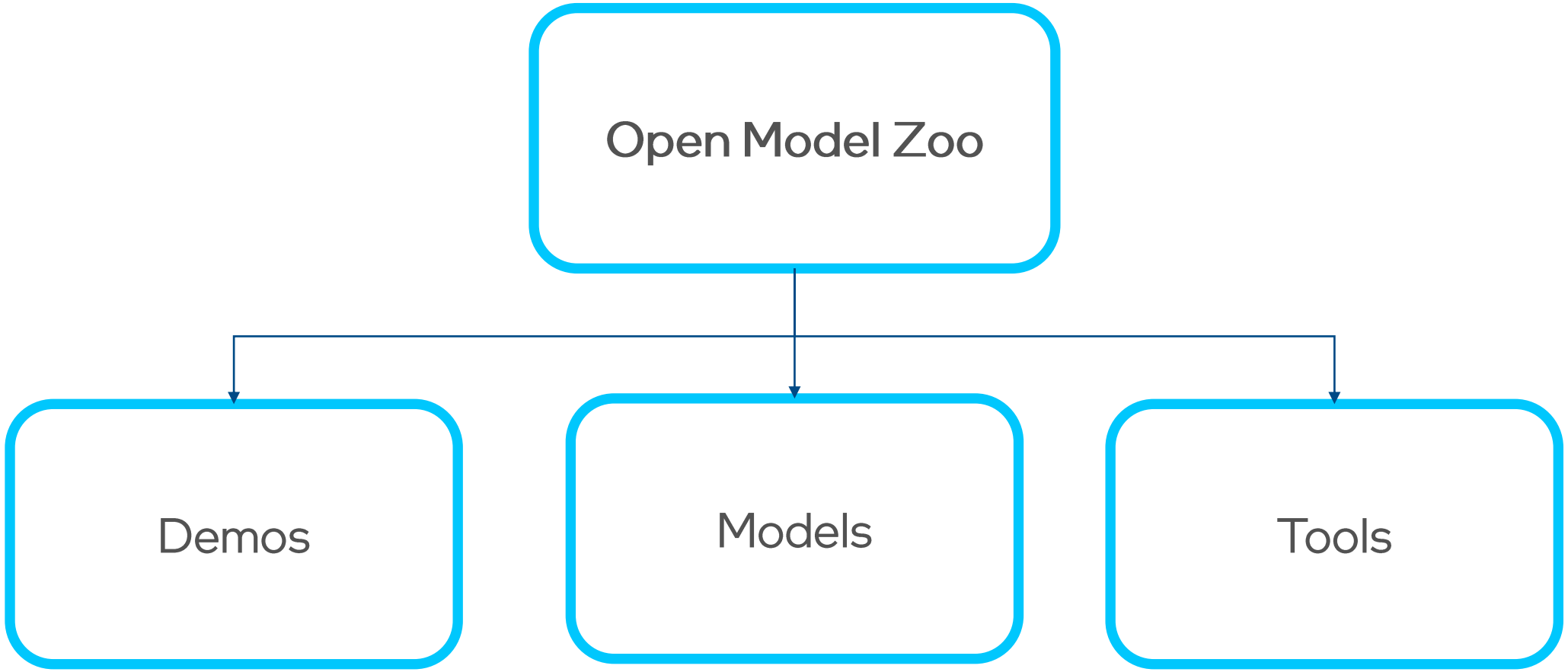
OpenModelZoo overview

Fedor Zharinov, ICV
Deep Learning Software Engineer
Eduard Zamaliev, ICV
Deep Learning Software Engineer

The OpenVINO toolkit: Open Model Zoo



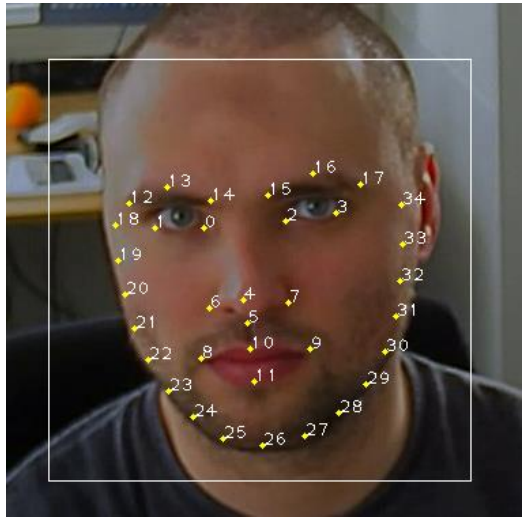
https://github.com/openvino/open_model_zoo

