

Mastering Transformers: From Building Blocks to Real-World Applications

Deployment

Prof. Alptekin Temizel
Graduate School of Informatics, Middle East Technical University
13 Sept 2023



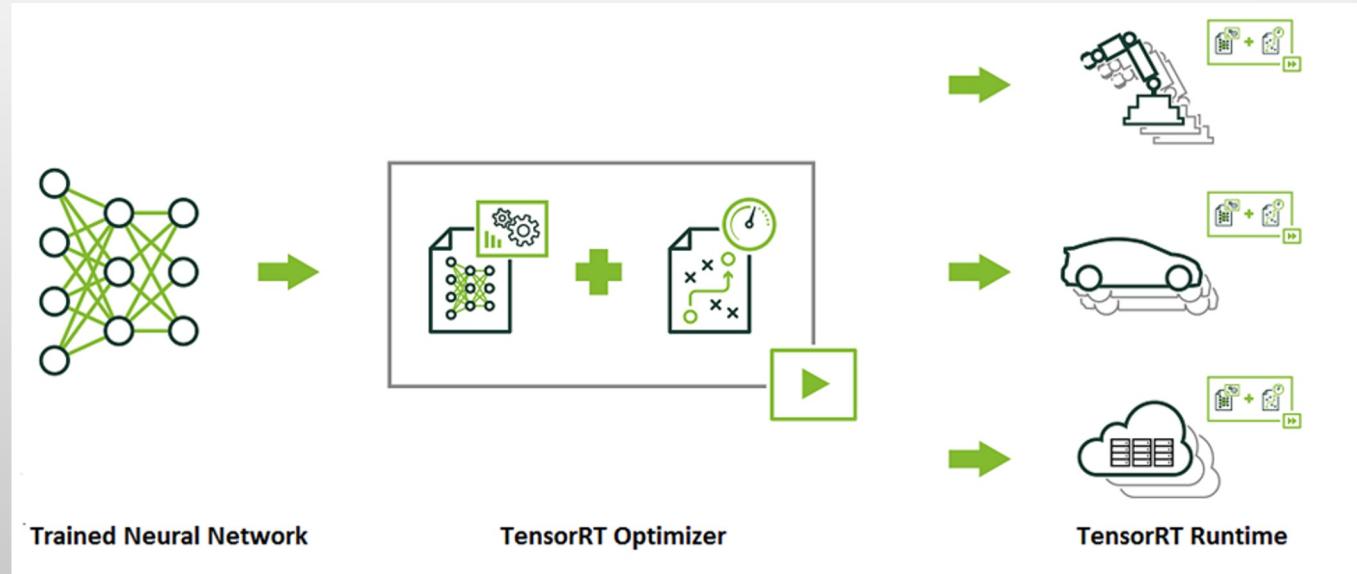
Deployment Frameworks

Training frameworks are not designed for efficient inference
Hence we need frameworks optimized for inference.

- NVIDIA TensorRT
- Intel OpenVINO
- Google MediaPipe
- Tencent TNN
- NVIDIA Triton Inference Server

