

DXNN Application (DX-APP) User Manual

2026/07/28 (v3.2.1)

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 Installation and Build 10
2.1 Overview & System Requirements 10
2.2 Installation on Linux 10
2.3 Installation on Windows 15
3. DX-APP C++ Example 24
3.1 DX-APP C++ Usage Guide 24
3.2 DX-APP C++ Example Tests 32
4. DX-APP Python Example 40
4.1 DX-APP Python Usage Guide 40
4.2 DX-APP Python Example Tests 49
5. DX-APP Post-processing 54
5.1 DX-APP C++ Post-processing 54
5.2 DX-APP Python Post-processing 58
6. Project README 63
6.1 Overview 63
6.2 Quick Start Guide 63
6.3 Prerequisites 66
6.4 Core Concepts & Architecture 68
6.5 Usage Reference 70
6.6 Advanced Features 74
7. DX-APP Developer Guide 76
7.1 DX-APP DX Tool Guide 76
7.2 DX-APP Example Source Structure 88
7.3 DX-APP YOLO Customizing Guide 103
7.4 DX-APP Agent-Driven Development Guide 112
8. Appendix 131
8.1 Third-Party License Notice 131
9. RELEASE_NOTES 133
9.1 DX-APP v3.2.1 / 2026-07-28 133
9.2 DX-APP v3.2.0 / 2026-06-25 133
9.3 DX-APP v3.1.1 / 2026-04-21 135

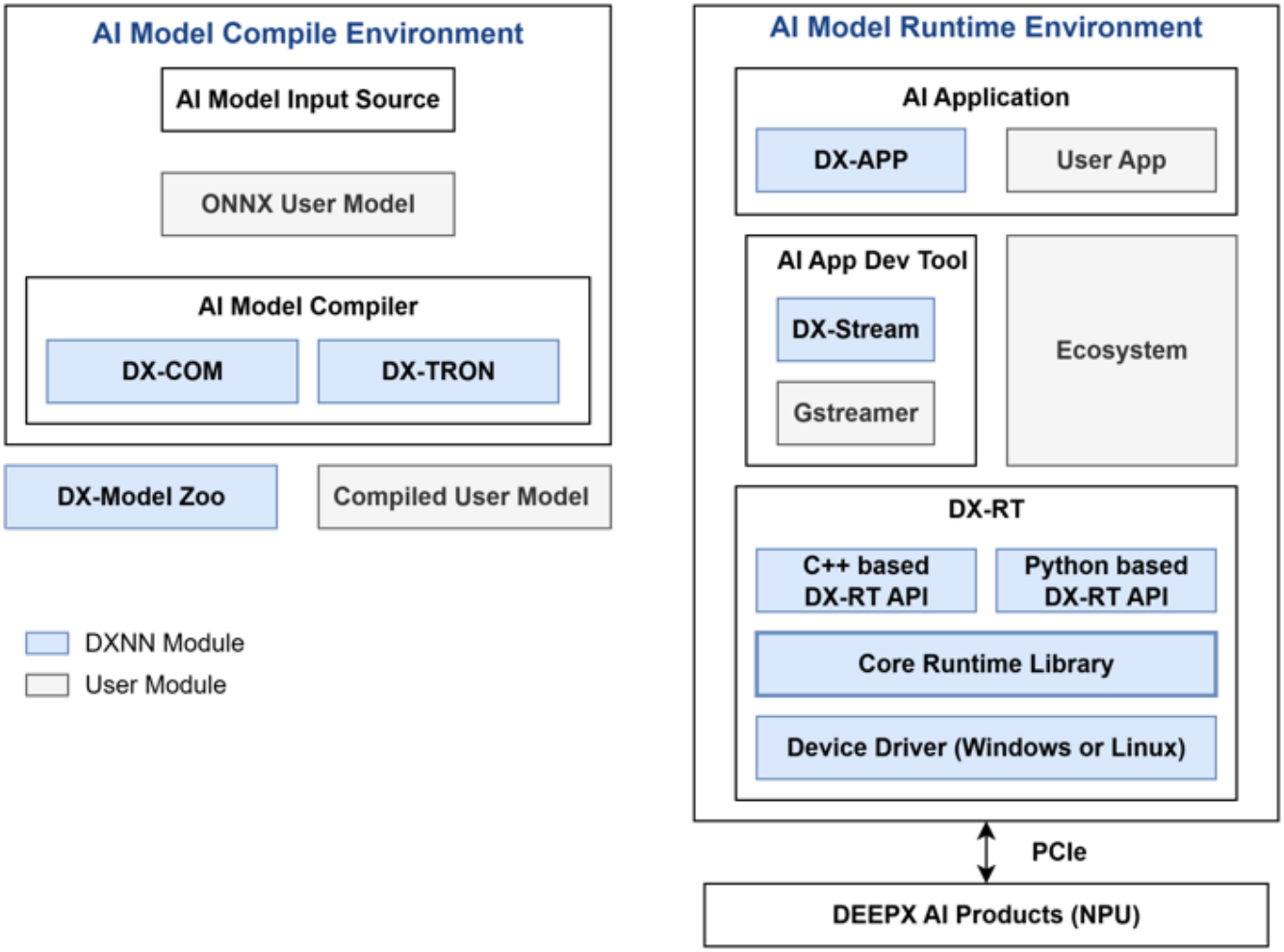
ⓒ Copyright 2026 DEEPX All Rights Reserved.

9.4 DX-APP v3.1.0 / 2026-04-06 136
9.5 DX-APP v3.0.2 / 2026-02-10 137
9.6 DX-APP v3.0.1 / 2026-02-05 137
9.7 DX-APP v3.0.0 / 2026-01-02 138
9.8 DX-APP v2.1.0 / 2025-11-28 142
9.9 DX-APP v2.0.0 / 2025-08-14 143
9.10 DX-APP v1.11.0 / 2025-07-24 145
9.11 DX-APP v1.10.0 / 2025-06-17 145

ⓒ 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

© Copyright 2026 DEEPX All Rights Reserved.

compilation parameters. DX-COM provides a fully compiled .dxnn file, optimized for low-latency and high-efficient inference on DEEPX NPU.

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

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

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

DX-APP is a collection of runnable example applications that execute compiled models on DEEPX NPUs through DX-RT. It provides examples across 22 AI task categories including classification, object detection, face detection, pose estimation, semantic/instance segmentation, depth estimation, OBB detection, embedding, image denoising, enhancement, super resolution, hand landmark detection, attribute recognition, person re-identification, face alignment, keypoint detection, object pose estimation, panoptic driving perception, 3D object detection, and hand detection. The current DX-APP repository is organized around refactored `src/cpp_example/` and `src/python_example/` trees, each with their 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. DX-APP is intended both as a quick-start runtime package for users and as a reusable application baseline for further customization. Below are representative run examples.

1.2 DX-APP Features

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

ⓒ Copyright 2026 DEEPX All Rights Reserved.

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 by a refactored 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**
>
> For installation and build steps, refer to DX-APP Installation and Build. 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.

Classification (EfficientNet-Lite0)

- Input: image (e.g., `224x224`)
- Output: Top-1 class
- Example run

```
./bin/efficientnet_lite0_async -m ./assets/models/efficientnet-lite0_256x256.dxnn -
i ./sample/ILSVRC2012/0.jpeg -l 1
./bin/efficientnet_lite0_sync  -m ./assets/models/efficientnet-lite0_256x256.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/yolov8n_async -m ./assets/models/yolov8-n_640x640.dxnn -i ./sample/img/
sample_kitchen.jpg --no-display -l 1
./bin/yolov8n_sync  -m ./assets/models/yolov8-n_640x640.dxnn -i ./sample/img/
sample_kitchen.jpg --no-display -l 1 -s
```

Face Detection (SCRFD)

- Input: image
- Output: face boxes, landmarks, log
- Example run

```
./bin/scrfd500m_async -m ./assets/models/scrfd-500m_640x640.dxnn -i ./sample/img/
sample_face.jpg --no-display -l 1
./bin/scrfd500m_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
- Output: person boxes + keypoints
- Example run

```
./bin/yolov8s_pose_async -m ./assets/models/yolov8-s-pose_640x640.dxnn -i ./sample/
img/sample_kitchen.jpg --no-display -l 1
./bin/yolov8s_pose_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.

Segmentation (DeepLabV3+)

- Input: image/video/camera
- Output: boxes + masks rendered, results saved
- Example run

```
./bin/deeplabv3plusmobilenet_async -m ./assets/models/
deeplabv3plus_mobilenetv1_512x512.dxnn -i ./sample/img/sample_parking.jpg --no-
display -l 1
./bin/deeplabv3plusmobilenet_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
- Output: segmentation masks rendered
- Example run

```
./bin/bisenetv1_async -m ./assets/models/bisenetv1_1024x2048.dxnn -i ./sample/img/
sample_parking.jpg --no-display -l 1
./bin/bisenetv1_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 Unified Post-Processing Engine

To ensure consistency and speed, all model-specific decoding (NMS, box scaling, mask generation) is implemented in optimized C++ libraries.

- **Cross-Language Parity:** These modules are exposed to Python via `pybind11` (`dx_postprocess`), ensuring Python developers achieve C++-level performance.
- **Logic Standardization:** Identical decoding logic across both environments guarantees consistent inference results.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

1.3.2 Execution Paradigms: Sync vs. Async

Templates 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 application template 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
  : **Pre-processing:** Image decoding, 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 Installation and Build

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

2.1 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.

2.2 Installation on Linux

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

2.2.1 Software Requirements on Linux

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

- **OS**: Ubuntu 18.04 / 20.04 / 22.04 / 24.04 (x64) and Debian 12 / 13 (x64)
- **Deepx M1 Runtime Lib Version**: v3.0.0 or higher

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

ⓒ Copyright 2026 DEEPX All Rights Reserved.

2.2.2 Prerequisites Setup

Step 1. Install DX-RT Device Driver

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 install the DX-RT library and NPU device driver, 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`.

2.2.3 DX-APP Application Setup

Step 1. DX-APP Installation Options

You can check the available **DX-APP** installation options by running the following command.

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

You can view more installation options by entering the `--help` flag.

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 are 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 and Run DX-APP

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.

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

> **NOTE**
>
> Running `setup.sh` beforehand is optional. When you run any individual example, missing models are **automatically downloaded** on demand. Videos are also auto-downloaded when a `--video` path is specified but the file does not exist.

> **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
`--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`	Force remove models if they exist
`--force-remove-videos`	Force remove videos if they exist
`--verbose`	Show detailed progress output

In internal-network environments, the setup flow can use the internal DX-ModelZoo source automatically when the intranet mode is enabled by the surrounding environment.

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

Internal-Network Setup

In internal environments, DX-APP can use the internal DX-ModelZoo source to prepare model assets without requiring manual model-by-model input during the standard setup flow.

Post-Processing Unit (PPU) Acceleration Integration

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

© Copyright 2026 DEEPX All Rights Reserved.

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 includes 11 PPU-accelerated model variants across multiple tasks. To run PPU models interactively, use `./run_demo.sh` and select "PPU Pipeline", or use the example runner / DX Model Tool:

```
# Interactive — select PPU from the category menu
scripts/run_examples.sh
./scripts/dx_tool.sh run

# Non-interactive
./scripts/dx_tool.sh run --lang cpp --category ppu
```

Available PPU models: YOLOv5S, YOLOv7, YOLOv7x, YOLOv8N, YOLOv8S, YOLOv9T, YOLOv10N, YOLOv11N, YOLOv12N, SCRFD500M, YOLOv5Pose.

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
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

2.3 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

2.3.1 Software Requirements on Windows

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

- **OS**: Windows 10 or later
- **Python**: Version 3.8 or higher (required for Python module support)
- **Compiler**: Visual Studio Community 2022 (required for building C++ examples)

2.3.2 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.