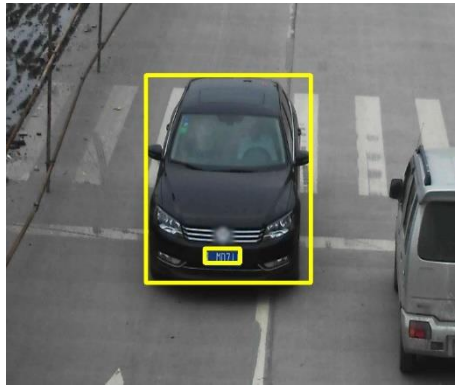
Open Model Zoo

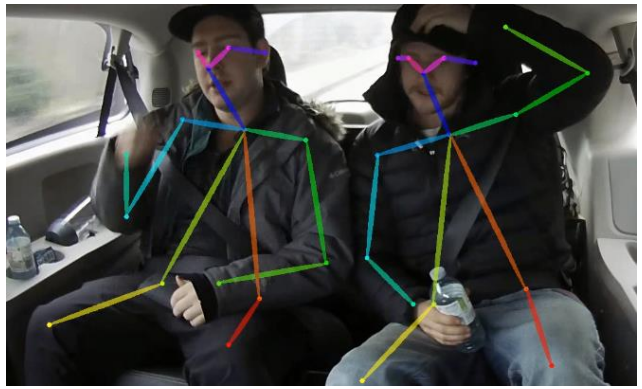
- Over 250 models, optimized and ready for inference
  - Models, developed, pretrained and finetuned by Intel`s data scientists
  - Most popular public models, supported by OpenVINO and validated by Intel`s engineers
  - Optimized models (quantized, binary and sparsed models)
- Over 40 demo applications
  - C\C++ and Python
  - Simple and complex cases demos
  - Use OpenCV and G-API
- Model API
  - Model wrappers
  - Async pipeline
- Tools for downloading and auto-converting models
- Accuracy Checkers for validation

Open Model Zoo

- Computer vision
  - Image classification
  - Object detection (common objects, faces, license plates, etc.)
  - Instance segmentation
  - Semantic segmentation
  - Keypoints detections (face landmarks, pose keypoints, etc.)

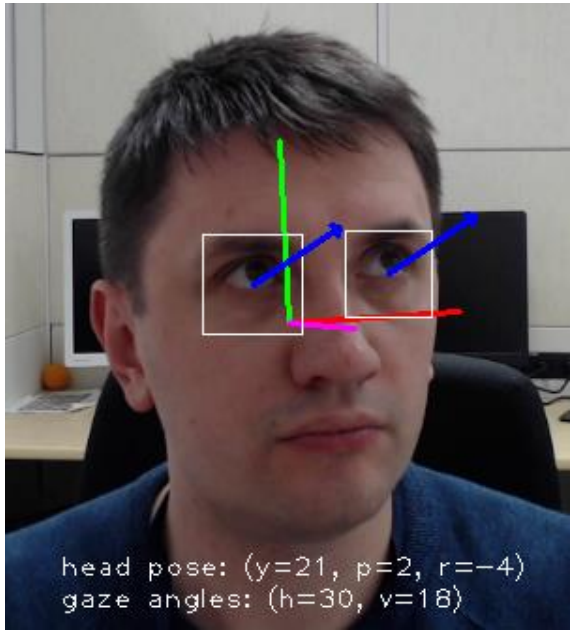






Open Model Zoo

- Attributes detection
  - Head and gaze direction
  - Open/close eyes
  - Mask detection
  - Emotions recognition
  - Gender recognition
  - Age recognition
- Gesture recognition
  - Common sign language
  - American sign language