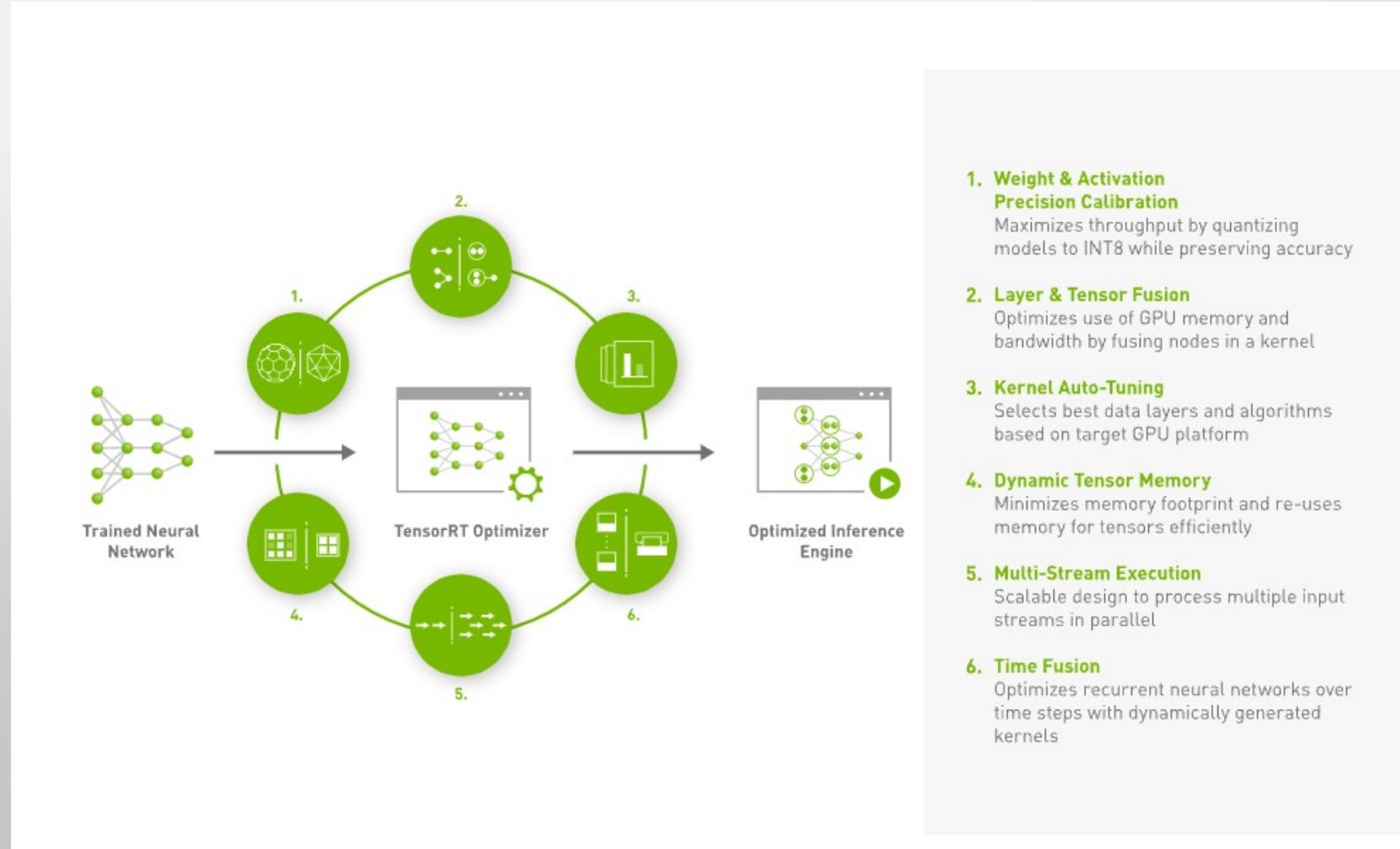
TensorRT

- Inference engine for production deployment of deep learning applications



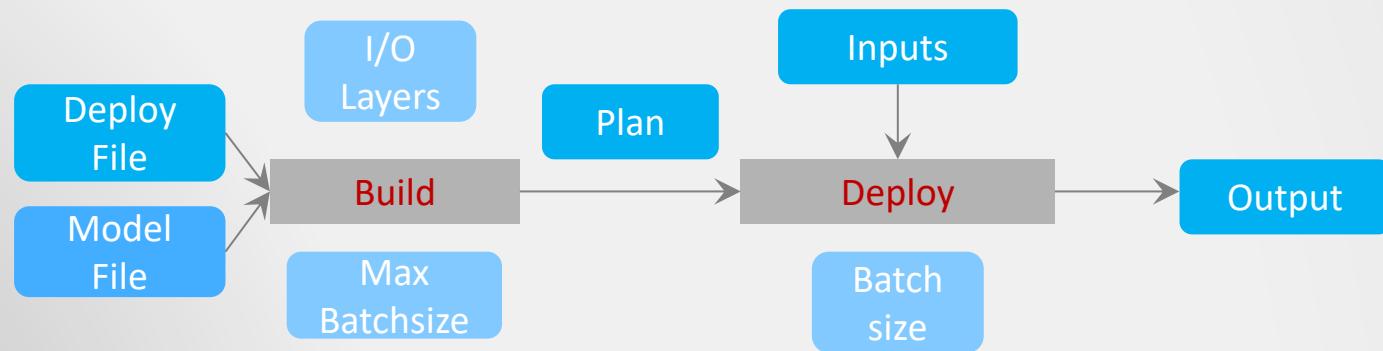
- Allows developers to focus on developing AI powered applications
 - TensorRT ensures optimal inference performance

TensorRT Optimizer



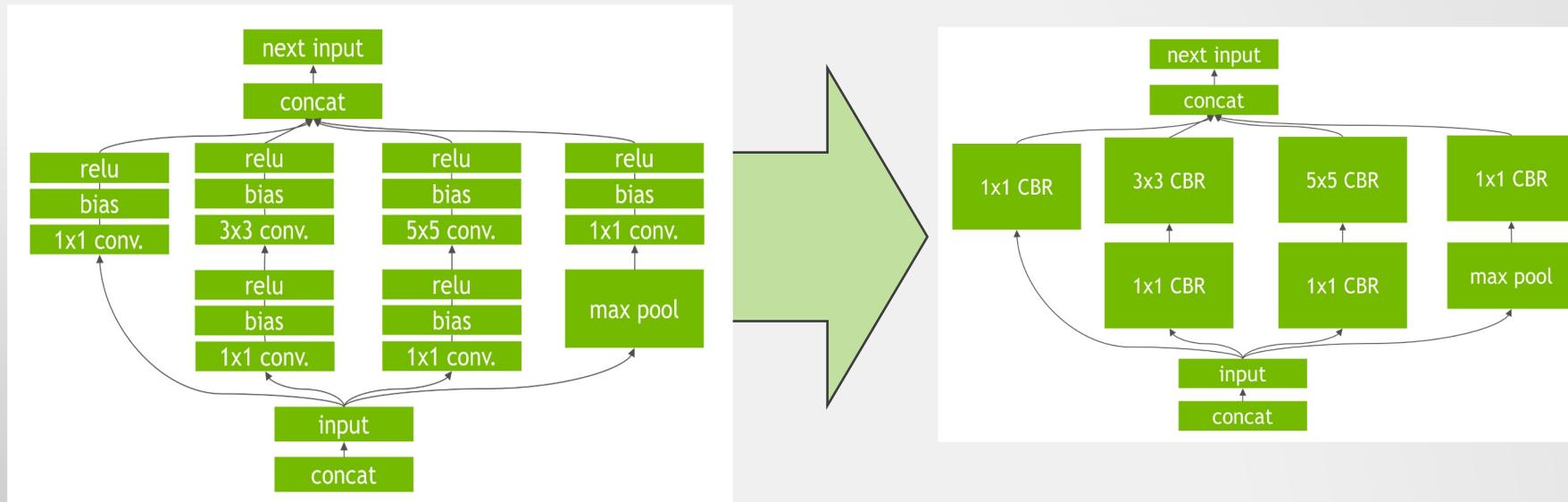
TensorRT RunTime

- Build: optimizations on the network configuration and generates an optimized plan for computing the forward pass
- Deploy: Forward and output the inference result



