degraded in dx-compiler(dx_com) v2.1.0. This will be fixed in the next release. The DeepLabV3 model used in the demo was converted using dx-compiler v2.0.0.
- When using the PPU model for face detection & pose estimation, dx-compiler v2.1.0 does not currently support converting face and pose models to PPU format. This feature will be added in a future release. The PPU models used in the demo were converted using dx-compiler v1.0.0(dx_com v1.60.1).

11.11 DX-APP v2.0.0 / 2025-09-08

11.11.1 1. Changed

- Moved the YOLO post-processing guide from 07_Python_Examples.md to a new, dedicated document 08_YOLO_Post_Processing_Pybind11.md.
- Refactored yolo_pybind_example.py to use a RunAsync() + Wait() structure instead of callbacks. This ensures the output tensor order is correctly handled.
- Major code refactoring and restructuring of demo applications
- Consolidated common utilities into directory
- Removed deprecated and legacy codes
- Updated documentation and resources
- YoloPostProcess now filters and selects the correct tensor by output_name when USE_ORT=ON
- Command-line help messages in various demos have been improved to clearly mark required parameters.
- Replaced YOLOv5s-1 example json to YOLOv5s-6 json configuration file has been added for object detection.
- Documentation has been updated to add Python requirements and modified some images.
- feat: add OS and architecture checks in build script & update CPU specifications in documentation

ⓒ Copyright 2026 DEEPX All Rights Reserved.

11.11.2 2. Fixed

- FPS calculation bug in yolo_multi
- Removed postprocessing code for legacy PPU models
- Fixed postprocessing logic to support new output shapes of YOLO models when USE_ORT=OFF
- Fixed typo error in framebuffer info file path (yolo_multi app)
- Improved error messages for output tensor size mismatch and missing in Yolo post processing
- Rename output tensors in json config 'yolov5s6_example.json'

11.11.3 3. Added

- Added to cleanly purge the pip-installed package and local build artifacts (shared library, dist-info/egg-info, and build directory).
- Added version guards in templates/python/yolo_pybind_example.py to ensure compatibility with DX-RT ≥ 3.0.0 and DXNN model version ≥ 7
- Enhanced the JSON configuration to support a target_output_tensor_name key and a name field for each layer parameter.
- Added a feature to filter output tensor using the target_output_tensor_name provided in the JSON configuration
- Added a feature to automatically reorder model layer parameters in the JSON configuration to match the model's actual output tensor sequence.
- Enhanced demo applications
- Added for easy demo execution
- Added SCRFD decoding method for run_detector example
- Added postprocessing support for yolo_pose and yolo_face models (available only when USE_ORT=ON)
- Listed supported YOLO model types for YoloPostProcess in README
- feat: add uninstall script and enhance color utility functions

11.12 DX-APP v1.11.0 / 2025-07-23

11.12.1 1. Changed

- feat: enhance --clean option in build script for pybind artifacts
- feat: update dxnn models version(1.40.2 to 1.60.1)

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- feat: auto run setup script or display a guide message when a file not found error occurs during example execution

11.12.2 2. Fixed

- feat: Improve error message readability in install, build scripts
- Apply color to error messages
- Reorder message output to display errors before help messages
- Updated tensor index assignment in Yolo layer reordering
- fix: resolve dx_postprocess Python lib build error and improve error handling

11.13 DX-APP v1.10.0 / 2025-06-17

- Initial create dx-app
- demo : classification
- demo : object detection
- demo : pose estimation
- demo : multi models for object detection and segmentation
- demo : semantic segmentation
- demo : multi channel object detection
- template : classification
- template : object detection
- template : python example (sync/async/pybind C++)

ⓒ Copyright 2026 DEEPX All Rights Reserved.