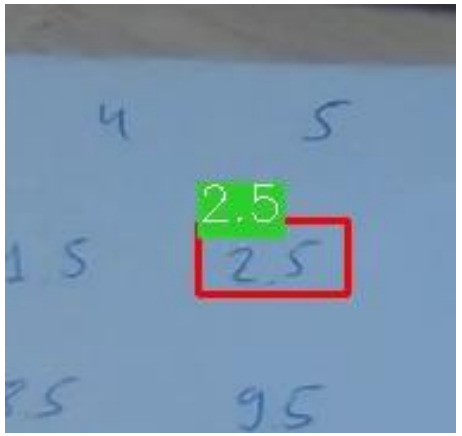


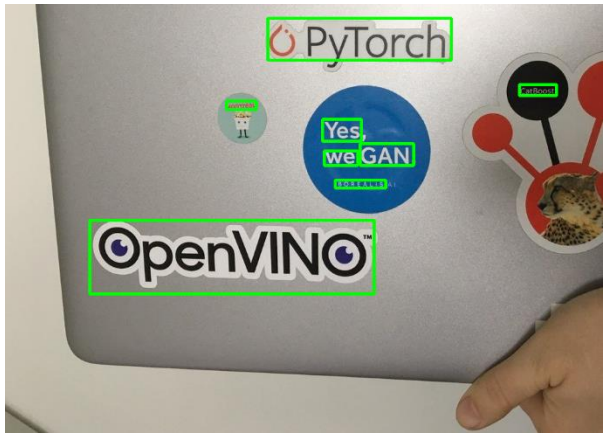
Open Model Zoo

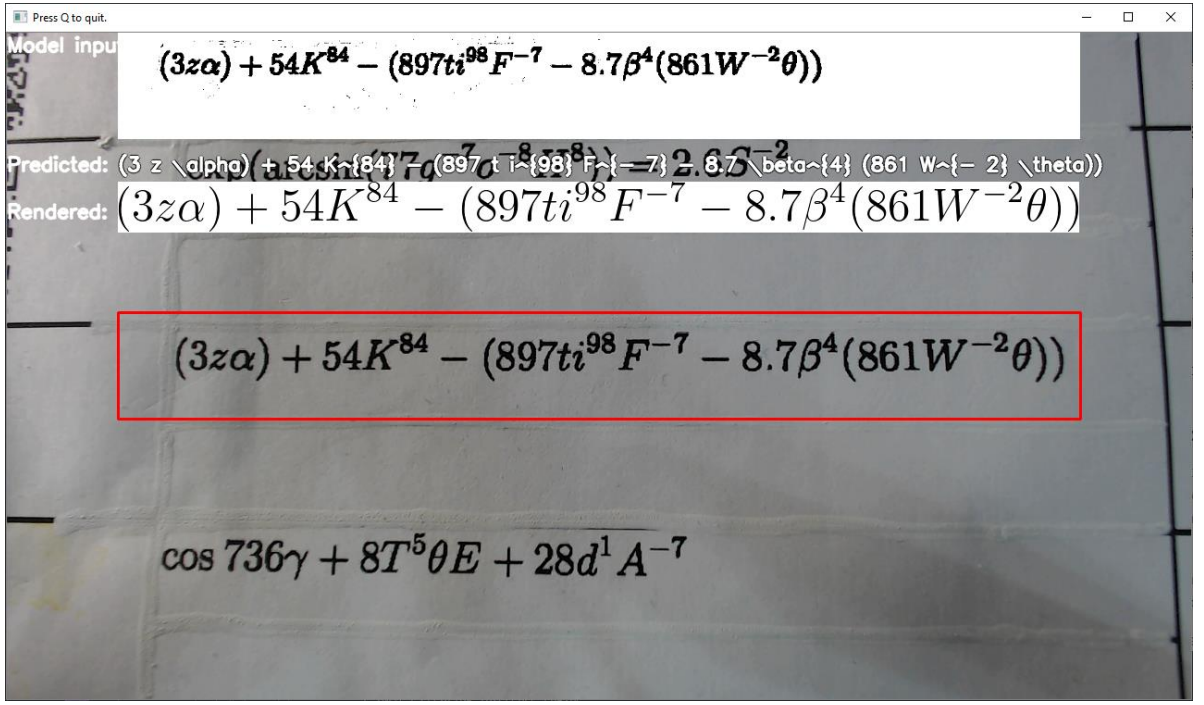
- Image generation
  - Super-resolution
  - Deblur
  - Colorization
  - Inpainting

Open Model Zoo: models

- Text manipulations
  - Text detection and recognition
  - Formula recognition
  - Machine translation
  - Question answering