2.3.3 Install Visual Studio Community 2022

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

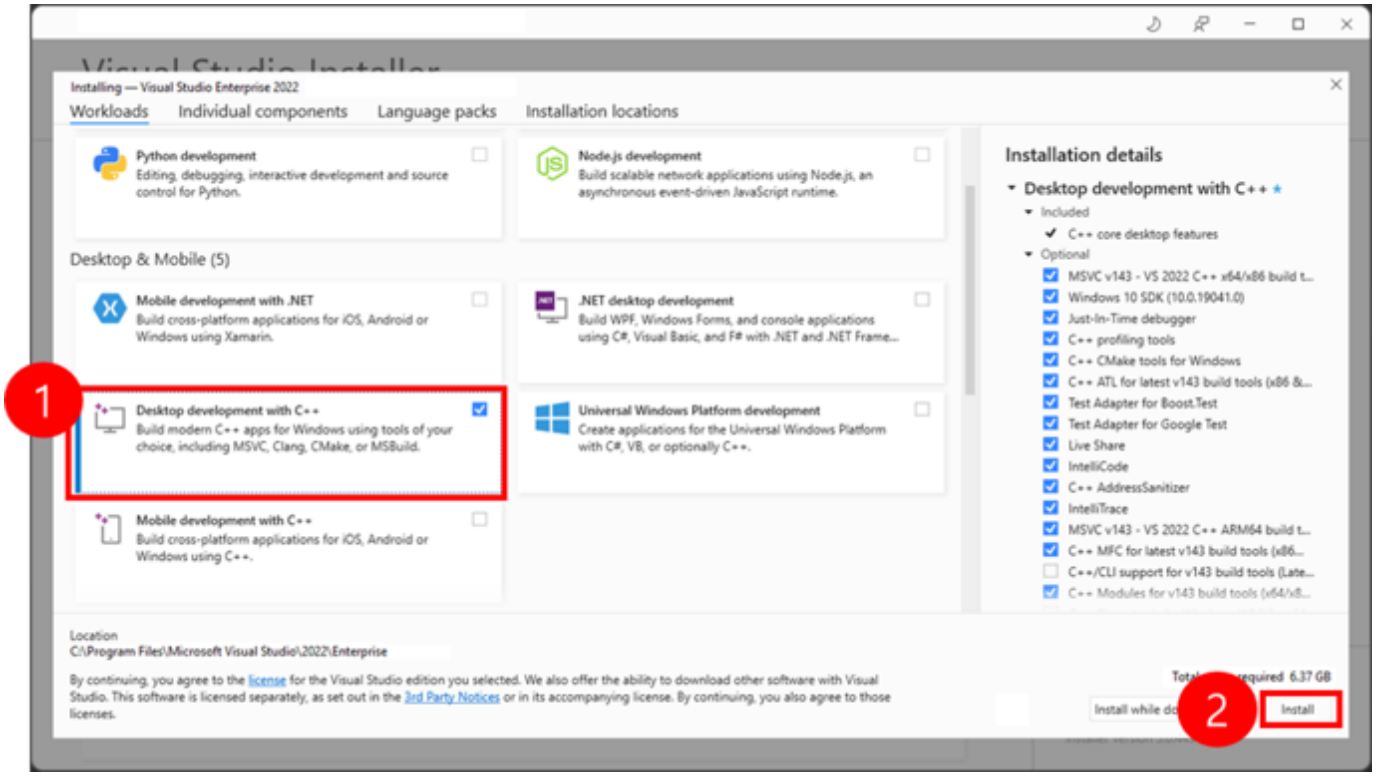
ⓒ Copyright 2026 DEEPX All Rights Reserved.

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



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

2.3.4 Install VCPKG

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

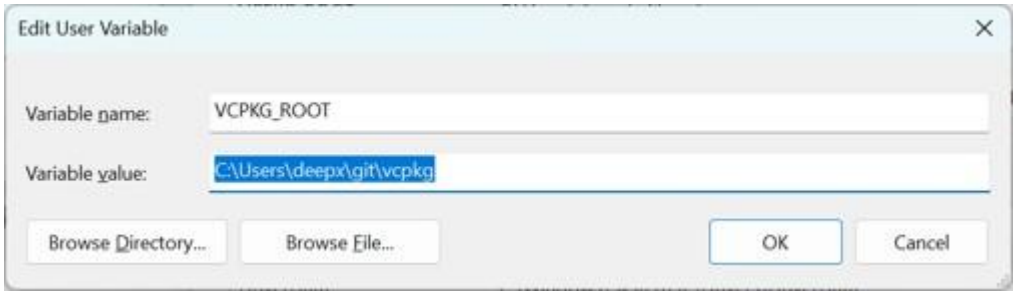
© Copyright 2026 DEEPX All Rights Reserved.

> **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

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



2.3.5 Build and Install dx_app in Visual Studio Community 2022

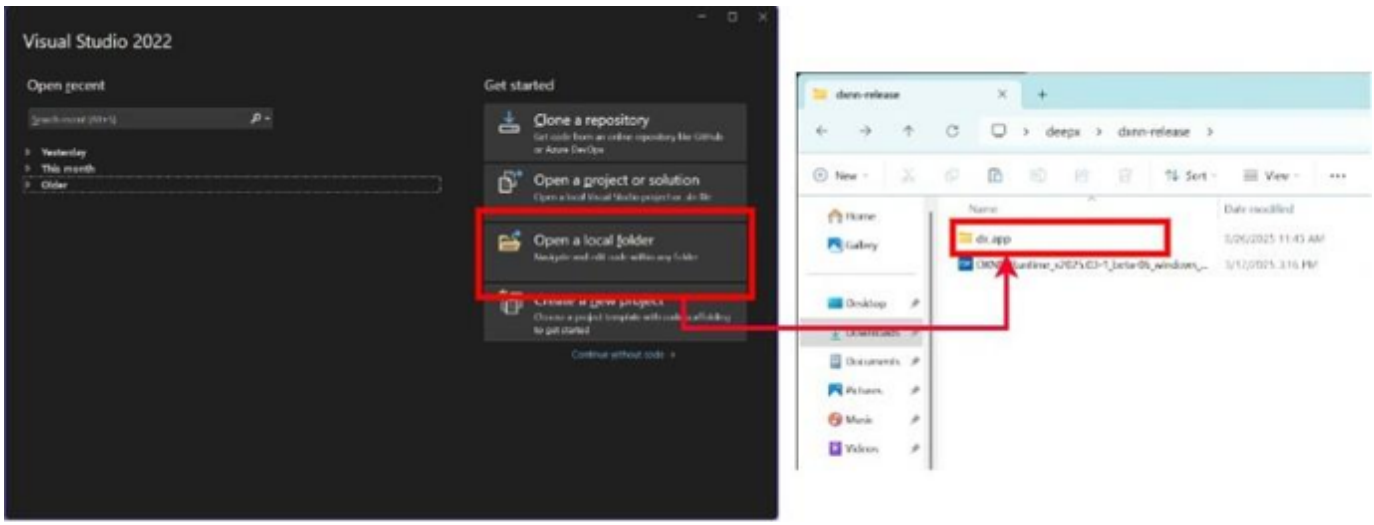
To build and run the `dx_app` application on Windows, follow the steps below using Visual Studio Community 2022.

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

© Copyright 2026 DEEPX All Rights Reserved.

> **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.

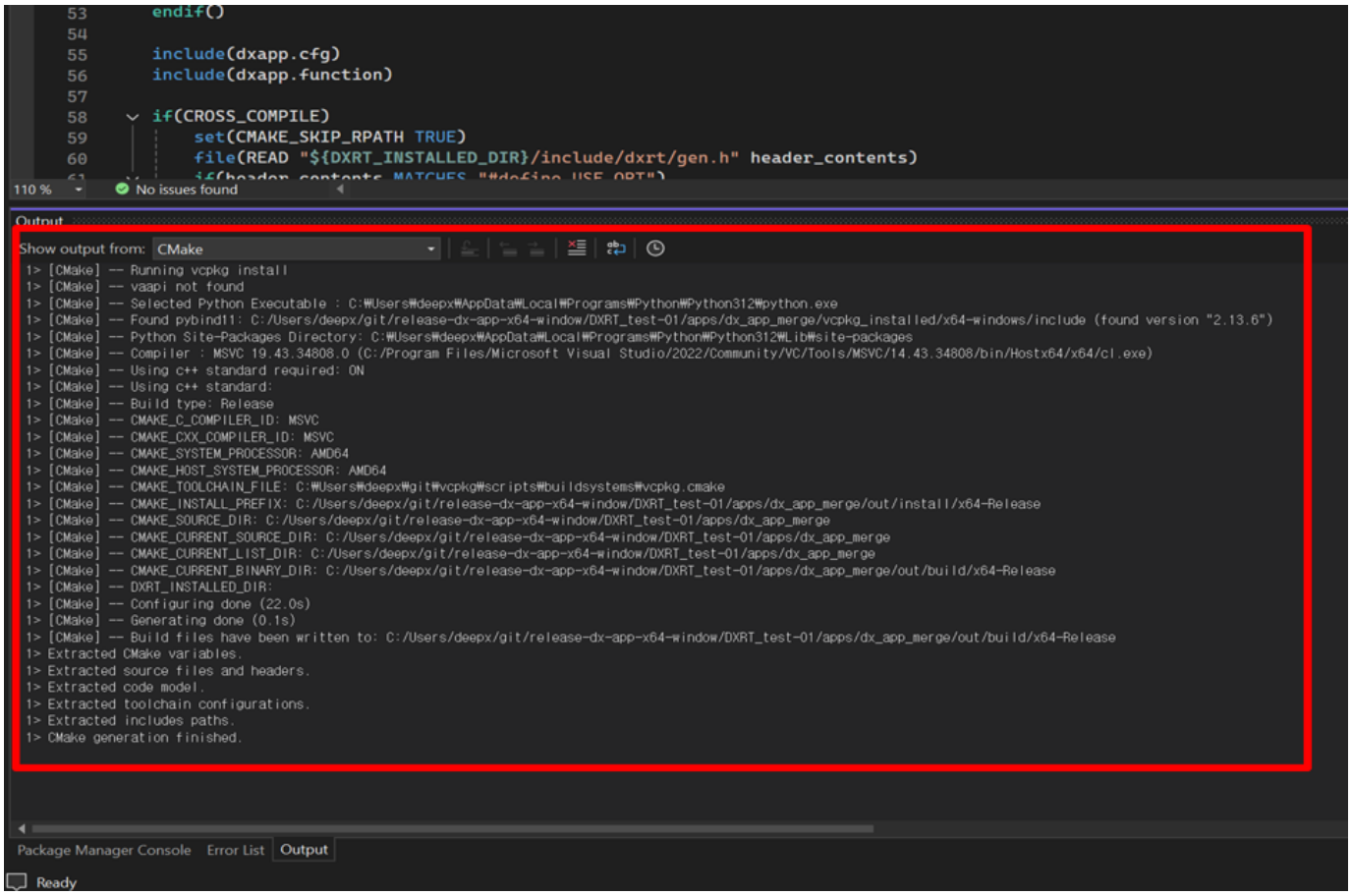


Step 2. Project Configuration

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`.