TensorRT Optimizer

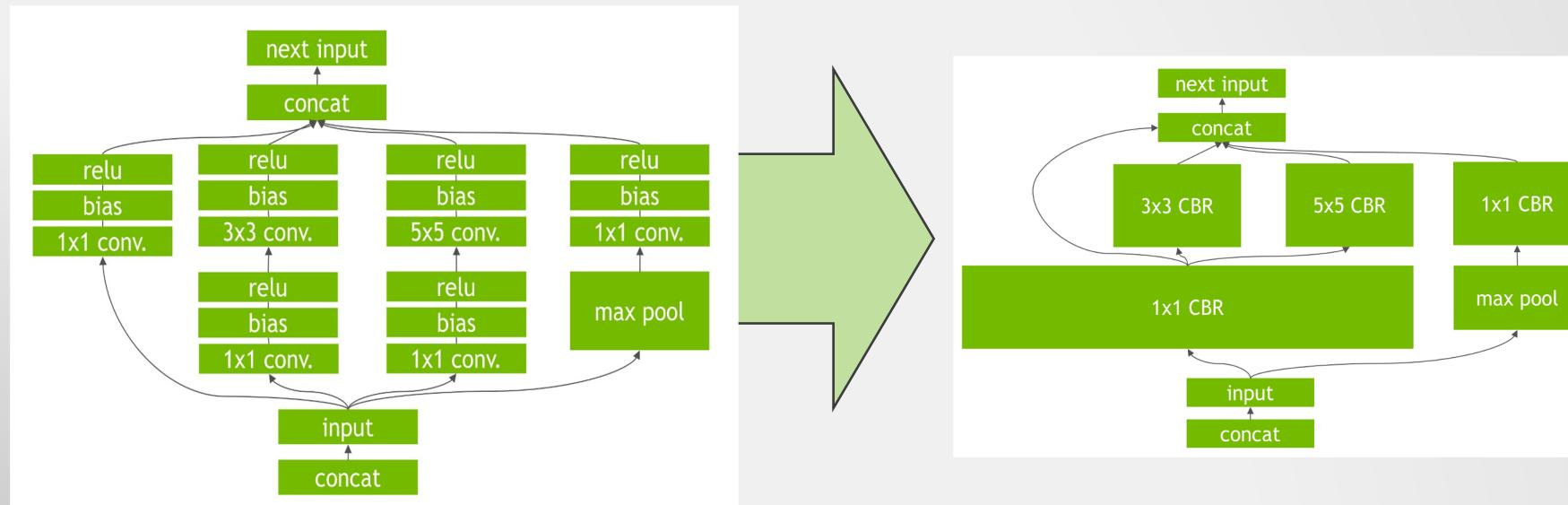
Vertical Layer Fusion



CBR = Convolution, Bias and ReLU

TensorRT Optimizer

Horizontal Layer Fusion (Layer Aggregation)