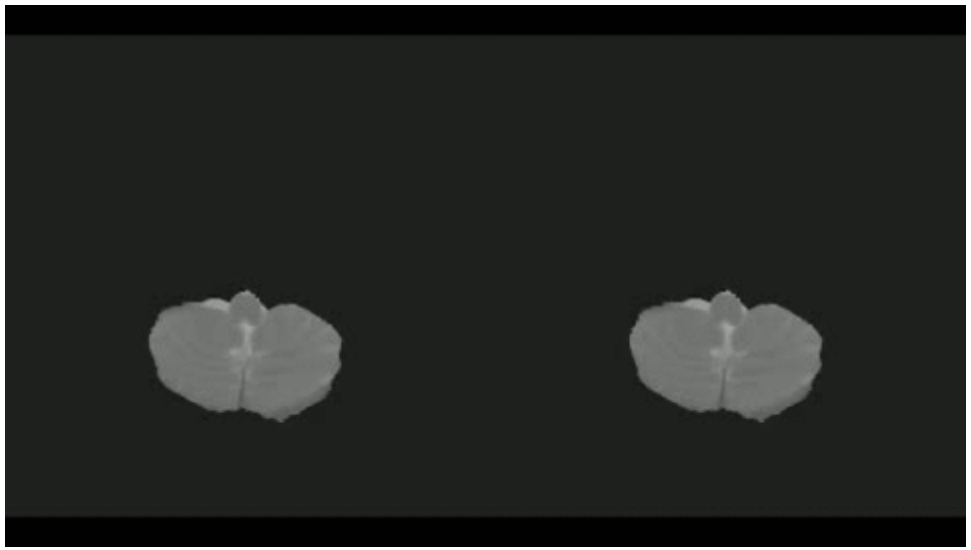






Open Model Zoo

- Brain tumor 3D segmentation
- Mono depth estimation
- Sound recognition
- Speech recognition
- Speech generation
- Action recognition
  - Common actions (sitting, eating, running, etc)
  - Sign language recognition
- Other cases:
  - Social distance detection



OMZ Repository Structure

- Default Location:
  - C:\Program Files (x86)\IntelSWTools\openvino_2021\deployment_tools\open_model_zoo
  - /opt/intel/openvino_2021/deployment_tools/open_model_zoo
  - or wherever you've cloned OMZ repository (https://github.com/openvinotoolkit/open_model_zoo)
- Structure:
  - **data** – datasets links, labels, etc.
  - **demos** – demos source code and building scripts.
  - **models** – model definitions and links. No pretrained model files are stored here !
  - **tools** :
    - **accuracy_checker** – tool for checking models accuracy
    - **downloader** - tools for downloading and converting pretrained models.

How to get models ?

- Install prerequisites:
  - Run this in tools/downloader/ (use pip3 for Linux or pip for windows):

    pip3 install -r requirements.in -r requirements-caffe2.in -r requirements-pytorch.in -r requirements-tensorflow.in

- Find proper CNN model name
  - From demo – look at **models.lst** file
  - From full models list:
    - models/public/index.md or models/intel/index.md
    - downloader –print_all

How to get models ?

- Download model (tools/downloader):
  - Run downloader:
    - python downloader.py --name *alexnet*
    - python downloader.py --name *resnet**
  - Model will be downloaded to tools/downloader/intel or tools/downloader/public
  - If you want to download to other location – use **-o** command line option
- Convert model
  - Intel models (tools/downloader/intel) are not required to be converted
  - Public models (tools/downloader/public) have to be converted to IE intermediate representation (IR) format:
    - python converter.py --name *alexnet*
  - As result of conversion, folders FP16, FP32, etc will be created in model location, they should contain .xml and .bin files
  - If you have troubles with finding model optimizer (error message pops up) - use **-mo** option with path to model optimizer

How to run demo

- Install/prepare prerequisites
  - Set up environment:
    - C:\Program Files (x86)\IntelSWTools\openvino_2021\bin\setupvars.bat
    - source /opt/intel/openvino_2021/bin/setupvars.sh
  - For python demos:
    - pip3 install -r demos/requirements.txt
    - some demos contains additional requirements.txt files
- Build demo (for C++ demos)
    - demos\build_demos_msvc.bat (builds to %USERPROFILE%\Documents\Intel\OpenVINO\omz_demos_build)
    - demos/build_demos.sh (builds to ~/omz_demos_build/intel64/Release)
    - … or you may use cmake and demos/CMakeLists.txt to generate project/make files and then build project manually
- Run demos
  - For every demo have to provide at least path to model (usually -m) and input data (usually -i).
  - For information about all command line options run demo with -h command line key.
  - Examples:
    - object_detection_demo -at centernet -loop -i faces.jpg -m ~/open_model_zoo/tools/downloader/public/ctdet_coco_dlav0_384/FP16/ctdet_coco_dlav0_384.xml
    - python3 object_detection_demo.py -at centernet --loop -i faces.jpg -m ~/open_model_zoo/tools/downloader/public/ctdet_coco_dlav0_384/FP16/ctdet_coco_dlav0_384.xml

Demo pipeline

1. Prepare network
2. Read image
3. Preprocess image
4. Run network (in asynchronous mode preferably)
5. Postprocess result
6. Show/save result
7. Return to 2.

