DXNN Application (DX-APP) User Manual

2026/03/31 (v3.1.0)

DEEPX.ai

Copyright 2026 DEEPX All Rights Reserved.

Table of contents

1. DXNN Application Overview 4
   1.1 DEEPX SDK Architecture 4
   1.2 DX-APP Features 5
   1.3 DX-APP Core Design & Capabilities 8
2. DX-APP 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 87
8. RELEASE_NOTES 102
   8.1 v3.1.0 / 2026-04-06 102
   8.2 v3.0.4 / 2026-03-27 103
   8.3 v3.0.3 / 2026-03-13 104

Copyright 2026 DEEPX All Rights Reserved.

8.4 v3.0.2 / 2026-02-10 106

8.5 v3.0.1 / 2026-02-05 107

8.6 v3.0.0 / 2026-01-02 107

8.7 v2.1.0 / 2025-11-28 112

8.8 v2.0.0 / 2025-08-14 113

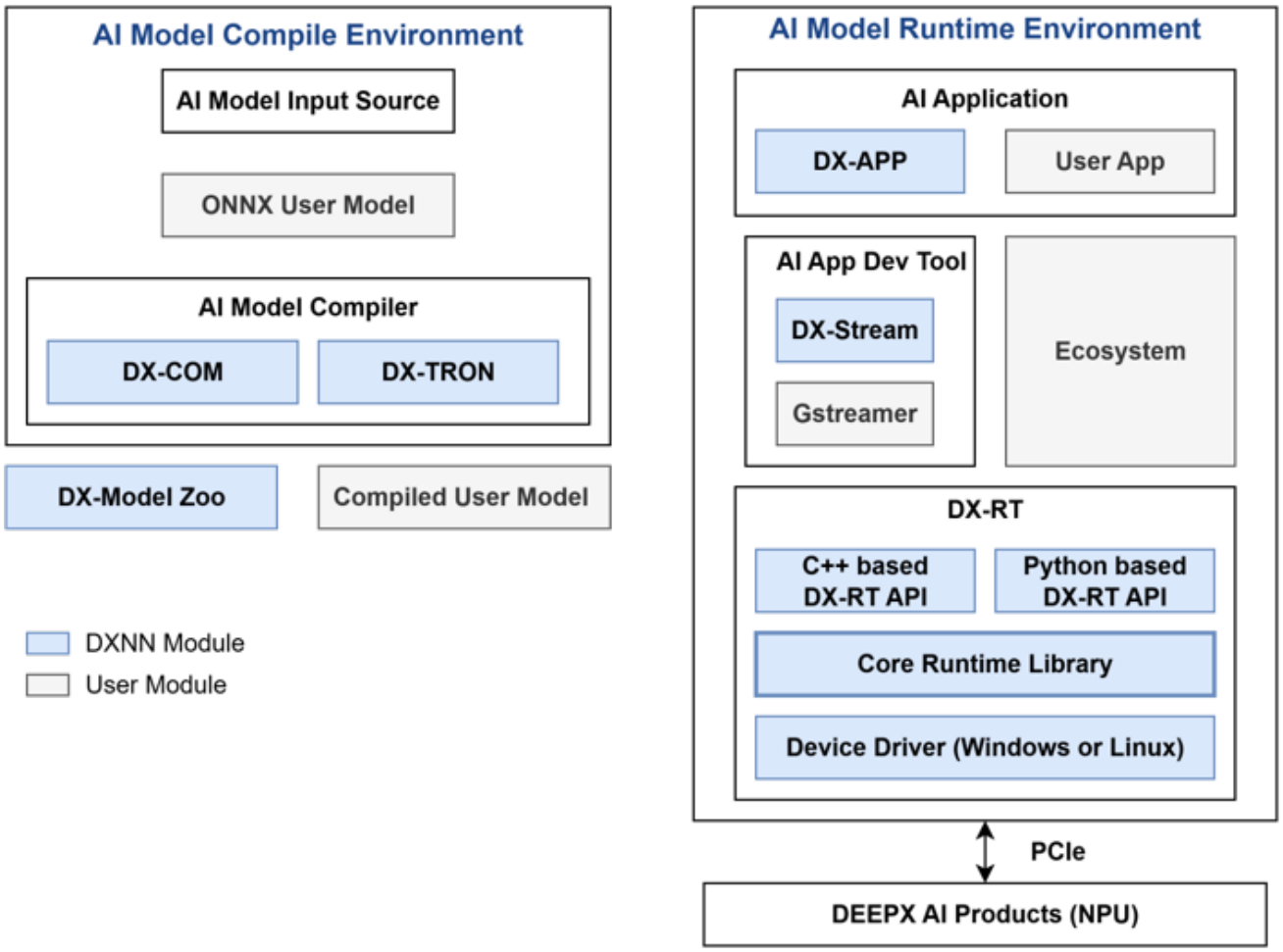
8.9 v1.11.0 / 2025-07-24 114

8.10 v1.10.0 / 2025-06-17 115

Copyright 2026 DEEPX All Rights Reserved.

1. DXNN Application Overview

1.1 DEEPX SDK Architecture



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

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

Copyright 2026 DEEPX All Rights Reserved.

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

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

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

DX-APP is a collection of runnable example applications that execute compiled models on DEEPX NPUs through DX-RT. It provides examples across 17 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, and face alignment. 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.

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.

Copyright 2026 DEEPX All Rights Reserved.

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 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.dxnn -i ./sample/
ILSVRC2012/0.jpeg -l 1
./bin/efficientnet_lite0_sync  -m ./assets/models/EfficientNet_Lite0.dxnn -i ./sample/
ILSVRC2012/0.jpeg -l 1
```

Object Detection (YOLOv8N)

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

```
./bin/yolov8n_async -m ./assets/models/YoloV8N.dxnn -i ./sample/img/
sample_kitchen.jpg --no-display -l 1
./bin/yolov8n_sync  -m ./assets/models/YoloV8N.dxnn -i ./sample/img/
sample_kitchen.jpg --no-display -l 1 -s
```


Copyright 2026 DEEPX All Rights Reserved.

Face Detection (SCRFD)

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

```
./bin/scrfd500m_async -m ./assets/models/SCRFD500M.dxnn -i ./sample/img/
sample_face.jpg --no-display -l 1
./bin/scrfd500m_sync  -m ./assets/models/SCRFD500M.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/yolov8s_pose.dxnn -i ./sample/img/
sample_kitchen.jpg --no-display -l 1
./bin/yolov8s_pose_sync  -m ./assets/models/yolov8s_pose.dxnn -i ./sample/img/
sample_kitchen.jpg --no-display -l 1 -s
```

Segmentation (DeepLabV3+)

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