- `DXRT_DIR` : Path to the installed 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"
}
```

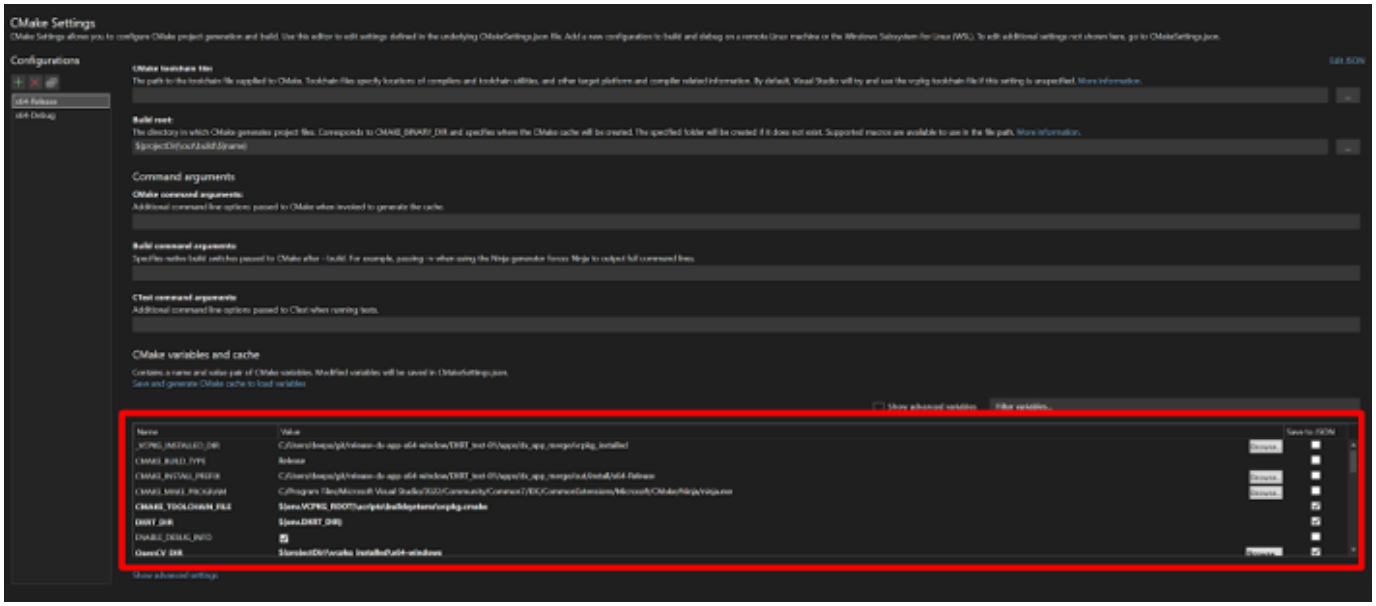

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Step 4. Register `PATH` Variable

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

- `DXRT_DIR` is referenced in CMAKE as `{env.DXRT_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.



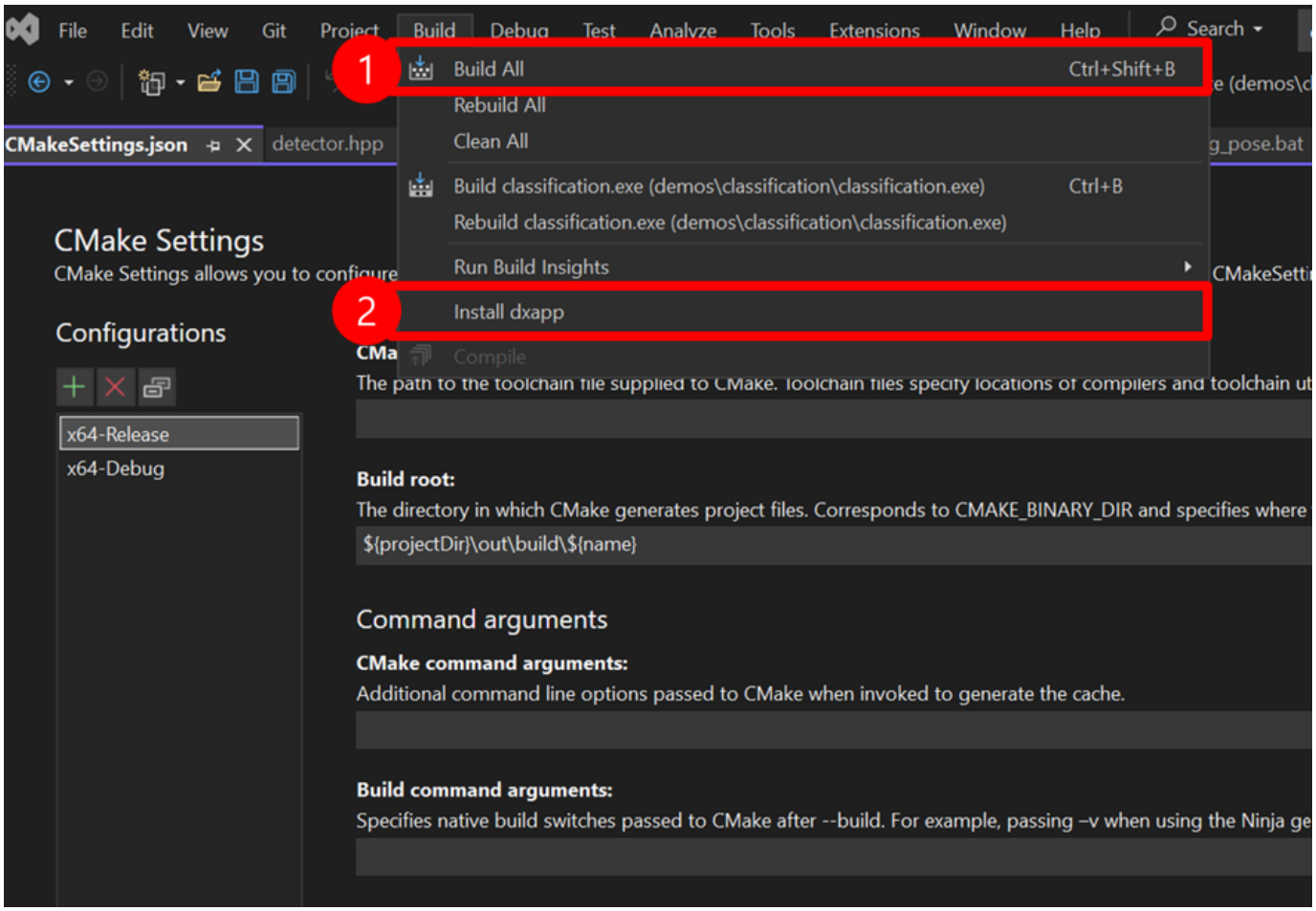
Step 5. Build and Install `dx_app`

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.



2.3.6 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., `DXRT_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

- `DXRT_DIR` set to the DX-RT installation directory
- Visual Studio 2022 with Desktop development with C++ workload
- CMake available in `PATH`

Usage

© Copyright 2026 DEEPX All Rights Reserved.

From the project root

```
build.bat
```

`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 scripts 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/                    # Installed executables
└── lib/                    # Installed libraries
```

2.3.7 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 automatically download all required models and sample videos.

- The downloaded assets will be placed in the `assets` folder.
- The assets include models for Classification, Object Detection, and Segmentation.

Step 2. Run Examples

ⓒ Copyright 2026 DEEPX All Rights Reserved.

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

classification example

```
./bin/efficientnet_lite0_async.exe -m ./assets/models/efficientnet-
lite0_256x256.dxnn -i ./sample/ILSVRC2012/0.jpeg
```

object detection example

```
./bin/yolov8n_sync.exe  -m ./assets/models/yolov8-n_640x640.dxnn -i ./sample/img/
sample_kitchen.jpg -l 10
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

3. DX-APP C++ Example

3.1 DX-APP C++ Usage Guide

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

3.1.1 Overview

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

- **task**
- **model family**
- **execution variant**

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 7-module factory-based architecture. Each model 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:

- `classification/`
- `object_detection/`
- `face_detection/`
- `pose_estimation/`
- `semantic_segmentation/`
- `instance_segmentation/`
- `depth_estimation/`
- `embedding/`
- `image_denoising/`
- `image_enhancement/`
- `super_resolution/`
- `hand_landmark/`
- `obb_detection/`
- `ppu/`
- `attribute_recognition/`
- `reid/`
- `face_alignment/`

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

3.1.2 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 interfaces (.hpp)	`IFactory`, `IProcessor`, `IVisualizer`, `IInputSource`
`common/config/`	`model_config.hpp`	

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Module	Contents	Role
		Loads `config.json` (input size, labels, thresholds)
`common/processors/`	44 processors	Shared decode logic for all model families
`common/runner/`	24 runner headers	12 sync + 12 async task-specific runner pairs
`common/inputs/`	5 source headers	Image, Video, Camera, RTSP input abstraction
`common/visualizers/`	12 visualizers	Task-specific result rendering
`common/utility/`	8 utility headers	Labels, preprocessing, profiling, run_dir, signal_handler, 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 model directory has a `factory/{model}_factory.hpp` that implements `IFactory`:

```
// factory/yolov9s_factory.hpp
#include "common/processors/yolov8_postprocessor.hpp"
#include "common/visualizers/detection_visualizer.hpp"

class YOLOv9sFactory : public IFactory {
    IProcessor* createProcessor() override { return new
YOLOv8Postprocessor(config); }
    IVisualizer* createVisualizer() override { return new DetectionVisualizer(); }
};
```

The entry-point delegates to a task-specific runner:

```
// yolov9s_sync.cpp
auto factory = YOLOv9sFactory(config);
SyncDetectionRunner runner(factory);
runner.run();
```


© Copyright 2026 DEEPX All Rights Reserved.

Directory Pattern & File Pattern

Each model family usually has its own directory.

Example:

```
src/cpp_example/object_detection/yolov9s/
├── config.json
├── factory/
│   └── yolov9s_factory.hpp    # Assembles processor + visualizer from common/
├── yolov9s_sync.cpp             # Entry point → sync_detection_runner
└── yolov9s_async.cpp            # Entry point → async_detection_runner
```

Common files:

- `config.json` : model-specific runtime settings
- `factory/` : factory header wiring shared `common/` components
- `*_sync.cpp` : synchronous execution example (via task-specific sync runner)
- `*_async.cpp` : asynchronous execution example (via task-specific async runner)

3.1.3 Execution Framework

Execution Variants

Synchronous Flow (`*_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 (`*_async.cpp`)

Use this variant when you want:

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

ⓒ Copyright 2026 DEEPX All Rights Reserved.

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 (auto-downloaded if 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
`--show-log`	—	bool	Enable verbose log output (default: quiet)
`--config`	—	string	Model config JSON path (auto-detected if omitted)
`--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.

© Copyright 2026 DEEPX All Rights Reserved.

> **NOTE**
>
> **Image-only tasks:** `embedding`, `reid`, and `attribute_recognition` tasks accept `--image_path` input only. `--video_path`, `--camera_index`, and `--rtsp_url` are not supported for these tasks because meaningful inference requires a crop of a pre-detected subject (face or person). Running a single embedding model on a raw video stream without a preceding detector would not produce valid results.

3.1.4 Getting Started (Workflow)

Step 1. Prepare assets

```
./setup.sh
```

Step 2. Build the repository

```
./build.sh
```

Step 3. Run a C++ example

```
./bin/yolov9s_sync -m assets/models/yolov9-s_640x640.dxnn -i sample/img/
sample_kitchen.jpg
./bin/yolov9s_async -m assets/models/yolov9-s_640x640.dxnn -v assets/videos/dance-
group.mov
```

3.1.5 Advanced Operations & Debugging

Runtime Features

Auto-Download

When a specified model file is not found locally, the runner automatically attempts to download it via `setup_sample_models.sh`. If a `--video` file is missing, `setup_sample_videos.sh` is invoked. If the download fails, a clear error message with manual download instructions is displayed.

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.

Signal Handling

All runners install SIGINT/SIGTERM handlers (`installSignalHandlers()`). Pressing Ctrl+C triggers a graceful shutdown with clean resource release.

Output Management (`--save`)

When `--save` is enabled, a timestamped directory is created (e.g., `artifacts/cpp_example/{model}-image-{name}-{timestamp}/`) containing `run_info.txt` , saved images/video, and optional tensor dumps.

Configuration Management (`--config`)

Runtime parameters (thresholds, top-k, etc.) can be customized per-model via `config.json` . If omitted, auto-detected adjacent to the model file.

Verification & Diagnostics

Numerical Verification (`DXAPP_VERIFY`)

Set `DXAPP_VERIFY=1` to serialize all post-processing results to `logs/verify/{model}.json` for inspection and debugging.

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

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

Environment Variables Reference

Variable	Description
`DXAPP_SAVE_IMAGE`	Save visualization to the specified file path
`DXAPP_VERIFY`	When `1` , dump JSON verification data

ⓒ Copyright 2026 DEEPX All Rights Reserved.

3.1.6 Supplementary Information

Component Relationships

`src/postprocess/` contains C++ post-processing implementations that are **not** used by `cpp_example/` `common/processors/` directly. Instead, 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.

3.2 DX-APP C++ Example Tests

3.2.1 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 meet performance benchmarks on the DEEPX NPU.

3.2.2 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` options, `--version` flag, 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: 100 seconds for image inference (300s for TTA models), 10 minutes for video inference.

Specialized Tests

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`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

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` JSON output validation
`test_multi_loop.py`	`multi_loop`	`-l N` loop count behavior
`test_signal_handling.py`	`signal_handling`	SIGINT graceful shutdown

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` from the project root if executables are missing.

For E2E Tests (Additional)

- **Models**: Run `./setup_sample_models.sh` to populate `assets/models/`. The current setup flow uses the DX-ModelZoo downloader path and can prepare models non-interactively for internal-network environments.
- **Test Data**: Ensure images exist in `sample/img/` and videos in `assets/videos/` (Run `./setup_sample_videos.sh`).
- **Libraries**: Shared libraries must be present in the `lib/` directory.

Available Markers (`pytest.ini`)

`cli`, `help`, `e2e`, `visualization`, `async_exec`, `sync_exec`, `save_mode`, `dump_tensors`, `verify`, `multi_loop`, `signal_handling`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

3.2.3 Test Infrastructure

Shared Module (`tests/common/`)

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

- `constants.py`: `TASK_IMAGE_MAP`, `MODEL_IMAGE_OVERRIDE`, `MULTI_MODEL_EXECUTABLES`, path constants
- `utils.py`: `setup_environment()`, `discover_cpp_executables()`, `normalize_model_name()`

Asset Management

Assets required for testing are automatically managed by scripts within the `scripts/` directory:

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

3.2.4 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 C++ tests with code coverage (standalone, for SonarQube)
./run_tc.sh --cpp --coverage
```


© Copyright 2026 DEEPX All Rights Reserved.

```
# 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 SonarQube 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

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

# Run all tests (excluding slow E2E tests)
pytest -m "not e2e"

# Run only CLI tests (fast)
pytest test_cli_help.py test_cli_basic.py

# 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_async"     # Only yolov7_async

# 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
```


© Copyright 2026 DEEPX All Rights Reserved.

```
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_async]
pytest test_e2e.py::test_image_inference_e2e[scrfd_async]

# Multiple specific tests
pytest test_e2e.py::test_stream_inference_e2e[yolov7_async] \
           test_e2e.py::test_stream_inference_e2e[yolov8_async]
```

Result Analysis

Performance Reporting

After running E2E 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 `performance_report_YYYYMMDD_HHMMSS.csv` in `tests/cpp_example/`.

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:** The slowest stage in the pipeline is automatically marked with an **asterisk (*)** for quick optimization targeting.

Test Coverage Summary

Example output: `./run_tc.sh --cpp --coverage`

Category	Count	Status
CLI Tests	~1,293	All binaries validated (help + basic)
E2E Image Tests	~242	sync + async, auto-discovered
Visualization Tests	~247	sync + async image verification
Feature Tests	~22	save_mode, dump_tensors, verify, multi_loop, signal_handling

© Copyright 2026 DEEPX All Rights Reserved.