Demo pipeline : conclusion

- Most demos/user applications have similar implementation approach for models of the same tasks class:
  - Object detection
    - SSD*
    - YOLO*
    - RetinaFace*
    - CenterNet*
    - FaceBoxes
  - Human pose estimation
    - OpenPose*
    - EfficientHRNet*

- But due to the pre-/postprocessing differences those models require much work while integrating them into applications

… so, we need some easy-to-use universal solution…

modelAPI – one framework to rule them all

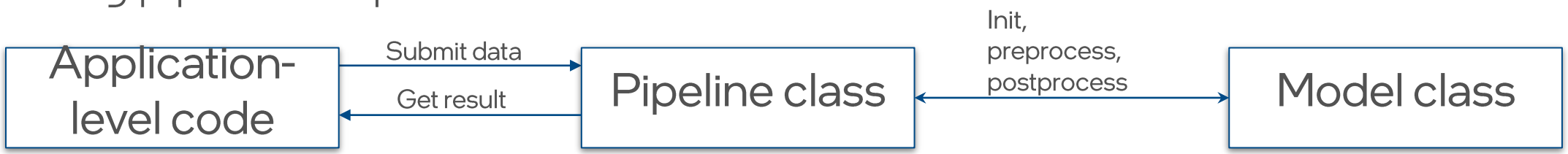
- Implements pre- and postprocessing algorithms as separate classes (one class - one model architecture).
- Implements common code for IE requests processing inside the “pipeline” class(es).
- Combines model and pipeline classes into separate library.

Model classes hide specific code inside and work as black-box:

Application feeds model class with input data.
Model returns post-processed output data in user-friendly form.