```
./bin/deeplabv3plusmobilenet_async -m ./assets/models/DeepLabV3PlusMobilenet.dxnn -
i ./sample/img/sample_parking.jpg --no-display -l 1
./bin/deeplabv3plusmobilenet_sync  -m ./assets/models/DeepLabV3PlusMobilenet.dxnn -
i ./sample/img/sample_parking.jpg --no-display -l 1 -s
```


Copyright 2026 DEEPX All Rights Reserved.

Semantic Segmentation (BiSeNetV1)

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

```
./bin/bisenetv1_async -m ./assets/models/BiSeNetV1.dxnn -i ./sample/img/
sample_parking.jpg --no-display -l 1
./bin/bisenetv1_sync  -m ./assets/models/BiSeNetV1.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.

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.

Copyright 2026 DEEPX All Rights Reserved.

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

Copyright 2026 DEEPX All Rights Reserved.

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

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 ## 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.

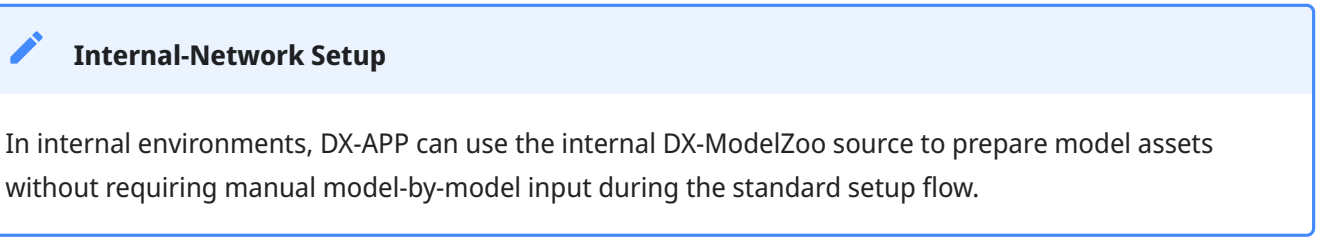
Copyright 2026 DEEPX All Rights Reserved.

`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.



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.

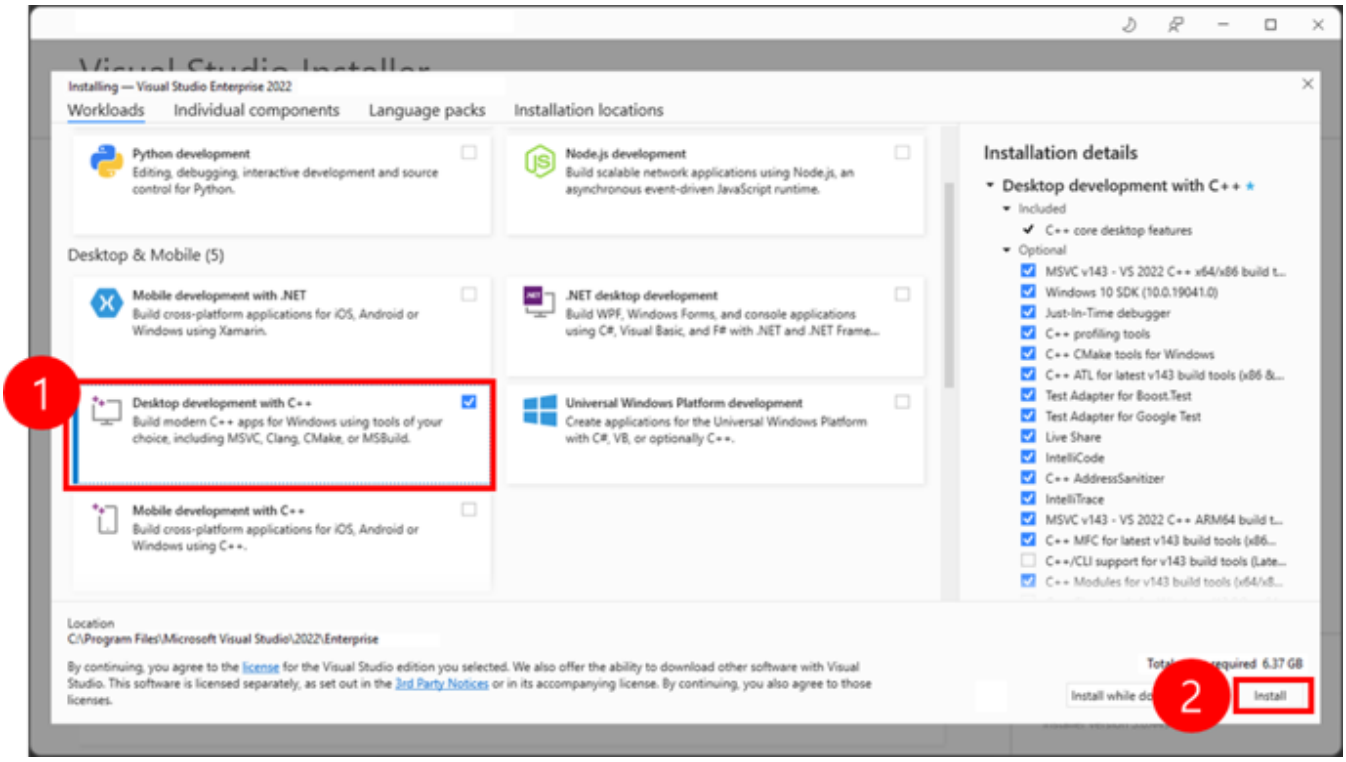
Copyright 2026 DEEPX All Rights Reserved.

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.

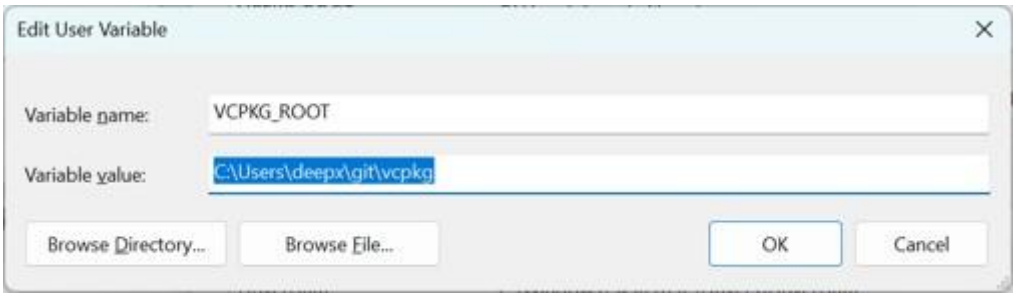
Copyright 2026 DEEPX All Rights Reserved.