3.2.5 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).

```
# 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
./build.sh --clean --coverage --type debug

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

# Verify coverage build (check for .gcno files)
ls build_x86_64/src/examples/*.gcno
```

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)

```
# [Recommended] Build with debug mode for most reliable coverage
cd ../../  # Go to project root
./build.sh --clean --coverage --type debug

# [Alternative] Build with relwithdebinfo for faster turnaround
./build.sh --clean --coverage --type relwithdebinfo
```

Report Generation

Trigger coverage analysis using the following commands

ⓒ Copyright 2026 DEEPX All Rights Reserved.

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 && ./run_tc.sh --cpp --coverage

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

Report Interpretation (HTML Report & Filtering)

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

- **Console summary:** Line and branch coverage percentages.
- **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.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Coverage Filtering Rules

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

3.2.6 Maintenance & CI/CD

Continuous Integration (CI) Integration

SonarQube Integration

XML reports generated via the `--coverage` flag are utilized by the CI server for static code analysis and test coverage tracking. This ensures long-term code quality and helps identify untested logic paths during the development lifecycle.

Nightly Build

A full suite of E2E tests is executed every night to perform regression testing. This process ensures that the application remains stable and functional despite frequent updates to NPU drivers, firmware, and the underlying runtime environment.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

4. DX-APP Python Example

4.1 DX-APP Python Usage Guide

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

4.1.1 Overview

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

- **task**
- **model family**
- **execution and post-processing variant**

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 7-module factory-based architecture. Each model directory contains only thin entry-point scripts and a factory that wires shared components together.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Representative task directories include:

- `classification/` — EfficientNet, AlexNet, ResNet, MobileNet, etc.
- `object_detection/` — YOLOv3/v5/v7/v8/v9/v10/v11/v12, YOLO26, YOLOX, NanoDet, DAMOYOLO, SSD
- `face_detection/` — SCRFD, YOLOv5Face, YOLOv7Face, RetinaFace
- `pose_estimation/` — YOLOv8-Pose
- `semantic_segmentation/` — BiSeNet, DeepLabV3+, SegFormer
- `instance_segmentation/` — YOLOv8Seg, YOLOv26Seg
- `depth_estimation/` — FastDepth, SCDepthV3
- `embedding/` — ArcFace
- `obb_detection/` — YOLOv26OBB
- `image_denoising/`, `image_enhancement/`, `super_resolution/`
- `hand_landmark/` — Hand landmark estimation
- `ppu/` — PPU-accelerated variants (YOLOv5/v7/v8/v9/v10/v11/v12/SCRFD/Pose)
- `attribute_recognition/` — Attribute recognition (DeepMAR)
- `reid/` — Person re-identification (CasViT)
- `face_alignment/` — Face alignment / 3D landmark (3DDFA v2)

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 Python examples:

Module	Role
`common/base/`	Abstract interfaces: `IFactory`, `IProcessor`, `IVisualizer`, `IInputSource`
`common/config/`	`ModelConfig` — loads `config.json` (input size, labels, thresholds)
`common/processors/`	35 shared post-processors covering all model families

© Copyright 2026 DEEPX All Rights Reserved.

Module	Role
`common/runner/`	`SyncRunner`, `AsyncRunner`, `run_dir`, `verify_serialize`, `args` — generic execution engines with built-in profiling
`common/inputs/`	Input source abstraction: image, video, camera, RTSP
`common/visualizers/`	10 task-specific visualizers (detection, segmentation, pose, etc.)
`common/utility/`	Labels, preprocessing, profiling, drawing helpers

Factory Pattern & Model Registry

Each model directory has a `factory/{model}_factory.py` that implements `IFactory`:

```
from common.processors import YOLOv5Postprocessor
from common.visualizers import DetectionVisualizer

class Yolov9sFactory(IFactory):
    def create_processor(self):
        return YOLOv5Postprocessor(self.config)
    def create_visualizer(self):
        return DetectionVisualizer(self.config)
```

The entry-point script simply delegates to the runner:

```
from common.runner import SyncRunner
runner = SyncRunner(factory)
runner.run()
```

Model Registry (`config/model_registry.json`)

A JSON registry stores per-model metadata (task, postprocessor type, input dimensions, thresholds). The `scripts/add_model.sh` tool reads this registry to auto-generate factory files, config.json, and all entry-point scripts — enabling zero-code model onboarding.

Directory & File Pattern

Each model family has its own directory with a consistent structure:

```
src/python_example/object_detection/yolov9s/
├── config.json                          # Model-specific runtime settings
├── factory/
```


© Copyright 2026 DEEPX All Rights Reserved.

```
│   └── yolov9s_factory.py                # Wires shared processor + visualizer
├── yolov9s_sync.py                        # Pure Python synchronous
├── yolov9s_async.py                       # Pure Python asynchronous
├── yolov9s_sync_cpp_postprocess.py        # Synchronous + C++ binding
└── yolov9s_async_cpp_postprocess.py       # Asynchronous + C++ binding
```

4.1.3 Execution Framework

Execution Variants

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

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`)
- better alignment with shared C++ decode logic
- 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 (auto-downloaded if missing)
`--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)
`--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)
`--output`	`-o`	string	Output file path (restoration/depth/SR only)
`--help`	`-h`	—	Show usage

- **Input source:** `--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.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

> **NOTE**
>
> **Image-only tasks:** `embedding`, `reid`, and `attribute_recognition` tasks accept `--image` input only. `--video`, `--camera`, and `--rtsp` are not supported for these tasks because meaningful inference requires a crop of a pre-detected subject (face or person). Running a single embedding model on a raw video stream without a preceding detector would not produce valid results. Passing a video/camera source to these tasks exits with an error.

4.1.4 Getting Started (Workflow)

Step 1. Prepare assets

```
./setup.sh
```

Step 2. Build shared libraries

```
./build.sh
```

Step 3. Run a Python example

```
python src/python_example/object_detection/yolov9s/yolov9s_sync.py --model assets/
models/yolov9-s_640x640.dxnn --image sample/img/sample_kitchen.jpg
python src/python_example/object_detection/yolov9s/yolov9s_async_cpp_postprocess.py -
-model assets/models/yolov9-s_640x640.dxnn --video assets/videos/dance-group.mov
```

4.1.5 Advanced Operations & Debugging

Runtime Features

Auto-Download

When a specified model file is not found locally, the runner automatically attempts to download it via `setup_sample_models.sh`. If a `--video` file is missing, `setup_sample_videos.sh` is invoked. If the download fails, a clear error message with manual download instructions is displayed.

Default Input Fallback

© Copyright 2026 DEEPX All Rights Reserved.

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:

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

Signal Handling

Both `SyncRunner` and `AsyncRunner` use `stop_event` (`threading.Event`) for graceful Ctrl+C shutdown. The async pipeline uses a SENTINEL chain to propagate stop signals through all queues.

Output Management (`--save`)

When `--save` is enabled, a timestamped directory is created (e.g., `artifacts/python_example/{model}-image-{name}-{timestamp}/`) containing `run_info.txt` , saved images/video, and optional tensor dumps.

Headless Mode

When `DISPLAY` / `WAYLAND_DISPLAY` environment variables are absent, `cv2.imshow()` is automatically skipped. Use `--no-display` for explicit headless operation.

Model Configuration (`--config`)

Runtime parameters (thresholds, top-k, etc.) are loaded via `_FactoryConfigMixin` with alias normalization (`score_threshold` → `conf_threshold`). If omitted, `config.json` is auto-detected adjacent to the model or 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 silently keep the standard path, so the flag is always safe to add. 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 — YOLOv5 / YOLOv7, EfficientDet
Approximate		

© Copyright 2026 DEEPX All Rights Reserved.

Tier	Output vs. standard path	Model families
	Differs only at sub-pixel mask boundaries (binary masks agree at high IoU)	Instance segmentation (YOLOv8-seg), YOLACT, SegFormer semantic segmentation

- **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).

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_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 all post-processing results to `logs/verify/{model}.json` for inspection and debugging.

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

4.1.6 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.

4.2 DX-APP Python Example Tests

4.2.1 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 → Integration → CLI → E2E**.
- **Smart Mocking:** Combines high-speed software mocks (hardware-free) with real NPU hardware validation.
- **Centralized Configuration:** Model metadata is managed in a single source of truth (config.py) for simple maintenance.
- **Automated Performance Tracking:** E2E tests automatically capture and aggregate NPU performance metrics (FPS, Latency).

4.2.2 Test Classification & Prerequisites

Test Strategy Levels

Quick Start Tests

The framework employs a layered approach to isolate issues effectively

- **Unit Tests (`-m 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)

Advanced Test Selection

- **Integration Tests (`-m integration`)** Validates error handling (e.g., missing files) and resource cleanup during interrupts (`Ctrl+C`)
- **E2E (End-to-End) Tests (`-m e2e`):** High-fidelity tests using real `.dxnn` models and NPU hardware to capture actual performance

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Environment Setup

Configuration Reference

- `pytest.ini` : Defines custom markers (50+), log formats, and global settings.
- `conftest.py` : Contains shared fixtures, mock infrastructure, and setup/teardown utilities.

> **Coverage Scope**
>
> The Python test framework uses centralized model registration under `tests/python_example/framework/` . Adding a new source directory under `src/python_example/` does not automatically guarantee full test coverage until the corresponding test registration and mappings are updated.

4.2.3 Test Infrastructure

Project Structure & Reference

The test suite mirrors the example structure for consistency

```
tests/python_example/
├── framework/              # Test framework core
│   ├── config.py           # Model configurations (60+ models)
│   ├── base_test.py        # Unit test base
│   ├── groups_test.py      # Variant group test
│   ├── integration_test.py # Integration template
│   ├── cli_test.py         # CLI template
│   ├── e2e_test.py         # E2E template
│   └── performance_collector.py  # Performance metrics
│
├── test_visualization.py   # Visualization tests (sync + async)
│                           #   Output → tests/test_visualization_result/
python_example/{sync,async}/<task>/
│
├── object_detection/       # Object detection tests (25+ models)
├── classification/         # Classification tests
├── face_detection/         # Face detection tests
├── pose_estimation/        # Pose estimation tests
├── instance_segmentation/  # Instance segmentation tests
├── semantic_segmentation/  # Semantic segmentation tests
├── depth_estimation/       # Depth estimation tests
├── hand_landmark/          # Hand landmark tests
├── embedding/              # Embedding tests
├── obb_detection/          # OBB detection tests
├── image_denoising/        # Image denoising tests
├── image_enhancement/      # Image enhancement tests
```


© Copyright 2026 DEEPX All Rights Reserved.

```
├── super_resolution/       # Super resolution tests
└── ppu/                    # PPU model tests
```

Shared Module (`tests/common/`)

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

- `constants.py` : `TASK_IMAGE_MAP` , `MODEL_IMAGE_OVERRIDE` , path constants
- `utils.py` : `setup_environment()` , `discover_python_scripts()` , `normalize_model_name()`

Asset Management

Assets required for testing are automatically managed by scripts within the `scripts/` directory:

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

4.2.4 Test Execution Guide

Execution Methods

Method A. Quick Start Guide

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 + Integration + CLI + E2E)
pytest

# 3. Run only software-based tests (skips NPU hardware)
pytest -m "not e2e"

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

Method B. Manual Execution (Advanced Selection)

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Filter tests by model family or AI task to optimize development time.

```
# By model
pytest -m yolov7
pytest -m scrfd
pytest -m efficientnet

# By task type
pytest -m object_detection
pytest -m classification
pytest -m semantic_segmentation

# Combinations
pytest -m "unit and yolov7"
pytest -m "(yolov7 or yolov8) and not e2e"
```

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:** Use `E2E_DISPLAY=1 pytest -m e2e` to enable UI output (available for sync variants).

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

> **NOTE**
>
> Display is only supported for `sync` variants due to thread-safety constraints.

- **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.csv`

© Copyright 2026 DEEPX All Rights Reserved.

4.2.5 Maintenance & CI/CD

Continuous Integration (CI) Integration

- **Automated Regression**: Python tests are integrated into the CI pipeline to catch breaking changes in post-processing logic or CLI interfaces.
- **Performance Baseline**: Nightly E2E runs compare current FPS/Latency against historical baselines to detect performance regressions in the NPU software stack.
- **Reporting**: Test results are exported in JUnit XML format for integration with CI dashboards (e.g., Jenkins, GitHub Actions).

ⓒ Copyright 2026 DEEPX All Rights Reserved.

5. DX-APP Post-processing

5.1 DX-APP C++ Post-processing

5.1.1 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++.
- **Logic Unification:** By sharing the exact same codebase between C++ examples and Python bindings (`dx_postprocess`), the SDK ensures identical inference results across all development environments.
- **Modularity:** Each model family (YOLO, DeepLab, etc.) is isolated into its own module, allowing for targeted updates and lightweight linking.