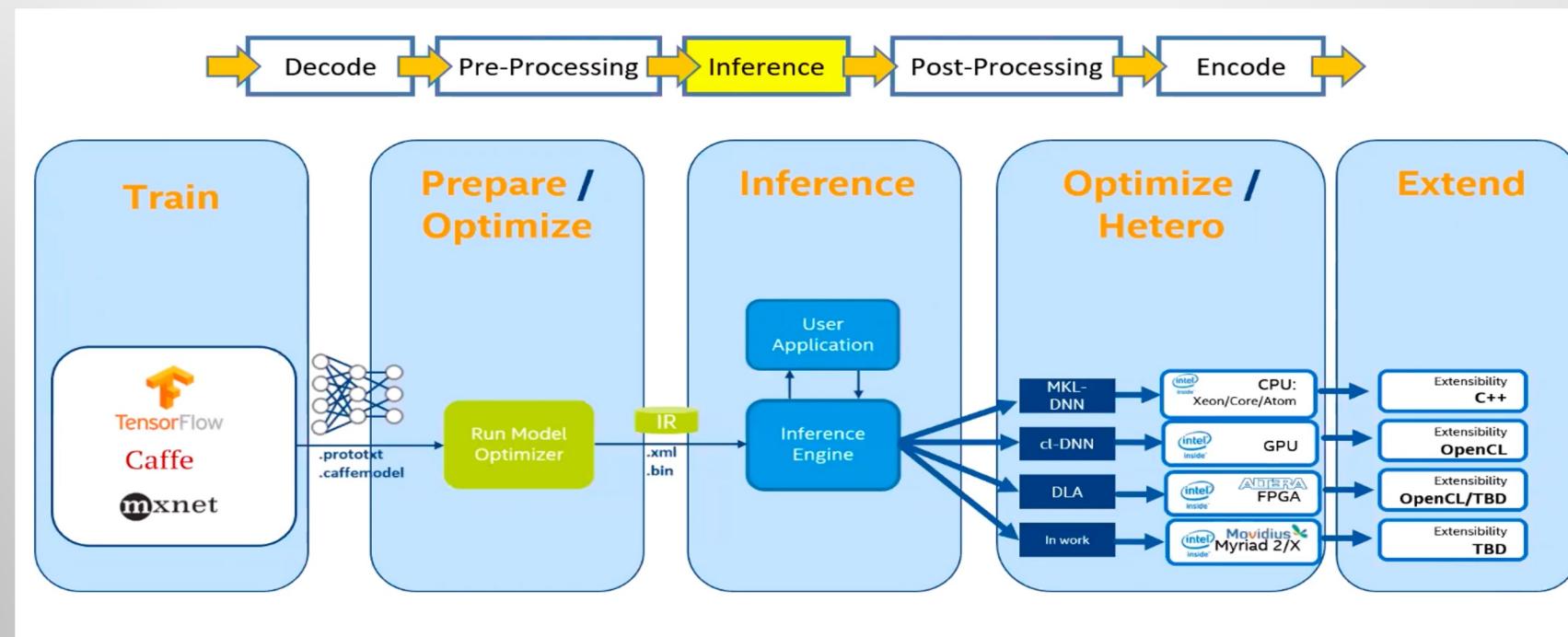


TensorRT RunTime

- No need to install and run a deep learning framework on the deployment hardware
- **Plan** is a runtime (serialized) object
 - smaller than the combination of model and weights
 - Ready for immediate use. Alternatively, state can be serialized and saved to disk or to an object store for distribution.
- Three files needed to deploy a classification neural network:
 - Network architecture file (deploy.prototxt)
 - Trained weights (net.caffemodel)
 - Label file to provide a name for each output class

OpenVINO:Open Visual Inference and Neural Network Optimization

- OpenVINO is a toolkit developed by Intel for accelerated inference
- Allows acceleration on CPU, GPU, Intel® Movidius™ Neural Compute Stick and FPGA
- Supports various deep learning frameworks



Google MediaPipe

- MediaPipe is an open source cross-platform, customizable ML solution for live and streaming media.



The MediaPipe logo features a stylized icon composed of three vertical bars of increasing height, followed by the word "MediaPipe" in a sans-serif font.

 End-to-end acceleration Built-in fast ML inference and processing accelerated even on common hardware	 Build once, deploy anywhere Unified solution works across Android, iOS, desktop/cloud, web and IoT	 Free and open source Framework and solutions both under Apache 2.0, fully extensible and customizable
--	---	--

<https://mediapipe.dev/>

Tencent TNN

- TNN is an inference optimization framework
- Produces optimized C++ compilations for Adreno, Mali, Apple and Nvidia GPUs from an ONNX input
- OpenVINO, TensorRT and various other optimizations can be done on the same framework



<https://github.com/Tencent/TNN>

OpenMMLab Detection Toolbox

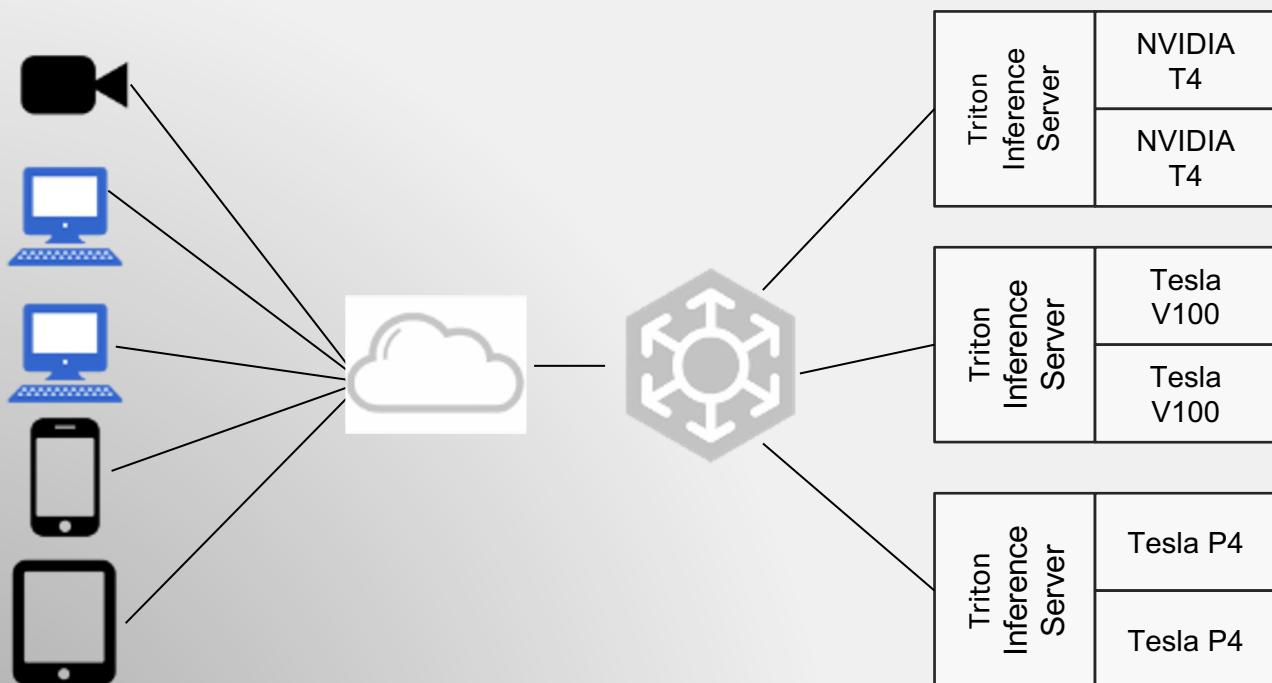
- All basic bounding box and mask operations run on GPUs.
- The training speed is faster than or comparable to other codebases, including Detectron2, maskrcnn-benchmark and SimpleDet
- It provides training support ONNX/TensorRT optimizations for various object detection and segmentation model



<https://github.com/open-mmlab/mm detection>

NVIDIA TRITON Inference Server

Production data center inference server



Aims to maximize real-time inference performance of GPUs

Deploy and manage multiple models per GPU per node

Scalable to heterogeneous GPUs and multi GPU nodes

NVIDIA TRITON Inference Server

Concurrent Model Execution

Multiple models (or multiple instances of same model) may execute on GPU simultaneously