> **NOTE**
>
> If you are using Visual Studio Community 2022, **VCPKG is pre-installed**. and 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

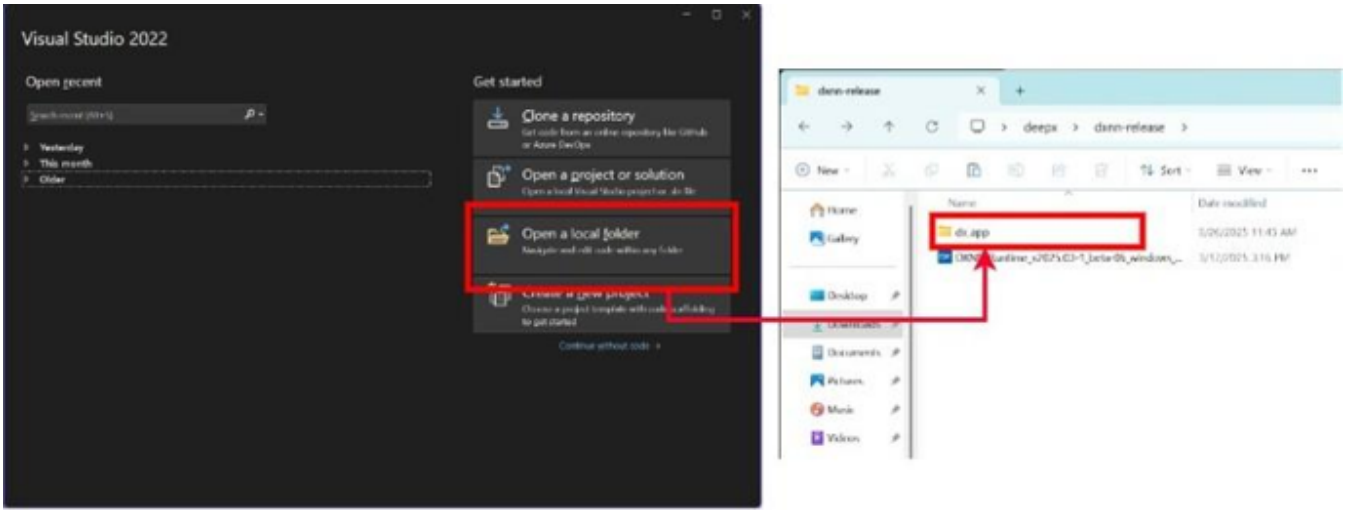
To build and run the `dx_app` application on Windows, follow the steps below using Visual Studio Community 2022.

Step 1. Open Project Folder

- **Step 1.** Launch Visual Studio Community 2022
- **Step 2.** From the start screen, select **Open a local folder**
- **Step 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.