5.1.2 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` : Converts segmentation logits into high-resolution per-pixel class labels or visual color maps.
- `*_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.sh`; 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

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` : Build configuration to compile the module into a standalone shared library.

```
src/postprocess/
├── CMakeLists.txt
├── README.md
├── centerpose/
├── classification/
├── damoyolo/
├── deeplabv3/
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
├── depth/
├── dncnn/
├── efficientdet/
├── embedding/
├── espcn/
├── face3d/
├── hand_landmark/
├── nanodet/
├── obb/
├── retinaface/
├── scrfd/
├── scrfd_ppu/
├── semantic_seg/
├── ssd/
├── ulfgfd/
├── yolact/
├── yolov5/
├── 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/
```

5.1.3 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

ⓒ Copyright 2026 DEEPX All Rights Reserved.

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

```
#include "yolov5_postprocess.h"

auto post_processor = YOLOv5PostProcess(conf_threshold, nms_threshold);
auto final_results = post_processor.postprocess(npu_output_tensors);
```

Method B. Python Usage

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

```
from dx_postprocess import YOLOv5PostProcess

post_processor = YOLOv5PostProcess(conf_threshold, nms_threshold)
final_results = post_processor.postprocess(npu_output_tensors)
```

Build Configuration

Compilation Guide

Post-processing libraries are compiled automatically during the main SDK build process.

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

The resulting shared libraries and headers are staged in the project build output directories and are then used by the C++ examples and Python binding build flow.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

5.2 DX-APP Python Post-processing

5.2.1 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 the DX-APP Python usage documentation in `docs/source/docs/05_DX-APP_Python_Example_Usage_Guide.md`.

© Copyright 2026 DEEPX All Rights Reserved.

5.2.2 Technical Strategy & Support

Technical Strategy

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 code from `src/postprocess/`. This guarantees that results in Python are 100% identical to those in C++.
- **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/`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Supported Models & Tasks

The library wraps **48 C++ post-processing classes** with pybind11 bindings, ensuring consistent results between C++ and Python implementations.

- **Object Detection:** YOLOv5, YOLOv7, YOLOv8, YOLOv9, YOLOv10, YOLOv11, YOLOv12, YOLOv26, YOLOX, NanoDet, DAMOYOLO, SSD, CenterPose, EfficientDet, YOLACT
- **Face Detection:** SCRFD, YOLOv5Face, RetinaFace, ULFGFD, Face3D
- **Pose Estimation:** YOLOv5Pose, YOLOv8Pose, VitPose
- **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
- **OBB Detection:** YOLOv26-OBB
- **Classification:** EfficientNet-family
- **Depth Estimation:** DepthPostProcess (FastDepth, SCDepth)
- **Image Denoising:** DnCNN
- **Super Resolution:** ESPCN, RealESRGAN
- **Image Enhancement:** Zero-DCE
- **Embedding:** ArcFace
- **Hand Landmark:** HandLandmark
- **PPU Variants:** YOLOv5-PPU, YOLOv7-PPU, YOLOv8-PPU, SCRFD-PPU, YOLOv5Pose-PPU, YOLOX-PPU, YOLOv3Tiny-PPU

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.

© Copyright 2026 DEEPX All Rights Reserved.

5.2.3 Installation & Setup

Prerequisites

Before building, ensure the following are installed on your system

- **Python:** 3.8 or higher
- **Compiler:** GCC 4.8+ or Clang 3.3+
- **Build Tool:** CMake 3.16 or higher

During the installation, `pybind11` is automatically cloned from GitHub to `extern/pybind11` if it is not already present.

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

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!')"
```


© Copyright 2026 DEEPX All Rights Reserved.

5.2.4 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/yolov9s/yolov9s_sync_cpp_postprocess.py --
model assets/models/yolov9-s_640x640.dxnn --image sample/img/sample_kitchen.jpg

# Async (Stream Inference)
python src/python_example/object_detection/yolov9s/yolov9s_async_cpp_postprocess.py -
-model assets/models/yolov9-s_640x640.dxnn --video assets/videos/dance-group.mov
```

Direct API Call

```
from dx_postprocess import YOLOv9PostProcess
import numpy as np

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

6. Project README

6.1 Overview

DX-APP is a production-ready suite of application templates designed to accelerate the development of AI services on **DEEPX NPUs**. It bridges the gap between raw model deployment and high-performance application engineering.

Key Features & Objectives

- **Rapid Deployment:** Ready-to-run examples across multiple AI task categories — Classification, Object Detection, Face Detection, Pose Estimation, Semantic/Instance Segmentation, Depth Estimation, OBB Detection, Embedding, and more.
- **Dual-Language Flexibility:** High-performance **C++** for production and **Python** for rapid prototyping, each with their own shared runtime layer (`src/cpp_example/common/` for C++, `src/python_example/common/` for Python).
- **Hardware Acceleration:** Native support for **PPU-enabled models** and **Async templates** that overlap pipeline stages to maximize FPS.
- **Modular Design:** Clean, task-oriented templates that serve as reusable blueprints for custom commercial applications.

Reference Documentation

For deeper technical specifications, refer to the `docs/` directory

- **Application Overview:** 01_DXNN_Application_Overview.md
- **Installation & Build Guide:** 02_DX-APP_Installation_and_Build.md

6.2 Quick Start Guide

Follow these steps to transition from a fresh installation to your first successful inference on DEEPX NPU.

Step 1. Environment Setup & Verification

ⓒ Copyright 2026 DEEPX All Rights Reserved.

First, verify that the NPU driver and DX-RT are correctly installed. This is a mandatory prerequisite.

```
## Verify hardware connection and driver status
dxrt-cli -s
```

> ⚠ **Caution: Prerequisite Check**
>
> If the command above fails, you **must** manually install the NPU Drivers and DX-RT before continuing with DX-APP setup. Refer to the installation and build documentation under `docs/source/docs/`.

Once hardware is verified, install the necessary toolchain and system libraries.

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

Step 2. Asset Acquisition

Download the required models and sample media files.

```
## Interactive mode (default) — select categories and models from a menu
./setup.sh

## Non-interactive — download all models automatically without prompts
./setup.sh --all

## Preview what would be downloaded (no actual download)
./setup.sh --dry-run

## Download only a specific category
./setup.sh --category=object_detection

## Download specific models by name
./setup.sh --models yolov8n yolov9s efficientnet_lite0

## Internal/air-gapped — copy from local mount instead of downloading from S3
./setup.sh --all --internal

## Internal mode with a custom local path
./setup.sh --all --internal --internal-path=/path/to/local/models
```

`setup.sh` Options

Option	Description
`--all`	Download all models non-interactively

© Copyright 2026 DEEPX All Rights Reserved.

Option	Description
`--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
`--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`	Force remove models if they exist
`--force-remove-videos`	Force remove videos if they exist
`--verbose`	Enable verbose logging

- **Models:** Saved to `assets/models/`. By default, an interactive menu lets you select which model categories and models to download. Use `--all` to skip the menu and download everything automatically.
- **Media:** Saved to `assets/videos/`.

For most users, `./setup.sh` is the only required entry point for asset preparation.

If you are maintaining examples rather than only consuming them, review DX Tool Guide.

Step 3. Build & Execution

Build the C++ binaries and the Python dx_postprocess bindings simultaneously.

```
## Default build (minimal — builds run_demo C++ targets and installs dx_postprocess)
./build.sh

## Full build (all C++ targets + install)
./build.sh --all

## For a clean rebuild, use: ./build.sh --clean
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
## Build specific targets only (faster incremental builds)
./build.sh --target yolov9s_sync yolov9s_async

## List all available build targets
./build.sh --target list
```

- **Output:** Binaries are located in `bin/`, and shared libraries are in their respective build folders.

Test the NPU performance using the YOLOv9 object detection template.

C++ Implementation (High Performance)

```
## Static Image Inference (Synchronous)
./bin/yolov9s_sync \
-m assets/models/yolov9-s_640x640.dxnn \
-i sample/img/sample_kitchen.jpg

## Video Stream Inference (Asynchronous)
./bin/yolov9s_async \
-m assets/models/yolov9-s_640x640.dxnn \
-v assets/videos/dance-group.mov
```

Python Implementation (Rapid Prototyping)

```
## Python Baseline (Synchronous)
python src/python_example/object_detection/yolov9s/yolov9s_sync.py \
   --model assets/models/yolov9-s_640x640.dxnn \
   --image sample/img/sample_kitchen.jpg

## Python Optimized (Asynchronous + C++ Post-processing)
python src/python_example/object_detection/yolov9s/yolov9s_async_cpp_postprocess.py \
  --model assets/models/yolov9-s_640x640.dxnn \
    --video assets/videos/dance-group.mov
```

Output and Analysis

Following execution, a window will render results (`boxes/masks`), and the console will output a **Performance Summary** (`Latency/FPS`). For additional usage details, refer to DX-APP C++ Usage Guide and DX-APP Python Usage Guide.

6.3 Prerequisites

Before building the templates, ensure your system meets the following hardware and software requirements.

© Copyright 2026 DEEPX All Rights Reserved.

A. DEEPX Runtime (DX-RT) and NPU Drivers

To utilize NPU acceleration, you **must** install the kernel-mode drivers and the user-space runtime library

- **DEEPX NPU Linux Driver:** Required for low-level NPU communication. Github Repository
- **DX-RT (Runtime & Tools):** The core library for model inference and hardware management. Github Repository

B. Development Toolchain and Libraries

The following tools are required to compile the C++ templates and the Python dx_postprocess bindings.

B-a. Build System

- **CMake:** Version 3.14 or higher.
- **Compiler:** C++14-compatible (GCC 7.5+, Clang, etc.).
- **Build Utility:** make or ninja.

B-b. Core Libraries

- **OpenCV:** Version 4.2.0 or higher (**4.5.5 recommended**). This is used for image I/O and pre/post-processing visualization.
- **Python Environment:** Python 3.8 or higher and pip are required for Python-based examples and pybind11 integration.

6.3.1 Development Workflow Overview

The process from environment setup to running your first AI application is divided into three main phases. For detailed commands and execution steps, please refer to the **Section. Quick Start Guide**.

- **Hardware & Driver Verification:** Ensure the NPU is recognized by the system using the `dxrt-cli` tool.
- **Asset & Dependency Preparation:** Install required libraries (OpenCV, Build tools) via `./install.sh` and prepare models/videos via `./setup.sh`. Model assets are fetched through the current DX-ModelZoo-based setup flow.
- **Build & Execution:** Compile the source code using `./build.sh` and run the generated binaries or Python scripts located in the `bin/` or `src/python_example/` directories.

For contributor workflows such as model onboarding, validation, filtered execution, and benchmarking, refer to DX Tool Guide.

© Copyright 2026 DEEPX All Rights Reserved.

6.4 Core Concepts & Architecture

6.4.1 Repository Layout

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

```
dx_app/
├── src/
│   ├── cpp_example/            # C++ end-to-end examples (347 models across 22 tasks)
│   │   └── common/             # ← Shared C++ runtime layer
│   │       ├── base/           #   Abstract interfaces (IFactory, IProcessor, ...)
│   │       ├── processors/     #   49 shared post-processors
│   │       ├── runner/         #   16 task-specific sync/async runner pairs
│   │       ├── inputs/         #   Image/Video/Camera/RTSP input sources
│   │       ├── visualizers/    #   13 task-specific visualizers
│   │       ├── config/         #   ModelConfig loader
│   │       ├── utility/        #   Labels, preprocessing, profiling, run_dir,
signal_handler, verify_serialize
│   │       └── third_party/    #   Header-only third-party libraries (nlohmann_json)
│   ├── python_example/         # Python end-to-end examples (347 models across 22
tasks)
│   │   └── common/             # ← Shared Python runtime layer
│   │       ├── base/           #   Abstract interfaces (IFactory, IProcessor, ...)
│   │       ├── processors/     #   49 shared post-processors
│   │       ├── runner/         #   SyncRunner, AsyncRunner, run_dir, verify_serialize,
args
│   │       ├── inputs/         #   Image/Video/Camera/RTSP input sources
│   │       ├── visualizers/    #   14 task-specific visualizers
│   │       ├── config/         #   ModelConfig loader
│   │       └── 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/
├── 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
│   ├── common/                 #   Shared test constants & utilities
│   ├── cpp_example/            #   C++ tests (CLI, E2E, visualization, features)
│   └── python_example/         #   Python tests (unit, integration, 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
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

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

> **User vs Contributor Guidance**
>
> This README is primarily a user-facing overview. If you are extending examples, onboarding new models, or maintaining the repository structure, use the contributor-oriented documents linked from this page.

6.4.2 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 refactored C++ tree is organized by **task** → **model family** → **variant**, with a shared `common/` layer providing base interfaces, 49 processors, 16 task-specific runner pairs, 13 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

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 `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).

6.4.3 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** use `src/postprocess/` directly. They have their own shared processors in `src/cpp_example/common/processors/` . The `src/postprocess/` library exists specifically for the pybind11 bridge.