CPU Model Inference Execution

Framework native models can execute inference requests on the CPU

Metrics

Utilization, count, memory, and latency

Custom Backend

Custom backend allows the user more flexibility by providing their own implementation of an execution engine through the use of a shared library

Model Ensemble

Pipeline of one or more models and the connection of input and output tensors between those models (can be used with custom backend)

Dynamic Batching

Inference requests can be batched up by the inference server to 1) the model-allowed maximum or 2) the user-defined latency SLA

Multiple Model Format Support

PyTorch JIT (.pt)
TensorFlow GraphDef/SavedModel
TensorFlow and TensorRT GraphDef
ONNX graph (ONNX Runtime)
TensorRT Plans
Caffe2 NetDef (ONNX import path)

CMake build

Build the inference server from source making it more portable to multiple OSes and removing the

Streaming API

Built-in support for audio streaming input e.g. for speech recognition



TensorRT

PYTORCH

ONNX

Chainer Microsoft CNTK

mxnet PYTORCH

NVIDIA TRITON Inference Server – Concurrent Model Execution

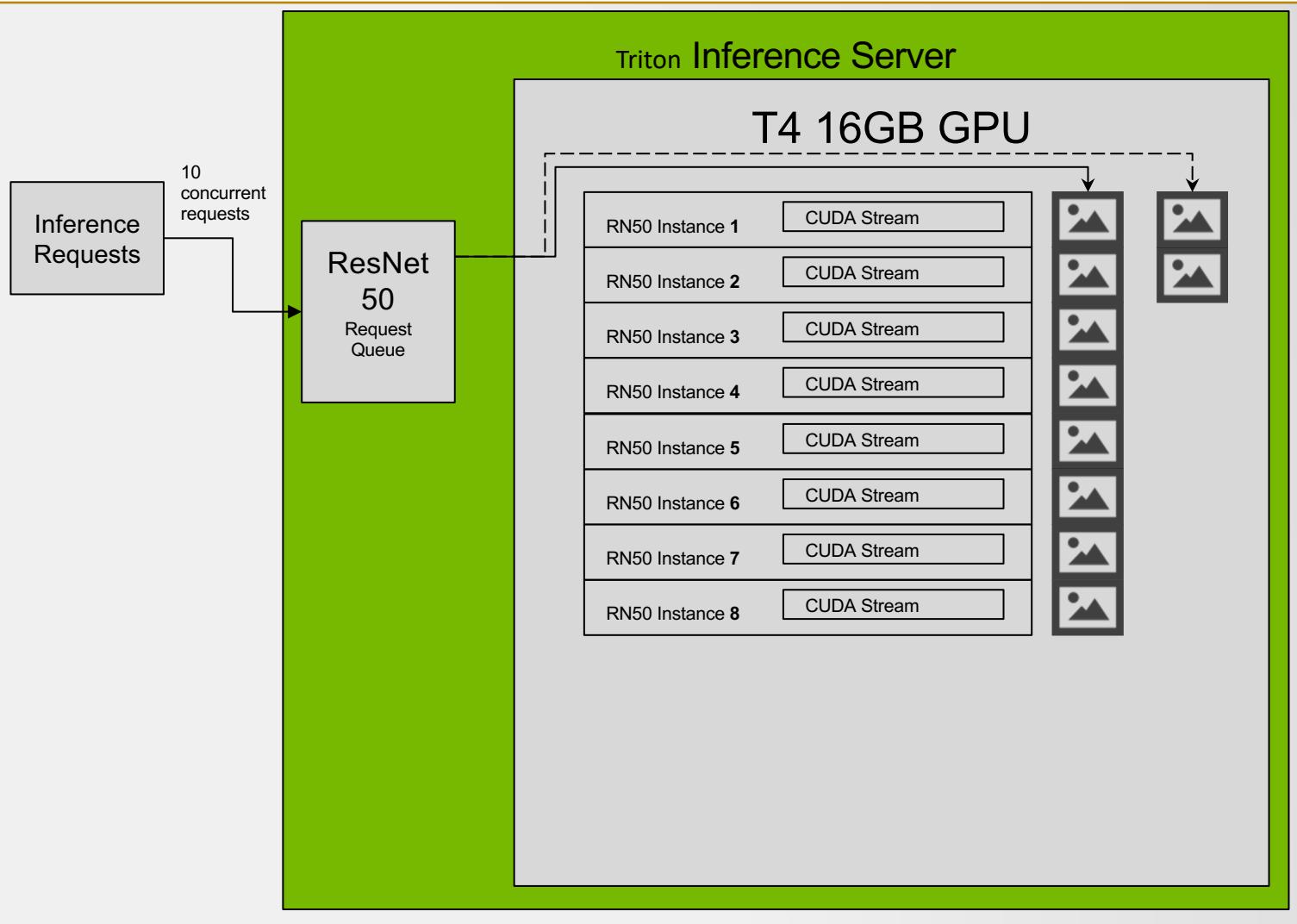
Better Performance and Improved GPU Utilization Through Multiple Model Concurrency