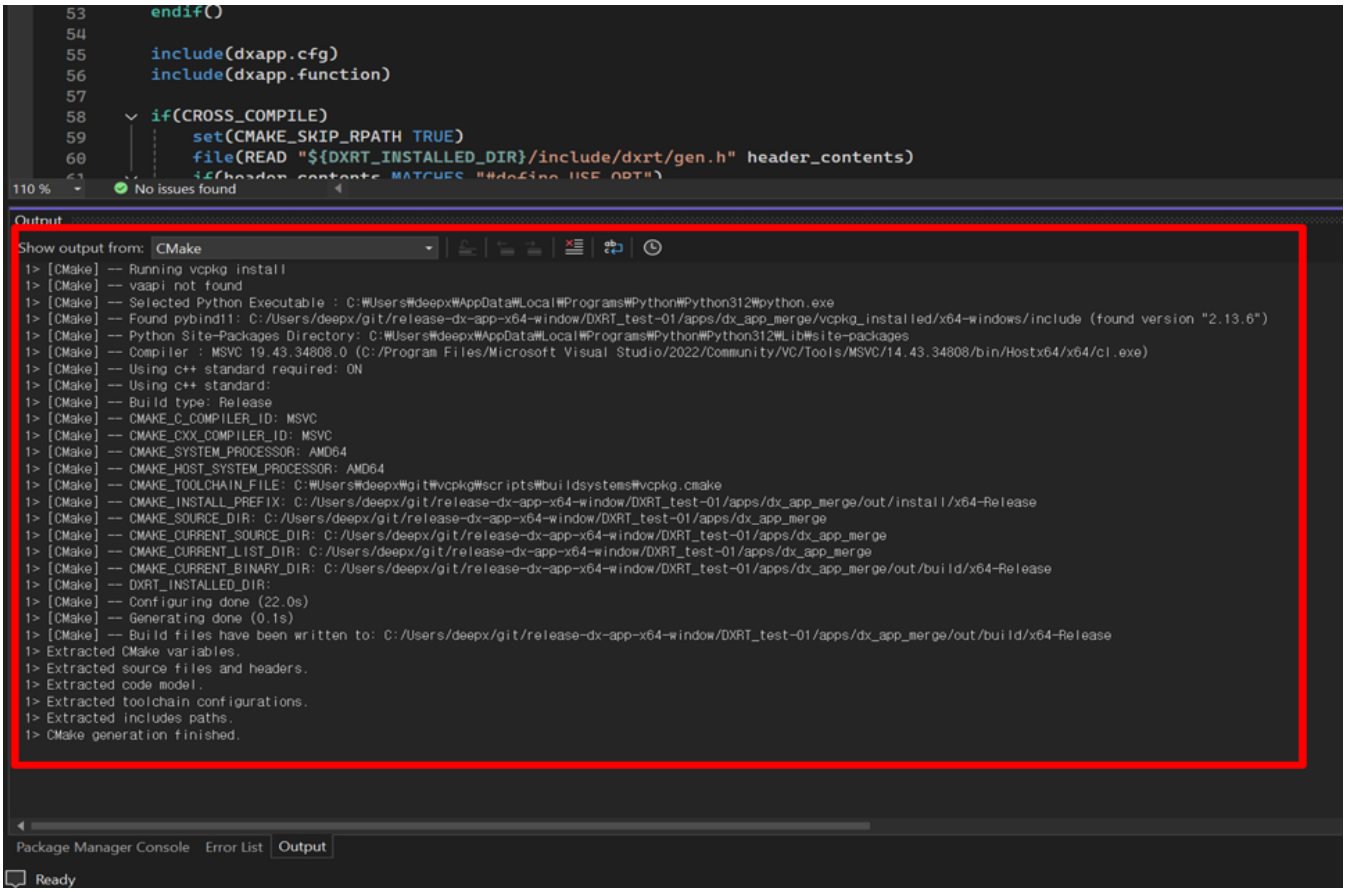


Step 2. Project Configuration

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

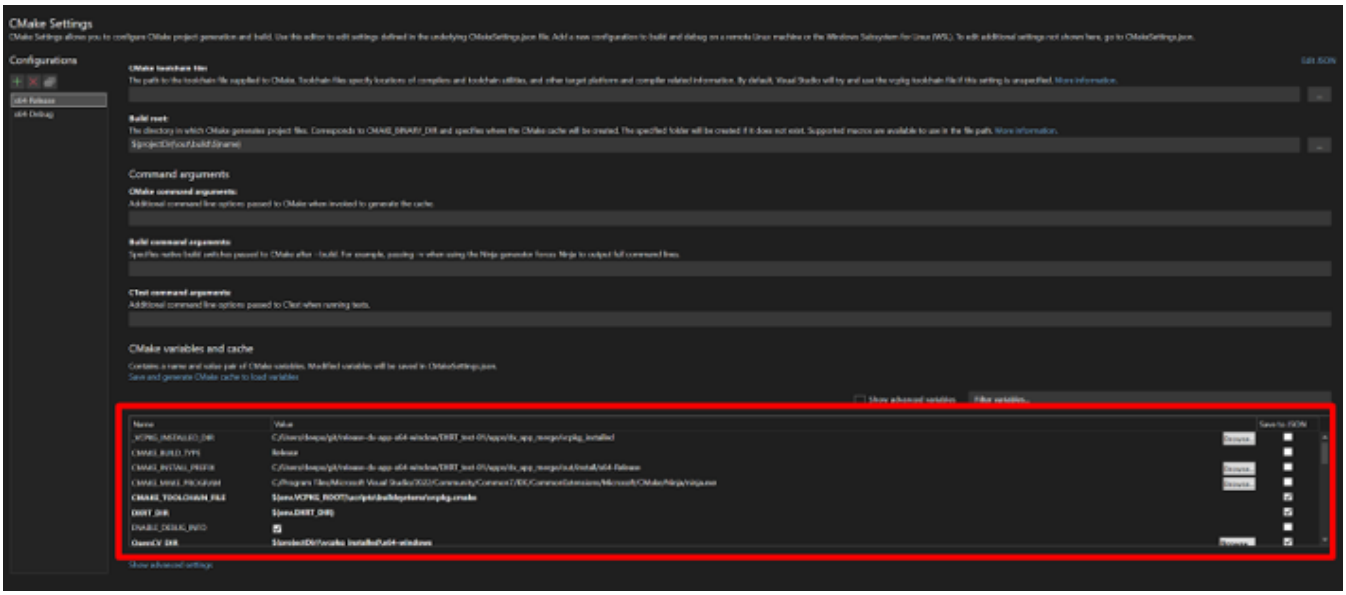
Step 4. Register `PATH` Variable

Copyright 2026 DEEPX All Rights Reserved.

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.