© Copyright 2026 DEEPX All Rights Reserved.

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`: Facilitates the compilation of these modules into reusable shared libraries.

Functional Responsibilities

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.

6.5 Usage Reference

6.5.1 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`	

© Copyright 2026 DEEPX All Rights Reserved.

Flag	C++	Python	Description
			Path to `.dxnn` model file (auto-downloaded if missing)
`-i` / `--image`	`-i`	`--image`	Input image file or directory
`-v` / `--video`	`-v`	`--video`	Input video file
`-c` / `--camera`	`-c`	`--camera`	Camera device index
`-r` / `--rtsp`	`-r`	`--rtsp`	RTSP stream URL
`-l` / `--loop`	`-l` (default: auto)	`--loop` (default: 1)	Inference repeat count
`--no-display`	`--no-display`	`--no-display`	Disable visualization window
`-s` / `--save`	`--save`	`--save`	Save rendered output to run directory
`--save-dir`	`--save-dir`	`--save-dir`	Base output directory (default: `artifacts/`)
`--dump-tensors`	`--dump-tensors`	`--dump-tensors`	Dump raw input/output tensors to files
`--config`	`--config`	`--config`	Model config JSON path (auto-detected if omitted)
`-h` / `--help`	`-h`	`-h`	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.

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`	

© Copyright 2026 DEEPX All Rights Reserved.

Variable	Description
	When set to `1`, dumps post-processing results to `logs/verify/{model}.json` for numerical verification

6.5.2 C++ Templates

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

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

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

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

- **Asynchronous (`*_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.

6.5.3 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.
- **Unified Logic:** Shares the exact same decoding logic as the C++ examples, ensuring consistent inference results across all platforms.
- **Installation:** -Automatically compiled during `./build.sh`.
  : **Manual install:** `cd src/bindings/python/dx_postprocess && pip install .`.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

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.

The refactored Python tree is organized by **task** → **model family** → **variant**, with a shared `common/` layer providing base interfaces, 41 processors, generic sync/async runners, 10 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:

- **Classification:** EfficientNet, AlexNet, ResNet, MobileNet, etc.
- **Object Detection:** YOLOv5/v7/v8/v9/v10/v11/v12, YOLOX, NanoDet, DAMOYOLO, SSD
- **Face Detection:** SCRFD, YOLOv5Face, YOLOv7Face, RetinaFace
- **Pose Estimation:** YOLOv8-Pose
- **Segmentation:** BiSeNet, DeepLabV3+, SegFormer, YOLOv8Seg
- **Depth, Embedding, OBB, Denoising, Enhancement, Super Resolution, Hand Landmark, Attribute Recognition, Re-ID, PPU**

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)

© Copyright 2026 DEEPX All Rights Reserved.

6.6 Advanced Features

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

Auto-Download

When running any example (C++ or Python), if the specified model file is not found locally, the runner automatically attempts to download it via `setup_sample_models.sh`. Similarly, if a `--video` file is missing, `setup_sample_videos.sh` is invoked automatically. If the download fails, a clear error message with manual download instructions is displayed.

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

All runners register SIGINT/SIGTERM handlers for graceful shutdown. Pressing Ctrl+C during inference prints `"Interrupted by user"` and cleanly exits, releasing all resources.

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)
    dump_tensors/       # (if --dump-tensors) raw tensor files
```

Numerical Verification (`DXAPP_VERIFY`)

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`
- (3) Supports all 12 result types (Detection, Classification, Pose, Segmentation, etc.)

Tensor Dump (`--dump-tensors`)

Dumps raw input/output tensors for debugging. On exception, tensors are auto-dumped with a `reason.txt` file. C++ outputs `.bin` files; Python outputs `.npy` files.

© Copyright 2026 DEEPX All Rights Reserved.

Model Config (`--config`)

Runtime parameters (score threshold, NMS threshold, top-k) can be tuned per-model via `config.json`. If not specified, the runner auto-detects `config.json` adjacent to the model or script.

Version Compatibility

All runners verify:

- **DX-RT library** ≥ 3.0.0
- **Compiled model format** ≥ v7

Incompatible versions produce a clear error message before exit.

Headless Mode

Python runners detect the absence of `DISPLAY` / `WAYLAND_DISPLAY` and skip `cv2.imshow()` automatically. Use `--no-display` for explicit headless operation in both C++ and Python.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

7. DX-APP Developer Guide

7.1 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.

7.1.1 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
- extract a standalone package into the current example layout
- 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.

7.1.2 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

# 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, validates layout consistency, and performs a fast example-oriented test pass.

Step 2. Execution Modes

Interactive mode

ⓒ Copyright 2026 DEEPX All Rights Reserved.

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

```
./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
```

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`	Extract a standalone package into repository layout
`list`	List registered models
`search`	Search models by keyword

© Copyright 2026 DEEPX All Rights Reserved.

Command	Purpose
`info`	Show model details
`delete`	Remove a model from the repository
`new-task`	Create a new task directory skeleton
`validate`	Validate current model/example layout
`run`	Execute examples with filters
`bench`	Benchmark examples with filters
`help`	Show usage help

7.1.3 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`	Unique identifier (e.g., `yolov9s`)
`dxnn_file`	Compiled model filename (e.g., `YOLOV9S.dxnn`)
`add_model_task`	Task category (e.g., `object_detection`)
`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

The `add` command reads this registry to auto-generate factory files, `config.json`, and all entry-point scripts.

© Copyright 2026 DEEPX All Rights Reserved.

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 and then update the related test registration if the example should be covered by automated tests.

Validation & Verification

Basic validation

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

This runs code generation + NPU inference for all supported models.

`validate_models.sh` options

Option	Purpose
`--lang cpp|py|both`	Language filter
`--skip-verify`	Code generation only (no inference)
`--no-video`	Image-only mode
`--list`	Print commands without executing
`--clean`	Remove all generated packages
`--start-from <model>`	Resume from a specific model
`<task_filter>`	Filter by task (e.g., `object_detection`)

Execution & Benchmarking

- **Selective Run**: The `run` command abstracts the complexity of finding specific binaries. It uses filters to execute subsets of the 130+ 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.

7.1.4 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
```

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 current task/model layout. Typical information includes:

- target language (`cpp` , `py` , or both)
- task category
- model name
- post-processing selection
- sync-only or multi-variant generation choice

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

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

```
# Create from a postprocessor template
./scripts/add_model.sh yolo_custom detection --postprocessor yolov8

# Copy from an existing model directory (useful for same-family variants)
./scripts/add_model.sh yolov7_w6 detection --base-model yolov7 --postprocessor yolov7

# Generate, verify, and push in one step
./scripts/add_model.sh yolo_custom detection --postprocessor yolov8 --verify --model
assets/models/yolo_custom.dxnn --git-push
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

Common `--postprocessor` values include:

Task	Common values
Object detection	`yolov5`, `yolov7`, `yolov8`, `yolov9`, `yolov10`, `yolov11`, `yolov12`, `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 <name>`	Copy from a specific existing model directory instead of the default reference
`--lang <cpp|py|both>`	Target language (default: `both`)
`--verify`	Build and run inference verification after generation
`--model <path>`	`.dxnn` model file for `--verify`
`--no-video`	Skip video verification (image only)
`--git-push`	After successful `--verify`, commit and push
`--auto-add`	Batch-generate source packages for all unregistered models

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

Step 1-3. Verify the integration

© Copyright 2026 DEEPX All Rights Reserved.

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

```
# Check if all required files and registry entries are consistent
./scripts/dx_tool.sh validate

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

> **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.

Step 2. Packaging & Distribution

Step 2-1. Extract a standalone package

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

Use this when you need to convert an external model package into the repository layout used by DX-APP.

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 classification/resnet50 --output-dir out_resnet50
```

The output is created under:

```
out_resnet50\sln\classification\resnet50\
```

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\classification\resnet50\build\dxapp_resnet50_sln_package.sln
```


© Copyright 2026 DEEPX All Rights Reserved.

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

```
cd .\out_resnet50\sln\classification\resnet50
.\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
`DXRT_DIR`	DXRT SDK 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 classification/resnet50 --output-dir out_resnet50
--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. This helps catch mismatched files, missing variants, or incomplete model onboarding.

© Copyright 2026 DEEPX All Rights Reserved.

Step 3-2. Run examples selectively

```
# 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

7.1.5 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
```

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`
- `docs/11_DX-APP_Example_Source_Structure.md`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

7.2 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.

7.2.1 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

7.2.2 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 (49 hpp), runners (32 hpp), visualizers (13 hpp), input sources (5 hpp), config (1 hpp), and utilities (11 hpp)
- **`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 (41 py), runners (5 py), visualizers (10 py), input sources (5 py), config (2 py), and utilities (7 py)
- **`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 7-module structure: `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:

- `classification/`
- `object_detection/`
- `face_detection/`
- `pose_estimation/`
- `semantic_segmentation/`
- `instance_segmentation/`
- `depth_estimation/`
- `embedding/`
- `image_denoising/`
- `image_enhancement/`
- `super_resolution/`
- `obb_detection/`
- `ppu/`
- `hand_landmark/`
- `attribute_recognition/`
- `reid/`
- `face_alignment/`

This makes it easy to answer both questions below:

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

Model-First Subdirectories

Within each task directory, examples are split again by model family.

Examples:

```
src/cpp_example/object_detection/yolov9s/
src/cpp_example/object_detection/yolov8/
src/python_example/object_detection/yolov9s/
src/python_example/object_detection/ssdmv1/
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

This gives each model family its own isolated workspace for:

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

7.2.3 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 – assembles processor + visualizer + runner
│   ├── i_processor.hpp            #   IProcessor – postprocess(outputs, meta) → results
│   ├── i_visualizer.hpp           #   IVisualizer – draw(frame, results) → frame
│   └── i_input_source.hpp         #   IInputSource – image/video/camera/RTSP abstraction
├── config/
│   └── model_config.hpp           # ModelConfig – loads config.json
├── processors/                    # 46 shared processors
│   ├── yolov5_postprocessor.hpp
│   ├── yolov8_postprocessor.hpp
│   ├── scrfd_postprocessor.hpp
│   ├── nanodet_postprocessor.hpp
│   ├── damoyolo_postprocessor.hpp
│   ├── ssd_postprocessor.hpp
│   ├── segmentation_postprocessor.hpp       # DeepLabV3 plus FastSegmentationPostprocessor
│   ├── instance_seg_postprocessor.hpp
│   ├── depth_postprocessor.hpp
│   └── ...
├── runner/                        # 24 task-specific runner pairs
│   ├── sync_detection_runner.hpp  #   Sequential detection loop
```


© Copyright 2026 DEEPX All Rights Reserved.

```
│   ├── async_detection_runner.hpp #   Pipelined detection loop
│   ├── sync_classification_runner.hpp
│   ├── async_classification_runner.hpp
│   ├── sync_segmentation_runner.hpp
│   ├── async_segmentation_runner.hpp
│   └── ...                       #   12 sync + 12 async = 24 total
├── inputs/                        # 5 input source headers
│   ├── camera_source.hpp
│   ├── image_source.hpp
│   ├── video_source.hpp
│   ├── rtsp_source.hpp
│   └── input_factory.hpp
├── visualizers/                   # 12 task-specific visualizers
│   ├── detection_visualizer.hpp
│   ├── classification_visualizer.hpp
│   ├── segmentation_visualizer.hpp
│   ├── face_visualizer.hpp
│   ├── pose_visualizer.hpp
│   └── ...
├── utility/                       # 8 utility headers
│   ├── common_util.hpp
│   ├── labels.hpp
│   ├── preprocessing.hpp
│   ├── profiling.hpp
│   ├── run_dir.hpp
│   ├── safe_queue.hpp
│   ├── verify_serialize.hpp
│   └── visualization.hpp
└── 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       #   IFactory — assembles processor + visualizer + runner
│   ├── i_processor.py     #   IProcessor — postprocess(outputs, meta) → results
│   ├── i_visualizer.py    #   IVisualizer — draw(frame, results) → frame
│   └── i_input_source.py  #   IInputSource — image/video/camera/RTSP abstraction
├── config/
│   ├── config_schema.py   # Config schema validation for config.json values
│   └── model_config.py    # ModelConfig — loads config.json (input size, labels,
thresholds)
├── processors/             # 41 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-
```


© Copyright 2026 DEEPX All Rights Reserved.