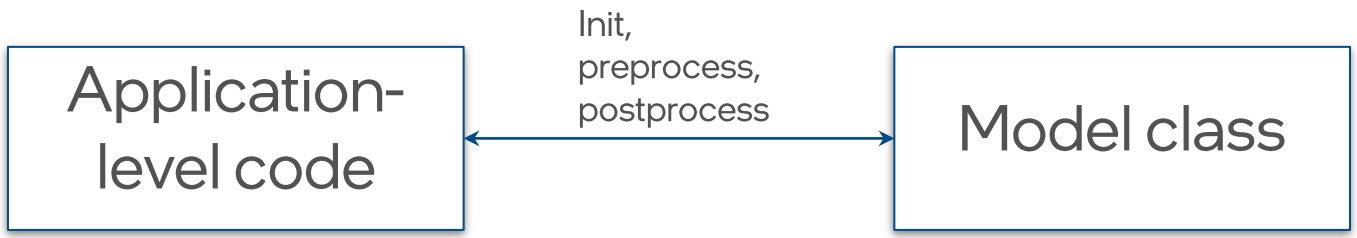
Suggested solution: use-case scenarios

Using pipeline helper class

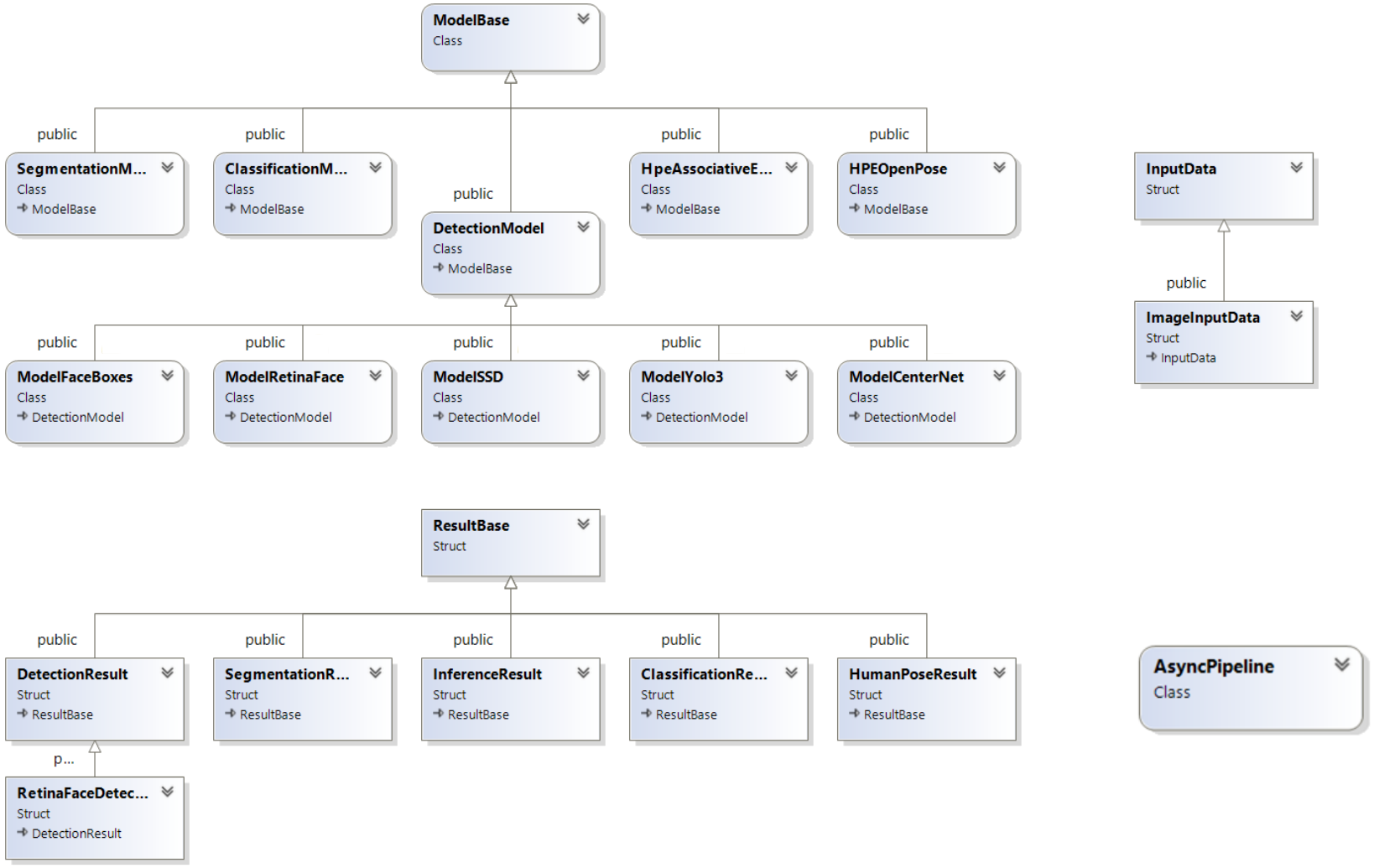


... or ...