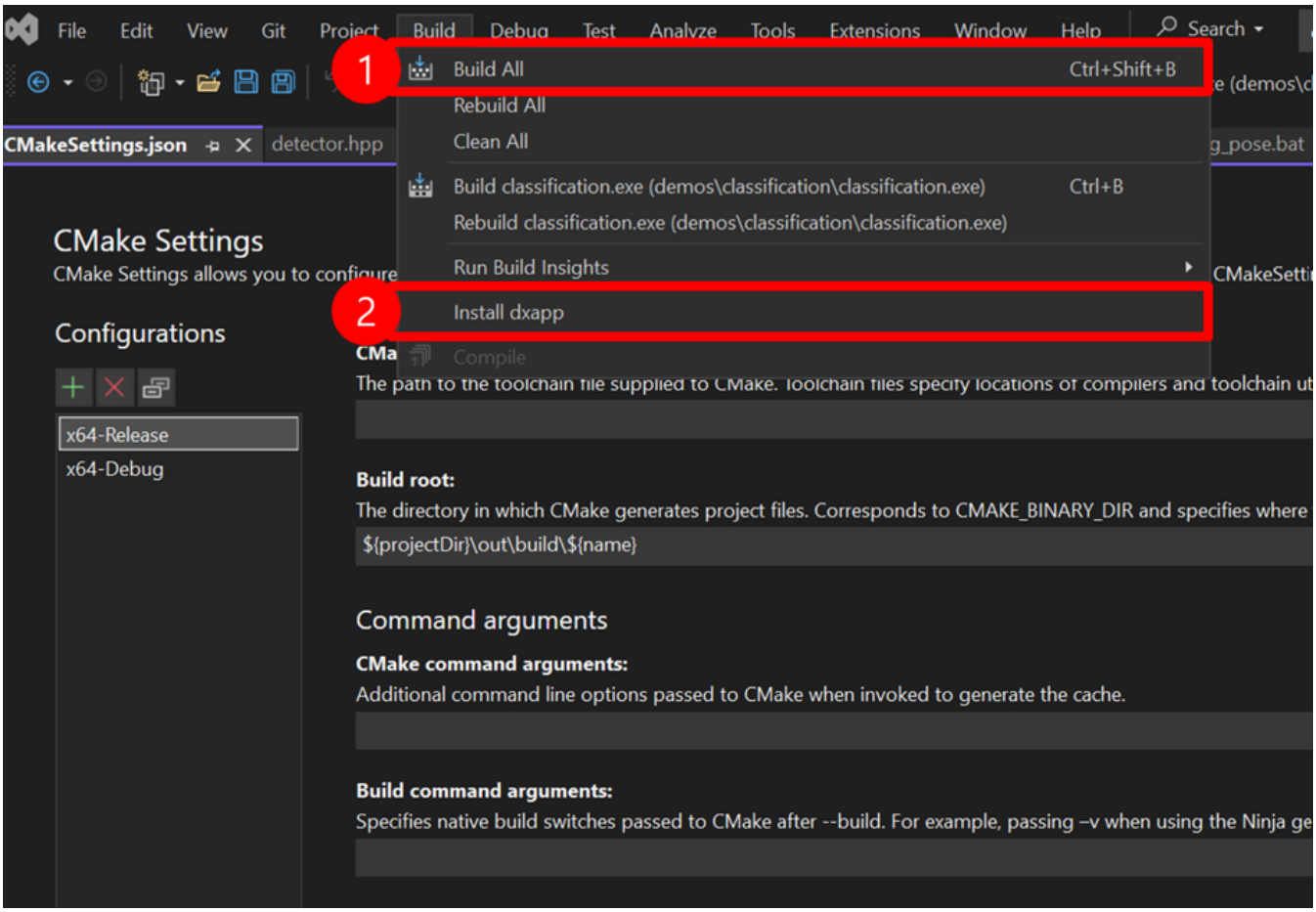
Step 5. Build and Install `dx_app`

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`

Copyright 2026 DEEPX All Rights Reserved.

Usage

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`

Copyright 2026 DEEPX All Rights Reserved.

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

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.dxnn -i ./
sample/ILSVRC2012/0.jpeg
```

object detection example

```
./bin/yolov8n_sync.exe  -m ./assets/models/YoloV8N.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/`)

Copyright 2026 DEEPX All Rights Reserved.

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`	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

Copyright 2026 DEEPX All Rights Reserved.

- **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.

> **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/YoloV9S.dxnn -i sample/img/sample_kitchen.jpg
./bin/yolov9s_async -m assets/models/YoloV9S.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

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.

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` . Use `scripts/verify_inference_output.py` to validate correctness.

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
pytest -m "e2e and sync_exec" -k "yolov5"   # Only sync yolov5 tests
```


Copyright 2026 DEEPX All Rights Reserved.

```
# 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
```


Copyright 2026 DEEPX All Rights Reserved.

Report Generation

Trigger coverage analysis using the following commands

Execution Flow (Manual vs Unified)

Manual Execution (from `tests/cpp_example/`)

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

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

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

Unified Test Runner (from project root)

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

Copyright 2026 DEEPX All Rights Reserved.

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
`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.

Copyright 2026 DEEPX All Rights Reserved.

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/
│   └── 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

Copyright 2026 DEEPX All Rights Reserved.

CLI Interface

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

Copyright 2026 DEEPX All Rights Reserved.

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
`--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)
`--output`	`-o`	string	Output file path (restoration/depth/SR only)
`--help`	`-h`	—	Show usage

Copyright 2026 DEEPX All Rights Reserved.

- **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.

> **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/YoloV9S.dxnn --image sample/img/sample_kitchen.jpg
python src/python_example/object_detection/yolov9s/yolov9s_async_cpp_postprocess.py --
model assets/models/YoloV9S.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.

Copyright 2026 DEEPX All Rights Reserved.

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:

```
[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.

Verification & Diagnostics

Numerical Verification (`DXAPP_VERIFY`)

Set `DXAPP_VERIFY=1` to serialize all post-processing results to `logs/verify/{model}.json` . Use `scripts/verify_inference_output.py` to validate correctness against task-specific rules.

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.

Copyright 2026 DEEPX All Rights Reserved.

Model Validation (optional)

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

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)

Copyright 2026 DEEPX All Rights Reserved.

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

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
```


Copyright 2026 DEEPX All Rights Reserved.

```
├── 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
├── 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"
```


Copyright 2026 DEEPX All Rights Reserved.

```
# 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)

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.

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/
├── depth/
├── dncnn/
├── efficientdet/
├── embedding/
├── espcn/
├── face3d/
├── hand_landmark/
├── nanodet/
├── obb/
├── retinaface/
├── scrfd/
```


Copyright 2026 DEEPX All Rights Reserved.

```
├── 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

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

```cpp
#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

Copyright 2026 DEEPX All Rights Reserved.

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 **41 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
- **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
- **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

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.

Copyright 2026 DEEPX All Rights Reserved.

Installation Steps

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

Method	Command	Recommended For
Full Build	`./build.sh`	Initial environment setup and full SDK deployment
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!')"
```

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/YoloV9S.dxnn --image sample/img/sample_kitchen.jpg

# Async (Stream Inference)
```


Copyright 2026 DEEPX All Rights Reserved.

```
python src/python_example/object_detection/yolov9s/yolov9s_async_cpp_postprocess.py --
model assets/models/YoloV9S.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
```


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`	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.

```
## Standard build
./build.sh

## 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/YoloV9S.dxnn \
-i sample/img/sample_kitchen.jpg

## Video Stream Inference (Asynchronous)
./bin/yolov9s_async \
-m assets/models/YoloV9S.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/YoloV9S.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/YoloV9S.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 (280 models across 17 tasks)
│   │   └── common/             # ← Shared C++ runtime layer
│   │     ├── base/           #   Abstract interfaces (IFactory, IProcessor, ...)
│   │     ├── processors/     #   45 shared processors (42 post + 3 pre)
│   │     ├── runner/         #   24 task-specific sync/async runner pairs
│   │     ├── inputs/         #   Image/Video/Camera/RTSP input sources
│   │     ├── visualizers/    #   12 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 (280 models across 17 tasks)
│   │   └── common/             # ← Shared Python runtime layer
│   │     ├── base/           #   Abstract interfaces (IFactory, IProcessor, ...)
│   │     ├── processors/     #   35 shared post-processors
│   │     ├── runner/         #   SyncRunner, AsyncRunner, run_dir, verify_serialize, args
│   │     ├── inputs/         #   Image/Video/Camera/RTSP input sources
│   │     ├── visualizers/    #   10 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
```

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

Copyright 2026 DEEPX All Rights Reserved.

> **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, 45 processors, 24 task-specific runners, 12 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.

Copyright 2026 DEEPX All Rights Reserved.

Common Arguments

Flag	C++	Python	Description
`-m` / `--model`	`-m`	`--model`	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.

Copyright 2026 DEEPX All Rights Reserved.

Environment Variables

Variable	Description
`DXAPP_SAVE_IMAGE`	When set to a file path, saves the visualization output to that path (no `--save` required)
`DXAPP_VERIFY`	When set to `1`, dumps post-processing results to `logs/verify/{model}.json` 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, 35 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 verification pipeline for validating inference correctness:

- (1) Set `DXAPP_VERIFY=1` before running any example
- (2) Post-processing results are serialized to `logs/verify/{model}.json`
- (3) Run `scripts/verify_inference_output.py` to validate against task-specific rules
- (4) Supports all 12 result types (Detection, Classification, Pose, Segmentation, etc.)

Tensor Dump (`--dump-tensors`)

Copyright 2026 DEEPX All Rights Reserved.

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.

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/validate_models.sh`
- `./scripts/verify_inference_output.py`
- `./scripts/run_examples.sh`
- `./scripts/bench_models.sh`

Key data files:

- `./config/model_registry.json` — model registry, single source of truth
- `./scripts/inference_verify_rules.json` — numerical verification thresholds per task

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

Copyright 2026 DEEPX All Rights Reserved.

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
`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

Copyright 2026 DEEPX All Rights Reserved.

`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.

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.

Numerical verification