```
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   # 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           # YOLOv8-Pose
│   ├── ppu_postprocessor.py            # PPU variants (YOLOv5/v7/SCRFD/Pose)
│   ├── 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, MiDaS
│   ├── nanodet_postprocessor.py        # NanoDet
│   ├── ssd_postprocessor.py            # SSD MobileNet
│   ├── damoyolo_postprocessor.py       # DAMOYOLO
│   ├── embedding_postprocessor.py      # CLIP, ArcFace
│   ├── restoration_postprocessor.py    # DnCNN, Zero-DCE
│   ├── nms_utils.py                    # Shared NMS / box utilities
│   ├── letterbox_preprocessor.py       # Shared letterbox preprocessing
│   └── ...
├── runner/                # 5 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.)
│   ├── run_dir.py         # Directory-based batch runner
│   └── 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/           # 10 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
│   └── ...
└── utility/                       # 7 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
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
    ├── safe_queue.py      # Thread-safe queue for async pipeline
    ├── skeleton.py        # Pose skeleton definitions
    └── visualization.py   # Drawing helpers (boxes, text, masks)
```

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 count	24 (12 sync + 12 async)	5 (2 runners + args + run_dir + verify)
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++ model directory

```
src/cpp_example/object_detection/yolov9s/
├── config.json
├── factory/
│   └── yolov9s_factory.hpp      # Assembles processor + visualizer from common/
├── yolov9s_sync.cpp               # Entry point → sync_detection_runner
└── yolov9s_async.cpp              # Entry point → async_detection_runner
```

Python model directory

```
src/python_example/object_detection/yolov9s/
├── config.json
├── factory/
│   └── yolov9s_factory.py               # Assembles processor + visualizer from
common/
├── yolov9s_sync.py                       # Entry point → SyncRunner
├── yolov9s_async.py                      # Entry point → AsyncRunner
├── yolov9s_sync_cpp_postprocess.py       # Entry point → SyncRunner + C++ binding
└── yolov9s_async_cpp_postprocess.py      # Entry point → AsyncRunner + C++ binding
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

The factory imports shared components from `common/`:

```cpp
// C++ factory example
#include "common/processors/yolov8_postprocessor.hpp"
#include "common/visualizers/detection_visualizer.hpp"
```

```python
# Python factory example
from common.processors import YOLOv5Postprocessor
from common.visualizers import DetectionVisualizer
```

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

```python
# Python
from common.runner import SyncRunner
runner = SyncRunner(factory)
runner.run()
```

This means adding a new model typically requires only a `config.json` and a factory file — the shared infrastructure handles everything else.

Variant Naming Rules

C++ variants

A typical C++ model directory contains:

```
config.json
factory/
<model>_sync.cpp
<model>_async.cpp
```

Example:

```
src/cpp_example/object_detection/yolov9s/
├── config.json
├── factory/
├── yolov9s_sync.cpp
└── yolov9s_async.cpp
```

Common C++ variant patterns:

- `*_sync.cpp`: sequential execution path
- `*_async.cpp`: pipelined or threaded execution path
- task/model-specific additional variants when required

© Copyright 2026 DEEPX All Rights Reserved.

Python variants

A typical Python model directory contains:

```
config.json
factory/
<model>_sync.py
<model>_async.py
<model>_sync_cpp_postprocess.py
<model>_async_cpp_postprocess.py
```

Example:

```
src/python_example/object_detection/yolov9s/
├── config.json
├── factory/
├── yolov9s_sync.py
├── yolov9s_async.py
├── yolov9s_sync_cpp_postprocess.py
└── yolov9s_async_cpp_postprocess.py
```

Common Python variant patterns:

- `*_sync.py` : Python-only synchronous path
- `*_async.py` : Python-only 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

---

7.2.4 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 7-module architecture:

Module	C++	Python	Role
`base/`	4 interfaces (.hpp)	4 interfaces (.py)	`IFactory` , `IProcessor` , `IVisualizer` , `IInputSource`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Module	C++	Python	Role
`config/`	`model_config.hpp`	`model_config.py`	Loads `config.json` (input size, labels, thresholds)
`processors/`	46 header files	41 Python files	Shared processors for all model families
`runner/`	24 runner headers	5 runner files	Sync/Async execution engines with profiling
`inputs/`	5 source headers	5 source files	Image, Video, Camera, RTSP input abstraction
`visualizers/`	12 visualizer headers	10 visualizer files	Task-specific result rendering
`utility/`	8 utility headers	7 utility files	Labels, preprocessing, profiling, drawing, run_dir, queue, verify

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** used by `cpp_example/` `common/processors/` directly. Instead, 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/`
	`src/postprocess/` via pybind11

© Copyright 2026 DEEPX All Rights Reserved.

Variant	Post-processing source
`*_sync_cpp_postprocess.py` / `*_async_cpp_postprocess.py`	

Numerical Verification (`DXAPP_VERIFY`)

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`
- **`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.

7.2.5 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": "YoloV9S.dxnn",
  "add_model_task": "object_detection",
  "postprocessor": "yolov8",
  "input_width": 640,
  "input_height": 640,
  "config": { "conf_threshold": 0.5, "num_classes": 80 },
  "supported": true
}
```

`scripts/add_model.sh`

The `add_model.sh` script reads an entry from the registry and auto-generates:

- `config.json` with correct input dimensions and thresholds
- `factory/{model}_factory.py` wired to the correct processor and visualizer
- 4 entry-point scripts (sync/async × python/cpp_postprocess)
- C++ equivalents under `src/cpp_example/`

© Copyright 2026 DEEPX All Rights Reserved.

This enables onboarding a new model with zero manual code in most cases.

Contributor Rules

Adding New Examples

When adding a new example, keep the following rules:

- (1) choose the correct task directory first
- (2) create a dedicated model directory under that task
- (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

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/model 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

7.2.6 Supplementary Information

Connection to Testing

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

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Shared Test Infrastructure (`tests/common/`)

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

- `constants.py` — paths, timeout values, suffix patterns (e.g., `_sync`, `_async`)
- `utils.py` — helper functions for executable/script discovery, process execution, result validation

C++ tests

C++ tests are organized into four categories:

Category	File	Description
CLI Help	`test_cli_help.py`	Validates `--help` output for all executables
CLI Basic	`test_cli_basic.py`	Validates `--model` argument handling
E2E	`test_e2e.py`	End-to-end inference with image/video inputs
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/SIGTERM graceful shutdown

All tests auto-discover executables from `bin/` using `tests/common/utils.py`.

Python tests

- Python example tests are driven by centralized configuration under `tests/python_example/framework/`
- `test_visualization.py` validates output image generation across all task types
- adding a new source directory alone does **not** automatically guarantee full test coverage

© Copyright 2026 DEEPX All Rights Reserved.

Relevant files include:

- `tests/python_example/framework/config.py`
- `tests/python_example/framework/performance_collector.py`
- `tests/python_example/test_visualization.py`
- `tests/python_example/<task>/` (14 task directories)
- `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

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
./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.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

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/`
- `tests/python_example/`
- `docs/10_DX-APP_DX-Tool_Guide.md`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

7.3 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

7.3.1 [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`	`YOLOv5Postprocessor`
YOLOv7 anchor-based detection	YOLOv7 anchors/strides	`yolov7`	`YOLOv7Postprocessor`	`YOLOv5Postprocessor` with model config
YOLOX	anchor-free YOLOX-style detection	`yolox`	`YOLOXPostprocessor`	`YOLOXPostprocessor`
YOLOv8/v9/v10/v11/v12/v26 detection		`yolov8`, `yolov9`, `yolov10`, `yolov11`,		`YOLOv8Postprocessor` family

© Copyright 2026 DEEPX All Rights Reserved.

Model type	Typical output	`add_model.sh --postprocessor`	C++ postprocessor	Python postprocessor
	anchor-free / DFL-style detection, no objectness field	`yolov12`, `yolov26`	matching `YOLOv*Postprocessor`	
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

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`.

7.3.2 [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 object_detection --postprocessor yolov8 --lang both

# YOLOv7-style anchor-based model
./scripts/add_model.sh yolov7_w6 object_detection --base-model yolov7 --postprocessor yolov7 --lang both
```


© Copyright 2026 DEEPX All Rights Reserved.

```
# YOLOX-style model
./scripts/add_model.sh custom_yolox object_detection --postprocessor yolox --lang bot
h
```

Generated files follow the standard layout:

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

src/python_example/object_detection/<model_name>/
├── config.json
├── factory/<model_name>_factory.py
├── <model_name>_sync.py
├── <model_name>_async.py
├── <model_name>_sync_cpp_postprocess.py
└── <model_name>_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.

7.3.3 [Step 3] Tune `config.json`

Most YOLO customization is done through each example's `config.json`. 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-style models with objectness
`score_threshold` or `conf_threshold`	class confidence threshold	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	custom dataset labels

© Copyright 2026 DEEPX All Rights Reserved.

Key	Meaning	Typical use
`anchors`	anchor map by stride	custom anchor-based YOLOv5/v7 variants
`strides`	output strides	non-standard multi-scale layouts

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

```
{
  "obj_threshold": 0.25,
  "score_threshold": 0.30,
  "nms_threshold": 0.45,
  "num_classes": 3,
  "class_names": ["person", "vehicle", "animal"],
  "anchors": {
    "8": [[10, 13], [16, 30], [33, 23]],
    "16": [[30, 61], [62, 45], [59, 119]],
    "32": [[116, 90], [156, 198], [373, 326]]
  },
  "strides": [8, 16, 32]
}
```

Example for a custom YOLOv8-style model:

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

> ⚠ **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.

7.3.4 [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`.

© Copyright 2026 DEEPX All Rights Reserved.

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 factory pattern:

```
def create_postprocessor(self, input_width: int, input_height: int):
    return YOLOv8Postprocessor(input_width, input_height, self.config)
```

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`.
- Specialized tasks use the correct task-specific visualizer and runner.

7.3.5 [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

© Copyright 2026 DEEPX All Rights Reserved.

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.

7.3.6 [Step 6] Build and Run

Build the generated targets:

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

Run C++ examples:

```
./bin/<model_name>_sync  -m assets/models/<Model>.dxnn -i sample/img/
sample_kitchen.jpg --no-display -l 1
./bin/<model_name>_async -m assets/models/<Model>.dxnn -v assets/videos/dance-
group.mov --no-display -l 1
```

Run Python examples:

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

python src/python_example/object_detection/<model_name>/<model_name>_async.py \
  --model assets/models/<Model>.dxnn \
  --video assets/videos/dance-group.mov \
```


© Copyright 2026 DEEPX All Rights Reserved.

```
  --no-display \
  --loop 1
```

Use saved output for visual checks:

```
./bin/<model_name>_sync -m assets/models/<Model>.dxnn -i sample/img/
sample_kitchen.jpg --save
```

7.3.7 [Step 7] Verify Before Registry Integration

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

Recommended sequence:

```
# Structure validation
./scripts/dx_tool.sh validate

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

# CLI smoke tests
./scripts/dx_tool.sh run --lang cpp --model <model_name>
./scripts/dx_tool.sh run --lang py --model <model_name>

# 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.

© Copyright 2026 DEEPX All Rights Reserved.

7.3.8 [Step 8] Update Registry and Test Metadata

When the model is intended to ship as a first-class example, update the repository metadata 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 customer-only artifacts unless the release explicitly includes them.

7.3.9 Troubleshooting

No detections appear

- Lower `score_threshold` / `conf_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.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- 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.
- 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 generated directory name matches `<model_name>`.
- Run `./scripts/dx_tool.sh validate` to catch layout issues.
- Check that CMake discovered the generated example directory.

7.3.10 Release Checklist

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.

7.4 DX-APP Agent-Driven Development Guide

7.4.1 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.