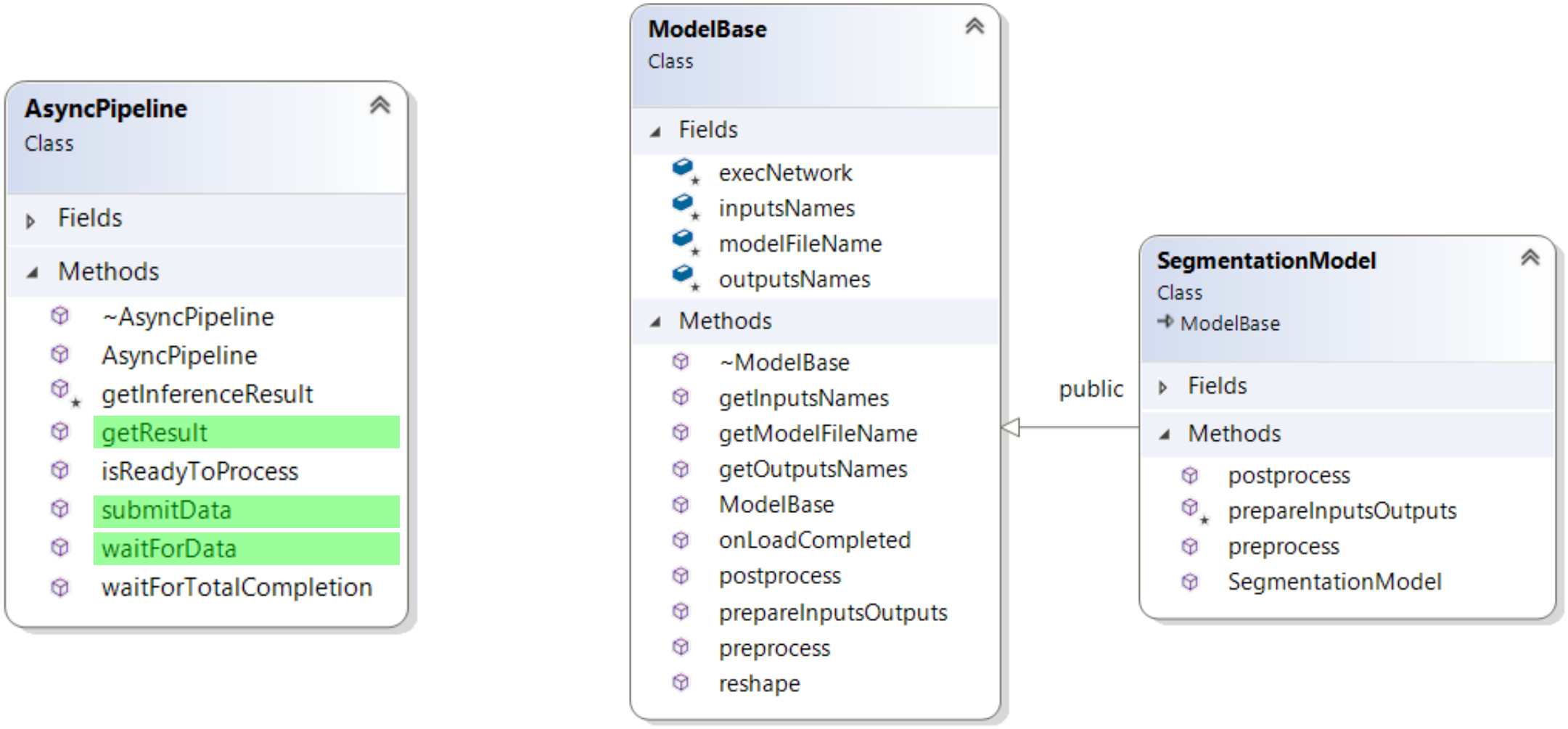
Calling model wrapper class directly



C++ implementation



C++ implementation



C++ implementation: application-level code

1. Submit data for processing using **submitImage()** or **submitRequest()** methods

2. Wait until output data is ready or another free data submission request becomes available

3. Get result and do painting

There's internal frames counter inside pipeline, results are buffered inside and returned in correct order, even if some frames are processed with different speed during parallel execution

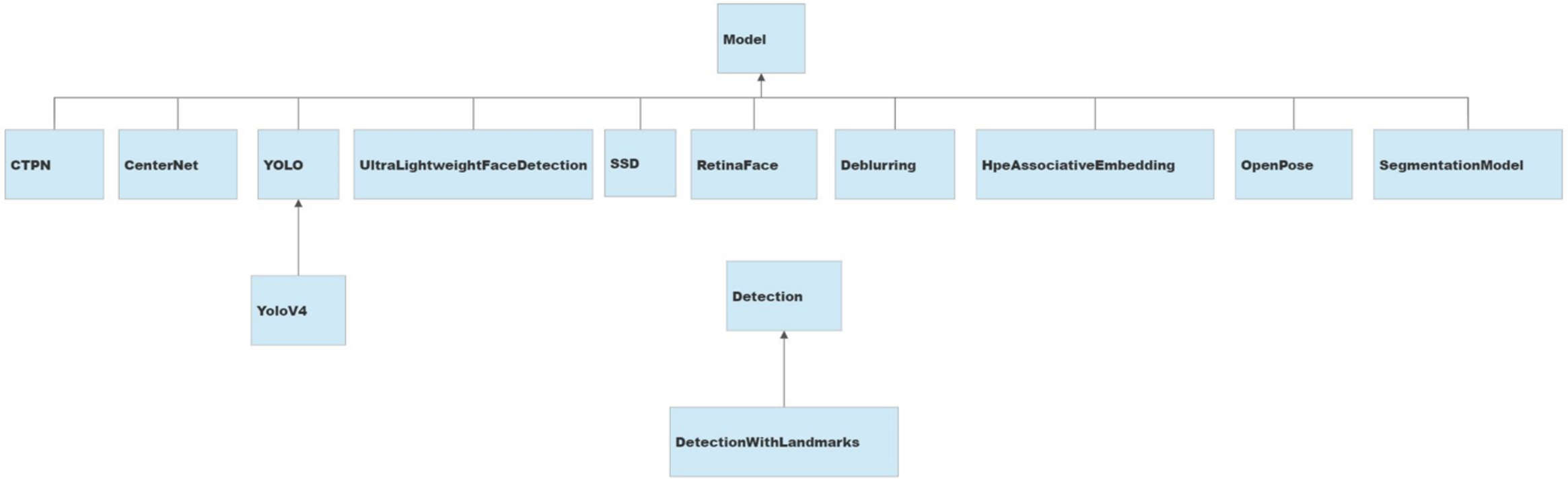
Check out segmentation_demo for complete code

```
// Simplified user code (without return status checks)
AsyncPipeline pipeline(
std::make_unique<SegmentationModel>("model.xml", true),
                                        config, core);

while (doProcessing) {
    if (pipeline.isReadyToProcess()) {
        auto st = std::chrono::steady_clock::now();

        cap >> curr_frame;
        pipeline.submitData(ImageInputData(curr_frame,
          std::make_shared<ImageMetaData>(curr_frame, st));
    }

    pipeline.waitForData();

    auto result = pipeline.getResult();

    if(result){
        paintResult(result);
    }
}
```