```
./scripts/validate_models.sh --numerical --lang py
```


Copyright 2026 DEEPX All Rights Reserved.

This additionally verifies that inference outputs are numerically correct:

- (1) **Inference** — runs each model through NPU
- (2) **Serialization** — `common/runner/verify_serialize.py` converts results to JSON
- (3) **Validation** — `scripts/verify_inference_output.py` checks results against `scripts/inference_verify_rules.json`

Verification covers 17 task types: bounding boxes, confidence ranges, class IDs, keypoints, segmentation masks, depth maps, embeddings, attributes, re-identification, face alignment, etc.

`validate_models.sh` options

Option	Purpose
`--lang cpp|py|both`	Language filter
`--numerical`	Enable output verification
`--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

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.

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:

```
# Create from a postprocessor template
./scripts/add_model.sh yolov30 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 yolov30 detection --postprocessor yolov8 --verify --model assets/
models/YoloV30.dxnn --git-push
```


Copyright 2026 DEEPX All Rights Reserved.

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/`.

1-3. Verify the integration

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.

Copyright 2026 DEEPX All Rights Reserved.

Step 2. Packaging & Distribution

2. 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 3. Repository Maintenance

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.

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.

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

Copyright 2026 DEEPX All Rights Reserved.

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

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
```


Copyright 2026 DEEPX All Rights Reserved.

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`
- `scripts/validate_models.sh`
- `scripts/verify_inference_output.py`
- `scripts/inference_verify_rules.json`
- `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 (45 hpp), runners (24 hpp), visualizers (12 hpp), input sources (5 hpp), config (1 hpp), and utilities (8 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 (35 py), runners (5 py), visualizers (10 py), input sources (5 py), config (1 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/                    # 45 shared processors
│   ├── yolov5_postprocessor.hpp
│   ├── yolov8_postprocessor.hpp
│   ├── scrfd_postprocessor.hpp
│   ├── nanodet_postprocessor.hpp
│   ├── damoyolo_postprocessor.hpp
│   ├── ssd_postprocessor.hpp
│   ├── segmentation_postprocessor.hpp
│   ├── instance_seg_postprocessor.hpp
│   ├── depth_postprocessor.hpp
│   └── ...
├── runner/                        # 24 task-specific runner pairs
│   ├── sync_detection_runner.hpp  #   Sequential detection loop
│   ├── async_detection_runner.hpp #   Pipelined detection loop
│   ├── sync_classification_runner.hpp
│   ├── async_classification_runner.hpp
│   ├── sync_segmentation_runner.hpp
```


Copyright 2026 DEEPX All Rights Reserved.

```
│   ├── 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/
│   └── model_config.py    # ModelConfig — loads config.json (input size, labels,
thresholds)
├── processors/             # 35 shared processors
│   ├── yolo_postprocessor.py           # YOLOv5/v7/v8/v9/v10/v11/v12/YOLOX
│   ├── face_postprocessor.py           # SCRFD, YOLOv5Face, YOLOv7Face
│   ├── segmentation_postprocessor.py   # BiSeNet, DeepLabV3+, SegFormer
│   ├── instance_seg_postprocessor.py   # YOLOv8Seg, YOLOv26Seg
│   ├── obb_postprocessor.py            # YOLOv26OBB
│   ├── pose_postprocessor.py           # YOLOv8-Pose
│   ├── ppu_postprocessor.py            # PPU variants (YOLOv5/v7/SCRFD/Pose)
│   ├── classification_postprocessor.py # EfficientNet, AlexNet, etc.
│   ├── depth_postprocessor.py          # FastDepth, MiDaS
│   ├── nanodet_postprocessor.py        # NanoDet
```


Copyright 2026 DEEPX All Rights Reserved.

```
│   ├── 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
    ├── 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`

Copyright 2026 DEEPX All Rights Reserved.

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
```

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

Copyright 2026 DEEPX All Rights Reserved.

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

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
```


Copyright 2026 DEEPX All Rights Reserved.

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`
`config/`	`model_config.hpp`	`model_config.py`	Loads `config.json` (input size, labels, thresholds)
`processors/`	45 header files	35 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

Copyright 2026 DEEPX All Rights Reserved.

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/`
`*_sync_cpp_postprocess.py` / `*_async_cpp_postprocess.py`	`src/postprocess/` via pybind11

Numerical Verification Framework

DX-APP includes an automated verification pipeline that validates model outputs after inference:

- **`scripts/validate_models.sh --numerical`** : runs all supported models through NPU inference and checks output correctness
- **`scripts/verify_inference_output.py`** : task-specific validators (14 types) that check bounding boxes, class IDs, confidence ranges, segmentation masks, depth maps, etc.
- **`scripts/inference_verify_rules.json`** : configurable thresholds per task type
- **`common/runner/verify_serialize.py`** : serializes postprocess results to JSON for comparison

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

Copyright 2026 DEEPX All Rights Reserved.

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/`

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

Contributor Rules

Adding New Examples

Copyright 2026 DEEPX All Rights Reserved.

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.

Shared Test Infrastructure (`tests/common/`)

Copyright 2026 DEEPX All Rights Reserved.

`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
- `scripts/validate_models.sh --numerical` additionally performs numerical verification using `verify_inference_output.py`
- `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 --numerical --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`
- `scripts/verify_inference_output.py`
- `tests/cpp_example/`
- `tests/python_example/`
- `docs/10_DX-APP_DX-Tool_Guide.md`

Copyright 2026 DEEPX All Rights Reserved.

8. RELEASE_NOTES

8.1 v3.1.0 / 2026-04-06

8.1.1 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)

8.1.2 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

Copyright 2026 DEEPX All Rights Reserved.