7.4.2 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)	—

ⓒ 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)
        │
        ▼
      dx-validator  (called automatically after generation)
```

7.4.3 Skills

Skills encapsulate reusable workflows that agents invoke during code generation.

General SWE Process

Skill	Description
`dx-swe-brainstorm`	Brainstorm and plan before any code generation
`dx-swe-tdd`	Test-driven development — validate each file immediately after creation
`dx-swe-verify`	Verify before claiming completion — evidence before assertions
`dx-swe-writing-plans`	Write structured implementation plans before touching code

© Copyright 2026 DEEPX All Rights Reserved.

Skill	Description
`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 5-level validation pyramid against generated code

© Copyright 2026 DEEPX All Rights Reserved.

7.4.4 Supported AI Tools

dx_app agent-driven development works with four AI coding tools. Each auto-loads the knowledge base through its own configuration.

Tool	Config Files	Agents Available
Claude Code	`CLAUDE.md`	All 6 agents via context routing
GitHub Copilot	`.github/copilot-instructions.md` 6 agents in `.github/agents/` 17 skills in `.github/skills/` 4 instructions in `.github/instructions/`	`@dx-app-builder`, `@dx-python-builder`, `@dx-cpp-builder`, `@dx-benchmark-builder`, `@dx-model-manager`, `@dx-validator`
Cursor	`.cursor/rules/dx-app.mdc` (always) 6 agent rules + 17 skill rules `python-example.mdc`, `cpp-example.mdc`, `tests.mdc` (27 total)	Free-form with auto-applied rules
OpenCode	`AGENTS.md`, `opencode.json` 6 agents in `.opencode/agents/` 17 skills in `.deepx/skills/`	`@dx-app-builder` or `/dx-agent-app-build-python`

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, SyncRunner/AsyncRunner usage, 4-variant naming
`src/cpp_example/**`	`cpp-example.instructions.md`	C++14 standard, RAII patterns, InferenceEngine API
`src/postprocess/**`	`postprocess.instructions.md`	Postprocessing conventions, pybind bindings

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Glob Pattern	Injected Instruction	Content
`tests/**`	`tests.instructions.md`	pytest patterns, fixtures, NPU markers

OpenCode Skills (Slash Commands)

General SWE Process

Slash Command	Description
`/dx-swe-brainstorm`	Brainstorm and plan before code generation
`/dx-swe-tdd`	Test-driven development with incremental validation
`/dx-swe-verify`	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

© Copyright 2026 DEEPX All Rights Reserved.

Slash Command	Description
`/dx-agent-app-validate`	Run the 5-level validation pyramid

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`		`.opencode/agents/dx-python-builder.md`	`.claude/agents/dx-python-builder.md`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Agent	Copilot (`@mention`)	OpenCode (`@mention`)	Claude Code (`.claude/agents/`)
	`.github/agents/dx-python-builder.agent.md`		
`dx-cpp-builder`	`.github/agents/dx-cpp-builder.agent.md`	`.opencode/agents/dx-cpp-builder.md`	`.claude/agents/dx-cpp-builder.md`
`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-swe-brainstorm`	`.deepx/skills/dx-swe-brainstorm/SKILL.md`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Skill	File
`/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`
`/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-swe-tdd`	`.deepx/skills/dx-swe-tdd/SKILL.md`
`/dx-agent-app-validate`	`.deepx/skills/dx-validate/SKILL.md`
`/dx-swe-verify`	`.deepx/skills/dx-swe-verify/SKILL.md`
`/dx-swe-writing-plans`	`.deepx/skills/dx-swe-writing-plans/SKILL.md`

Shared Knowledge Base (`.deepx/`)

The `.deepx/` directory is the canonical source of truth for all agent knowledge. Platform-specific files (`.github/` , `.cursor/` , `.opencode/`) are generated from `.deepx/` by

© Copyright 2026 DEEPX All Rights Reserved.

`dx-agent-gen generate --repo dx-runtime/dx_app`. It is NOT auto-loaded — agents and skills reference 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/`	17 directories	Detailed skill workflows (one `SKILL.md` per directory)
`.deepx/templates/`	Template files	Code generation templates
`.deepx/knowledge/`	Knowledge files	Domain-specific reference material
`.deepx/toolsets/`	5 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	Context-dependent rules activated by task type

7.4.5 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` routes to `dx-agent-app-build-python` skill. Asks 2-3 questions (variant, task type, model), generates files in `dx-`

© Copyright 2026 DEEPX All Rights Reserved.

Tool	How to Use
	`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 all 4 variants, runs `dx-validator`.
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. Routes to `dx-agent-app-build-cpp` skill.
GitHub Copilot	`@dx-cpp-builder` followed by the prompt.
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

Prompt:

© Copyright 2026 DEEPX All Rights Reserved.

```
"Download yolo26n model for DX-M1"
```

Tool	How to Use
Claude Code	`@dx-model-manager` followed by the prompt.
GitHub Copilot	`@dx-model-manager` followed by the prompt.
Cursor	Type the prompt directly.
OpenCode	`@dx-model-manager` 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 src/python_example/object_detection/yolo26n/`

Scenario 5: Build a Pose Estimation App

Prompt:

```
"Build a pose estimation app with yolo26n-pose"
```

Tool	How to Use
Claude Code	Type the prompt directly. Routes to `dx-agent-app-build-python` skill with `pose_estimation` task

© Copyright 2026 DEEPX All Rights Reserved.

Tool	How to Use
	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. `python-example.mdc` activates for generated files in `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. Routes to `dx-agent-app-build-python` skill with `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	Type the prompt directly. `python-example.mdc` activates for generated files in `src/python_example/instance_segmentation/`.
OpenCode	`@dx-app-builder` followed by the prompt, or `/dx-agent-app-build-python` skill directly.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Scenario 7: Build a Classification App

Prompt:

```
"Build an image classification app with EfficientNet-B0"
```

Tool	How to Use
Claude Code	Type the prompt directly. Routes to `dx-agent-app-build-python` skill with `classification` task type. 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. `python-example.mdc` activates for generated files in `src/python_example/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:

```
"Build an async high-performance detection app with yolo26n"
```

Tool	How to Use
Claude Code	Type the prompt directly. Routes to `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. `python-example.mdc` activates for generated async files.
OpenCode	`@dx-app-builder` followed by the prompt, or `/dx-agent-app-build-async` skill directly.

© Copyright 2026 DEEPX All Rights Reserved.

7.4.6 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 clarifying questions** — variant (`sync` / `async`), model precision, task type (`detection` , `classification` , `segmentation` , etc.)
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** — `dx-validator` runs checks and prints a summary

Mandatory Questions (HARD-GATE)

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.

7.4.7 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)
```


© Copyright 2026 DEEPX All Rights Reserved.

```
└── src/python_example/{task}/{model}/
    ├── __init__.py
    ├── config.json
    ├── {model}_factory.py
    ├── {model}_sync.py
    ├── {model}_async.py
    ├── {model}_sync_cpp_postprocess.py
    └── {model}_async_cpp_postprocess.py
```

Session ID format: `YYYYMMDD-HHMMSS_<agent>_<model>_<task>` where `<agent>` is `claude`, `copilot`, `cursor`, or `opencode` (e.g., `20260403-143022_claude_yolo26n_detection`).

Production Output (src/)

When you explicitly request production placement, files are written directly to `src/python_example/{task}/{model}/` — the standard source tree.

File Descriptions

File	Purpose
`config.json`	Model path, task type, input dimensions, label map
`{model}_factory.py`	Implements `IFactory` — the 5-method interface for pre/post processing
`{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

© Copyright 2026 DEEPX All Rights Reserved.

7.4.8 5-Level Validation Pyramid

`dx-validator` applies checks in ascending order of cost. Each level gates the next.

```
    ▲
       /5\       Performance benchmarks (FPS targets)
      /———\
     / 4   \     NPU integration tests (requires hardware)
    /———————\
   /   3     \   Smoke tests (--help, module import)
  /———————————\
 /     2       \ Config validation (model paths, task types)
/———————————————\
       1         Static checks (imports, naming, structure)
```

Level	What It Checks	Requires Hardware
1 — Static	Absolute imports, naming conventions, file structure, IFactory methods	No
2 — Config	`config.json` schema, `.dxnn` model path resolution, valid task types	No
3 — Smoke	`--help` flag runs without error, modules import cleanly	No
4 — NPU Integration	End-to-end inference on a sample image with NPU present	Yes
5 — Performance	FPS meets target thresholds for the model and accelerator	Yes

7.4.9 Validation Commands

```
# Static checks (11 checks across Level 1 and Level 2)
python .deepx/scripts/validate_app.py src/python_example/{task}/{model}/

# Include smoke tests (Levels 1–3)
python .deepx/scripts/validate_app.py src/python_example/{task}/{model}/ --smoke-test

# Framework integrity — verify .deepx/ directory structure
python .deepx/scripts/validate_framework.py
```


© Copyright 2026 DEEPX All Rights Reserved.

7.4.10 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/`	17	Skill workflow directories (one `SKILL.md` each)
`templates/`	—	Code generation templates
`knowledge/`	—	Domain-specific reference material
`toolsets/`	5	API references (InferenceEngine, IFactory, dxrt-cli, model registry, pybind helpers)
`memory/`	5	Persistent knowledge (common pitfalls, platform API notes, optimization patterns, camera/display notes, model config cache)
`contextual-rules/`	4	Coding standards, import rules, naming conventions, directory layout rules
`prompts/`	4	System prompts for each specialist agent
`scripts/`	2	`validate_app.py`, `validate_framework.py`

NOTE

Platform files (`.github/` , `.cursor/` , `.opencode/` , `CLAUDE.md` , `AGENTS.md`) are generated by `dx-agent-gen generate --repo dx-runtime/dx_app` . Do not edit them directly. Edit `.deepx/` source files and re-run the generator. A pre-commit hook runs `dx-agent-gen` automatically.

Agents read from these directories at task start. Memory files are updated when new patterns or fixes are discovered during development.

© Copyright 2026 DEEPX All Rights Reserved.

7.4.11 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.

Rules:

1. **CRITICAL** — Output `[DX-AGENT-DEV: START]` as the absolute first line of the first response, before any other text. This is non-negotiable even if the user says to proceed autonomously.
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.

7.4.12 Troubleshooting

- **Problem:** Agent writes relative imports (`from .factory import ...`)
  Cause: Default LLM behavior
  Solution: Use absolute imports only.
  Example: `from dx_app.python_example.detection.yolo26n.yolo26n_factory import ...`

- **Problem:** Factory class missing methods
  Cause: Incomplete `IFactory` implementation

© Copyright 2026 DEEPX All Rights Reserved.

Solution: Implement all 5 required methods: `create_preprocessor`, `create_postprocessor`, `create_label_map`, `create_input_config`, `create_visualizer`

- **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 model.

- **Problem:** NPU not available / device error
  Cause: Accelerator not detected by driver
  Solution: Run `dxrt-cli -s` to check device status and verify the DEEPX kernel module is loaded.

- **Problem:** `validate_app.py` fails immediately
  Cause: Python path or venv not configured
  Solution: Activate the dx_app virtual environment and ensure `PYTHONPATH` includes the project root.

7.4.13 Further Reading

- DX-APP Project Overview
- DX-APP Python Example Usage Guide
- DX-APP C++ Example Usage Guide
- DX-APP Example Source Structure

© Copyright 2026 DEEPX All Rights Reserved.

8. Appendix

8.1 Third-Party License Notice

8.1.1 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.

8.1.2 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 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.

© Copyright 2026 DEEPX All Rights Reserved.

8.1.3 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.

9. RELEASE_NOTES

9.1 DX-APP v3.2.1 / 2026-07-28

9.1.1 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.

9.1.2 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

9.1.3 3. Added

9.2 DX-APP v3.2.0 / 2026-06-25

9.2.1 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
- 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`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- 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

9.2.2 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`)

9.2.3 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`

© Copyright 2026 DEEPX All Rights Reserved.

- Added DX-APP build selection options for minimal and category-based builds, plus Windows build selection TUI(Text User Interface).
- Added knowledge base (`.deepx/`) : specialized agents, app-building/SWE skills, and multi-platform agent-instruction generation (CLAUDE/AGENTS/copilot/cursor) via `dx-agent-gen`
- 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

---

9.3 DX-APP v3.1.1 / 2026-04-21

9.3.1 1. Changed

9.3.2 2. Fixed

- Fixed typo error in document(DX-APP User Manual)

9.3.3 3. Added

- Added License information for third-party models & datasets

---

ⓒ Copyright 2026 DEEPX All Rights Reserved.

9.4 DX-APP v3.1.0 / 2026-04-06

9.4.1 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
- Display windows now show original resolution frames with auto-sized window (~1/4 screen area)

9.4.2 2. Fixed

9.4.3 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 taskcategories 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`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- 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

---

9.5 DX-APP v3.0.2 / 2026-02-10

9.5.1 1. Changed

- Copy of dxrt and vkpkg DLLs into the dx-app/bin directory when building with MSVC.

9.5.2 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

9.5.3 3. Added

- Added vcpkg installation script for windows build.

9.6 DX-APP v3.0.1 / 2026-02-05

9.6.1 1. Changed

9.6.2 2. Fixed

- Hardcoded attribute size in YOLO post-processing to dynamically adjust based on model output shape

9.6.3 3. Added

- Added yolov26 cls, yolo26 pose, yolo26 seg, yolo26 obb examples

---

ⓒ Copyright 2026 DEEPX All Rights Reserved.

9.7 DX-APP v3.0.0 / 2026-01-02

9.7.1 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
- 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

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- 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

9.7.2 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
- 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

9.7.3 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

ⓒ Copyright 2026 DEEPX All Rights Reserved.

• 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)

• 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

9.7.4 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/`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- 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`)

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

9.7.5 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`)

ⓒ Copyright 2026 DEEPX All Rights Reserved.

9.7.6 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)`.

9.8 DX-APP v2.1.0 / 2025-11-28

9.8.1 1. Changed

- Enhanced build script documentation and usage instructions
- Updated cmake configuration in build.bat to use C++17 and v143 for 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.

9.8.2 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)

ⓒ Copyright 2026 DEEPX All Rights Reserved.

9.8.3 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
- 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

9.8.4 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).

9.9 DX-APP v2.0.0 / 2025-08-14

9.9.1 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

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- 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

9.9.2 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'

9.9.3 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

ⓒ Copyright 2026 DEEPX All Rights Reserved.

9.10 DX-APP v1.11.0 / 2025-07-24

9.10.1 1. Changed

- feat: enhance --clean option in build script for pybind artifacts
- feat: update dxnn models version(1.40.2 to 1.60.1)
- feat: auto run setup script or display a guide message when a file not found error occurs during example execution

9.10.2 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

9.11 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 oject detection
- template : classification
- template : object detection
- template : python example (sync/async/pybind c++)

ⓒ Copyright 2026 DEEPX All Rights Reserved.