Common Scenario 1

One API using multiple copies of the same model on a GPU

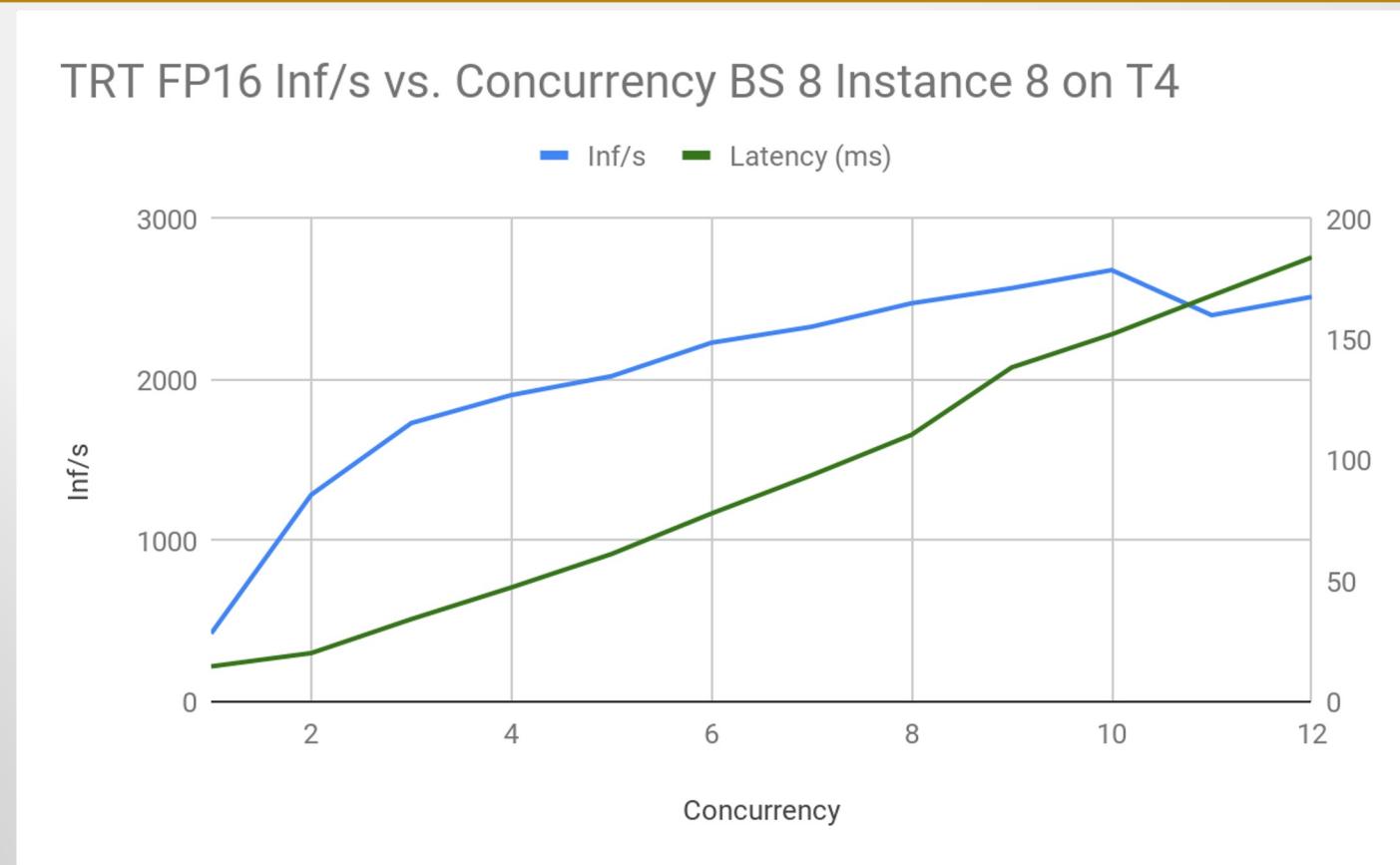
Example:

- 8 instances of TRT FP16 ResNet50 (each model takes 2 GB GPU memory) are loaded onto the GPU and can run concurrently on a 16GB T4 GPU.
- 10 concurrent inference requests happen: each model instance fulfills one request simultaneously and 2 are queued in the per-model scheduler queues in Triton Inference Server to execute after the 8 requests finish.



NVIDIA TRITON Inference Server – Concurrent Model Execution

Better Performance and Improved GPU Utilization Through Multiple Model Concurrency



With this configuration, 2680 inferences per second at 152 ms with batch size 8 on each inference server instance is achieved.