- Implemented https://sdk.deepx.ai manifest-based DX-ModelZoo auto-download system (`scripts/download_models.py`)
- Auto-download for models and videos — automatically invokes `setup_sample_models.sh` when model file is missing, videos via `setup_sample_videos.sh`
- Interactive mode for `scripts/run_examples.sh` — 6-stage menu (Language→Category→Filter→ExecMode→InputType→Options) when run without arguments, with case-insensitive keyword filtering
- `dx_tool.sh run` unified with `run_examples.sh` interactive mode
- Real-time performance table output during example execution
- `--show-log` option for Python examples — controls per-frame detailed log output

---

8.2 v3.0.4 / 2026-03-27

8.2.1 Changed

Test Infrastructure Restructuring

- **`tests/common/`** : extracted shared test constants (`constants.py`) and utilities (`utils.py`) from individual test files into a reusable module
- **Unified Visualization Tests**: consolidated per-task visualization test scripts into `tests/cpp_example/test_visualization.py` and `tests/python_example/test_visualization.py` — auto-discover all sync/async executables and scripts
- **Feature Test Suite**: added dedicated test modules for `--save` / `--save-dir` (`test_save_mode.py`), `--dump-tensors` (`test_dump_tensors.py`), `DXAPP_VERIFY` (`test_verify.py`), `--loop` (`test_multi_loop.py`), `SIGINT` / `SIGTERM` (`test_signal_handling.py`)
- Deleted 4 redundant root-level test scripts (`run_e2e_test.sh` , `run_visualization_test.sh` , `run_inference_test.sh` , `run_unit_test.sh`) — all functionality covered by `run_tc.sh`

Documentation Comprehensive Update

- **README.md**: added CLI Reference (full argument tables for C++/Python), Advanced Features section (signal handling, run directory, `DXAPP_VERIFY` , tensor dump, model config, version compatibility, headless mode), environment variables table, updated module counts and test tree
- **C++ Usage Guide** (`03`): added CLI Arguments table (12 flags), Advanced Features section, updated utility count (6→8)

Copyright 2026 DEEPX All Rights Reserved.

- **Python Usage Guide** (`05`): added CLI Arguments table (13 flags), Advanced Features section, updated module counts (processors 34→35 description, visualizers 9→10 description)
- **Example Source Structure** (`11`): updated C++ utility count (6→8), Python utility count (6→7), added complete test infrastructure section with 9 test categories, `tests/common/` documentation
- **C++ Test README**: added visualization tests, feature tests (5 types), test infrastructure section, updated coverage summary
- **Python Test README**: expanded to all 14 task directories, added `test_visualization.py`, `tests/common/` reference

8.2.2 Added

- `tests/common/constants.py` — shared test constants (paths, timeouts, patterns)
- `tests/common/utils.py` — shared test utilities (discovery, execution, validation)
- `tests/cpp_example/test_visualization.py` — unified C++ visualization result tests
- `tests/cpp_example/test_save_mode.py` — `--save` / `--save-dir` flag tests
- `tests/cpp_example/test_dump_tensors.py` — `--dump-tensors` output tests
- `tests/cpp_example/test_verify.py` — `DXAPP_VERIFY` environment variable tests
- `tests/cpp_example/test_multi_loop.py` — `--loop` repeated execution tests
- `tests/cpp_example/test_signal_handling.py` — SIGINT/SIGTERM graceful shutdown tests
- `tests/python_example/test_visualization.py` — unified Python visualization result tests

8.2.3 Removed

- `run_e2e_test.sh` — replaced by `run_tc.sh --cpp --e2e`
- `run_visualization_test.sh` — replaced by `run_tc.sh --cpp --viz`
- `run_inference_test.sh` — replaced by `run_tc.sh --python`
- `run_unit_test.sh` — replaced by `run_tc.sh --unit`

8.3 v3.0.3 / 2026-03-13

8.3.1 Changed

- Expanded model support across multiple AI task categories

Copyright 2026 DEEPX All Rights Reserved.

8.3.2 Fixed

Post-Processing Bug Fixes (16 models)

- **Segmentation** (bisenetv1, bisenetv2, deeplabv3plusmobilenet): Fixed pre-argmaxed NPU output (uint16 `[1,H,W]`) being misinterpreted as logits — added heuristic detection for integer dtype / shape[0]==1
- **NanoDet** (nanodet_repvgg, nanodet_repvgga1): Fixed degenerate bounding boxes (y2==y1) caused by clipping to image boundary — added zero-area box filter
- **FastDepth** (fastdepth_1): Fixed `DepthResult` not handled by verify_serialize.py — added DepthResult serialization handler
- **YOLOv5Pose PPU** (yolov5pose_ppu): Fixed raw logit keypoint confidence (negative values) — applied sigmoid activation
- **YOLOX** (yoloxs, yoloxtiny, yolox_l_leaky, yolox_s_leaky, yolox_s_wide_leaky): Fixed zero bounding boxes from raw logit coordinates — implemented standalone `YOLOXPostprocessor` with grid decode (`cx=(cx_raw+grid_x)*stride`)
- **YOLOv7 Face** (yolov7_face, yolov7s_face, yolov7_w6_face, yolov7_w6_tta_face): Fixed confidence > 1.0 from misread column layout — added auto-dispatch by output column count (16-col vs 21-col) with sigmoid on raw class logit

8.3.3 Added

Shared Runtime Layer (`common/`)

- **C++ (`src/cpp_example/common/`)**: Base interfaces (`IFactory`, `IProcessor`, `IVisualizer`, `IInputSource`), 45 processors, 24 task-specific sync/async runner pairs, 12 visualizers, input source abstraction, config loader, utility
- **Python (`src/python_example/common/`)**: Base interfaces (`IFactory`, `IProcessor`, `IVisualizer`, `IInputSource`), 35 processors, generic `SyncRunner` / `AsyncRunner`, 10 visualizers, input source abstraction, `ModelConfig` loader, utility
- Both languages share the same 7-module architecture (`base/`, `config/`, `processors/`, `runner/`, `inputs/`, `visualizers/`, `utility/`) and factory-based delegation pattern

Model Registry System

- **`config/model_registry.json`**: centralized registry of 280 models with per-model metadata (task, postprocessor, input dimensions, thresholds)
- **`scripts/add_model.sh`**: registry-driven auto-generation of factory files, config.json, and entry-point scripts (4 variants per model)

Copyright 2026 DEEPX All Rights Reserved.