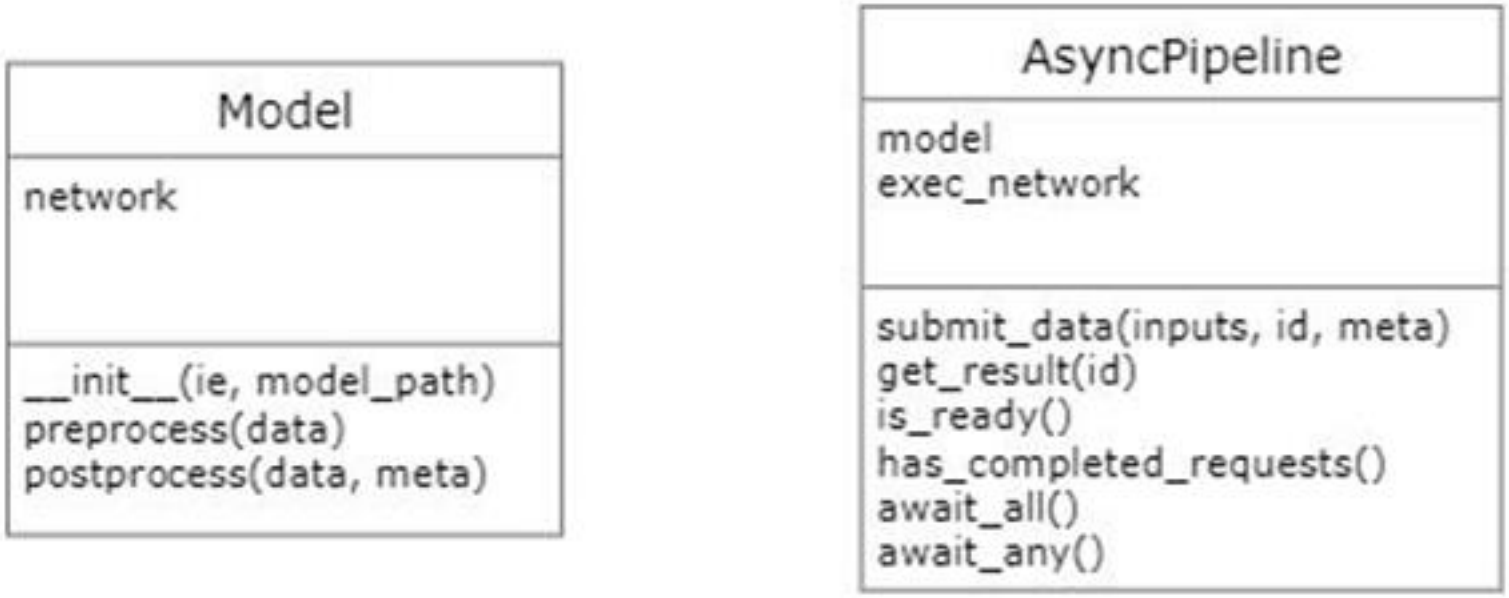
C++ implementation: pure model usage

```
SegmentationModel model("modelName.xml", false);
auto execNet = model.loadExecutableNetwork(config, core);
auto req = std::make_shared<InferenceEngine::InferRequest>(execNet.CreateInferRequest());
while (keepRunning) {
     curr_frame = cap->read();

    model.preprocess(ImageInputData(curr_frame), req);
    req->Infer();

    InferenceResult res;
    res.internalModelData = std::make_shared<InternalImageModelData>(
        curr_frame.cols, curr_frame.rows);
    res.metaData = std::make_shared<ImageMetaData>(curr_frame,std::chrono::steady_clock::now());
    for (const auto& outName : model.getOutputsNames()) {
        auto blob = InferenceEngine::as<InferenceEngine::TBlob<int>>(req->GetBlob(outName));
        res.outputsData.emplace(outName, blob);
    }

    auto result = model.postprocess(res);
     paintResult(result);
}
```

Python implementation



Python implementation



Python implementation: application-level code

1. Submit data for processing using **submitData()** method
2. Wait until output data is ready or another free data submission request becomes available
3. Get result and do painting

```
model = get_model(...)
detector_pipeline = AsyncPipeline(...)
while True:
    if detector_pipeline.callback_exceptions:
        raise detector_pipeline.callback_exceptions[0]
    results =
detector_pipeline.get_result(next_frame_id_to_show)
    if results:
        ...
        continue
    if detector_pipeline.is_ready():
        detector_pipeline.submit_data(frame,
next_frame_id, meta)
        next_frame_id += 1
    else:
        detector_pipeline.await_any()
```

Demos supporting modelAPI

C++:

- Object detection demo (incorporates 5 different model architectures)
- Segmentation demo
- Classification demo
- Human pose estimation demo

- Python:
  - Object detection demo (incorporates 5 different model architectures)
  - Segmentation demo
  - Human pose estimation demo (expands model coverage)


intel