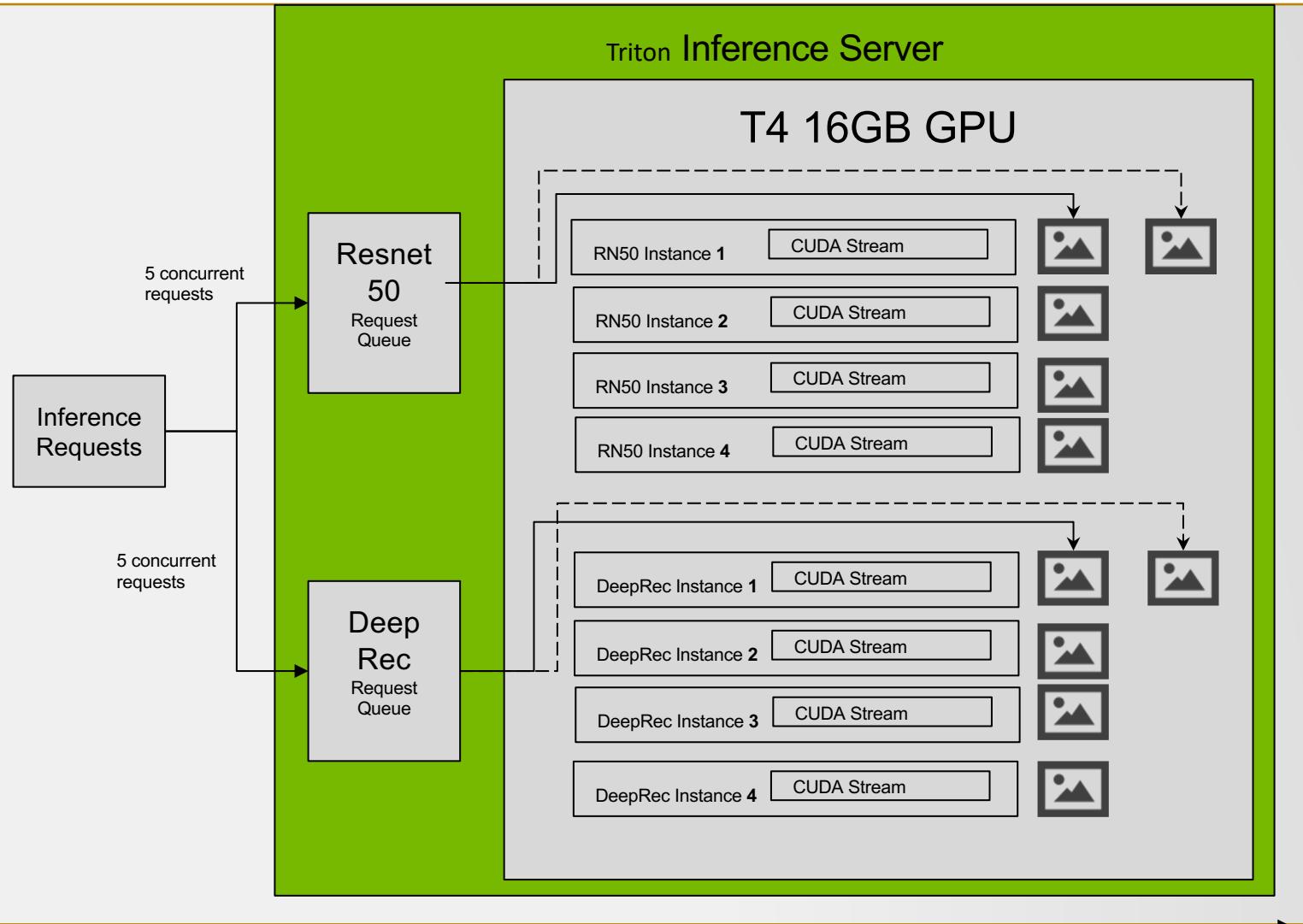
NVIDIA TRITON Inference Server – Concurrent Model Execution

Common Scenario 2

Many APIs using multiple different models on a GPU

Example:

- 4 instances of TRT FP16 ResNet50 and 4 instances of TRT FP16 Deep Recommender are running concurrently on one GPU.
- Ten requests come in for both models at the same time (5 for each model) and fed to the appropriate model for inference.
- The requests are fulfilled concurrently and sent back to the user. One request is queued for each model.



NVIDIA TRITON Inference Server – Autoscaling

Before Triton Inference Server - 800 FPS



Before Triton Inference Server - 5,000 FPS



- One model per GPU
- Requests are steady across all models
- Utilization is low on all GPUs

- Spike in requests for blue model
- GPUs running blue model are being fully utilized
- Other GPUs remain underutilized

NVIDIA TRITON Inference Server – Autoscaling

After Triton Inference Server - 5,000 FPS



- Load multiple models on every GPU
- Load is evenly distributed between all GPUs

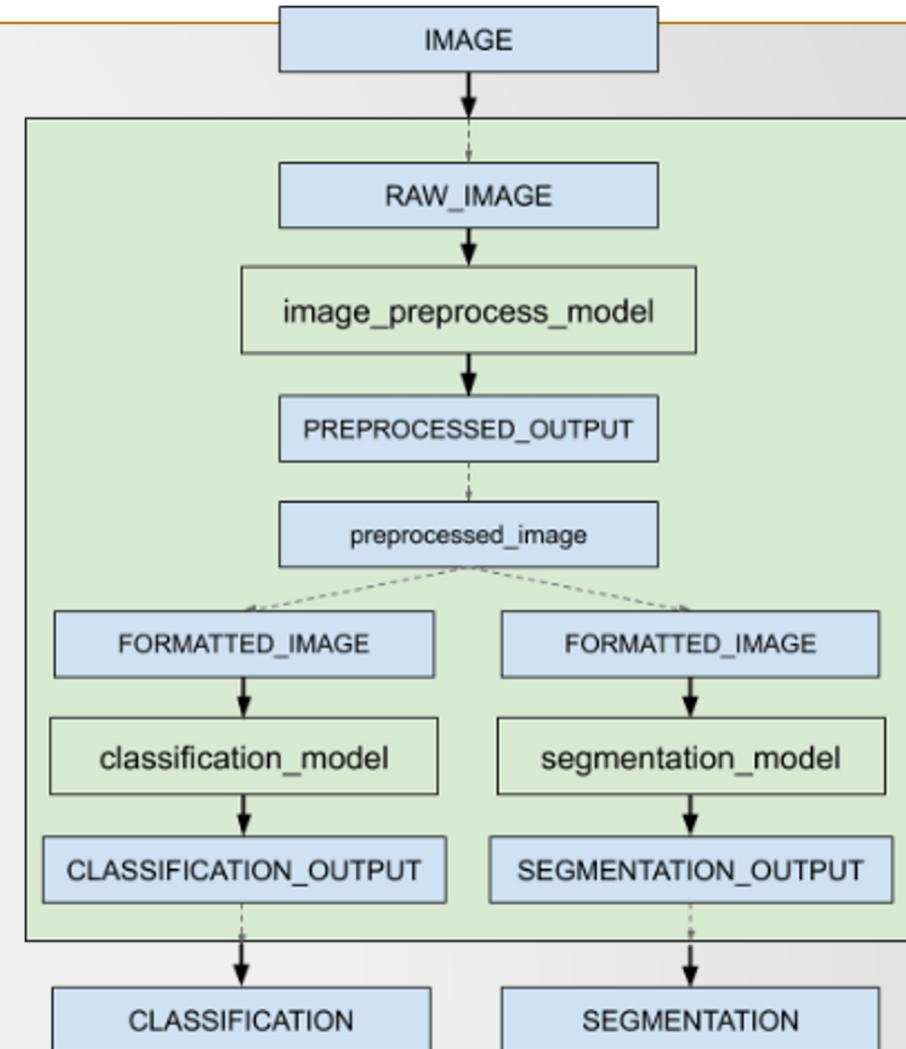
After Triton Inference Server - 15,000 FPS