Numerical Verification Framework

- `scripts/verify_inference_output.py` : 14 task-specific validators for bounding boxes, confidence ranges, class IDs, keypoints, segmentation masks, depth maps, embeddings
- `scripts/inference_verify_rules.json` : configurable thresholds per task type
- `common/runner/verify_serialize.py` : result-to-JSON serialization for automated comparison
- `scripts/validate_models.sh --numerical` : full-pipeline NPU verification for all supported models

New Model Families

- DAMOYOLO (5 variants), NanoDet (2), CenterNet, SSD MobileNet V1/V2, YOLOv3, YOLOv6
- FastDepth, MiDaS (depth estimation)
- CLIP, ArcFace (embedding)
- DnCNN, Zero-DCE (image denoising/enhancement)
- ESPCN (super resolution)
- Hand Landmark (2 variants)
- BiSeNet V1/V2, SegFormer (semantic segmentation)
- YOLOv7Face (4 variants), RetinaFace (face detection)
- YOLOv26 OBB, Seg, Pose, Cls variants

CI/CD Integration

- `python-test.yml` : automated unit/CLI/integration tests on PR (Python, no NPU required)
- `npu-model-verify.yml` : NPU numerical verification pipeline (self-hosted runner with NPU hardware)

8.4 v3.0.2 / 2026-02-10

8.4.1 Changed

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

Copyright 2026 DEEPX All Rights Reserved.

8.4.2 Fixed

- Remove experimental filesystem includes and update float literals in example cpp files for build error on windows
- Refactor apply_argmax to reduce nesting and fix gcovr warnings

8.4.3 Added

- Added vcpkg installation script for windows build.

8.5 v3.0.1 / 2026-02-05

8.5.1 Fixed

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

8.5.2 Added

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

---

8.6 v3.0.0 / 2026-01-02

8.6.1 Changed

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

Copyright 2026 DEEPX All Rights Reserved.

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

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

Model Support Expansion

- **YOLOv10, YOLOv11, YOLOv12** examples added
- **YOLOv8 Segmentation** (YOLOv8-seg) support
- **DeepLabv3** segmentation model support
- **PPU (Post-Processing Unit)** module integration:
  : YOLOv5, YOLOv7 PPU version support
  : SCRFD PPU version support
  : Both Python and C++ examples provided

Documentation Improvements

- Newly written example guides and installation guides
- Added detailed usage examples and parameter descriptions for each model

8.6.2 Fixed

Code Quality Improvements

- Added try-catch error handling to all projects

Copyright 2026 DEEPX All Rights Reserved.

- 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

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

8.6.3 Added

Post-processing Library (dx_postprocess)

- **Pybind11-based Python binding**:
  : Provides Python binding for C++ post-processing functions
  : Automatically installs to current Python execution environment

Multi-channel Processing Support

- C++ **YOLOv5s multi-channel processing**: Multi-channel support using frame provider
- Added multi-input source examples
- Enhanced multi-model image inference using preprocessing threads

Test Infrastructure Construction

- **Pytest-based integrated test system**:
  : Automated testing for all Python examples
  : Achieved code coverage of 93.65% or higher
  : E2E(End-to-End) test framework
  : Includes all model tests for classification, object detection, segmentation, and pose estimation
- Added `.coveragerc` file for code coverage configuration
- Support for display mode and E2E mode testing

Copyright 2026 DEEPX All Rights Reserved.

New Examples and Features

- **Classification Models**:
  : EfficientNet example integration
  : ImageNet classification examples (synchronous/asynchronous)
- **Object Detection Models**:
  : 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

8.6.4 Removed or Replaced

Legacy Demo Removal

- Complete removal of `demos/classification/`
- Complete removal of `demos/object_detection/`
- Complete removal of `demos/segmentation/`
- Complete removal of `demos/pose_estimation/`
- Complete removal of `demos/face_recognition/`
- Removal of `demos/denoiser/`
- Removal of `demos/dncnn_yolo/`
- Removal of `demos/object_det_and_seg/`
- Removal of `demos/noiseVideoMaker/`

Legacy Configuration File Removal

- Complete removal of JSON configuration files in `example/` directory

Copyright 2026 DEEPX All Rights Reserved.

- Removal of `example/dx_postprocess/` JSON files
- Removal of Debian package related files (`debian/`)
- Removal of Docker build files (`docker/Dockerfile.app.build`)

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

8.6.5 Migration Guide

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.

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`)

8.6.6 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)` .

Copyright 2026 DEEPX All Rights Reserved.

8.7 v2.1.0 / 2025-11-28

8.7.1 Changed

- Enhance build script documentation and usage instructions
- Update 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.

8.7.2 Fixed

- Fix Windows MSBuild compilation warnings by replacing implicit type casts with explicit static_cast
- Improve tensor allocation in imagenet classification example
- Update numBoxes calculation based on post-processing type in LayerReorder
- Rename YOLO post-processing types and add aliasing for backward compatibility
- Add 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
- Enhance user input handling for run_demo selection with re-prompt loops (invalid input re-asks instead of timing out)

8.7.3 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)

Copyright 2026 DEEPX All Rights Reserved.

- Add 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

8.7.4 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).

8.8 v2.0.0 / 2025-08-14

8.8.1 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
- Update documentation and resources
- YoloPostProcess now filters and selects the correct tensor by output_name when USE_ORT=ON
- Command-line help messages in various demos have been improved to clearly mark required parameters.
- Replaced YOLOv5s-1 example json to YOLOv5s-6 json configuration file has been added for object detection.
- Documentation has been updated to add Python requirements and modified some images.
- feat: add OS and architecture checks in build script & update CPU specifications in documentation

Copyright 2026 DEEPX All Rights Reserved.

8.8.2 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
- fix typo error in framebuffer info file path (yolo_multi app)
- Improve error messages for output tensor size mismatch and missing in Yolo post processing
- Rename output tensors in json config 'yolov5s6_example.json'

8.8.3 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

8.9 v1.11.0 / 2025-07-24

8.9.1 Changed

- feat: enhance --clean option in build script for pybind artifacts

Copyright 2026 DEEPX All Rights Reserved.

- 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

8.9.2 Fixed

- feat: Improve error message readability in install, build scripts
  : Apply color to error messages
  : Reorder message output to display errors before help messages
- Update tensor index assignment in Yolo layer reordering
- fix: resolve dx_postprocess Python lib build error and improve error handling

---

8.10 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.