- Spike in requests for blue model
- Each GPU can run the blue model concurrently
- Metrics to indicate time to scale up
 - GPU utilization
 - Power usage
 - Inference count
 - Queue time
 - Number of requests/sec

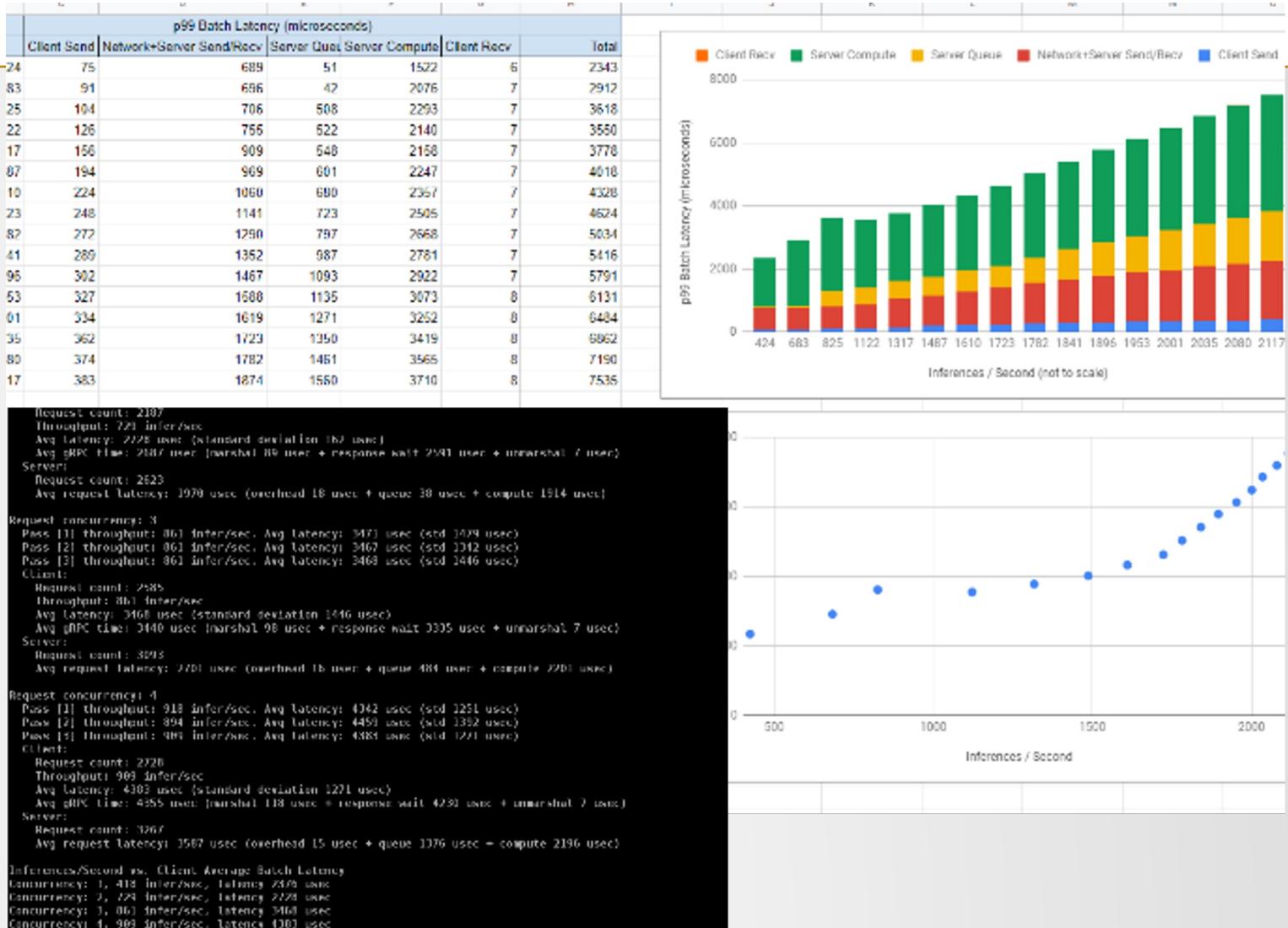
NVIDIA TRITON Inference Server – Model Ensembling

- Pipeline of one or more models and the connection of input and output tensors between those models
- Use for model stitching or data flow of multiple models such as data preprocessing → inference → data post-processing
- Collects the output tensors in each step, provides them as input tensors for other steps according to the specification
- Ensemble models will inherit the characteristics of the models involved, so the meta-data in the request header must comply with the models within the ensemble



NVIDIA TRITON Inference Server – Perf_Client Tool

- Measures throughput (inf/s) and latency under varying client loads
- **perf_client Modes**
 1. Specify how many concurrent outstanding requests and it will find a stable latency and throughput for that level
 2. Generate throughput vs latency curve by increasing the request concurrency until a specific latency or concurrency limit is reached
- Generates a file containing CSV output of the results
- Easy steps to help visualize the throughput vs latency tradeoffs



NVIDIA TRITON Inference Server – Perf_Client Tool

Deploy the CPU workloads used today and benefit from Triton Inference Server features (TRT not required)

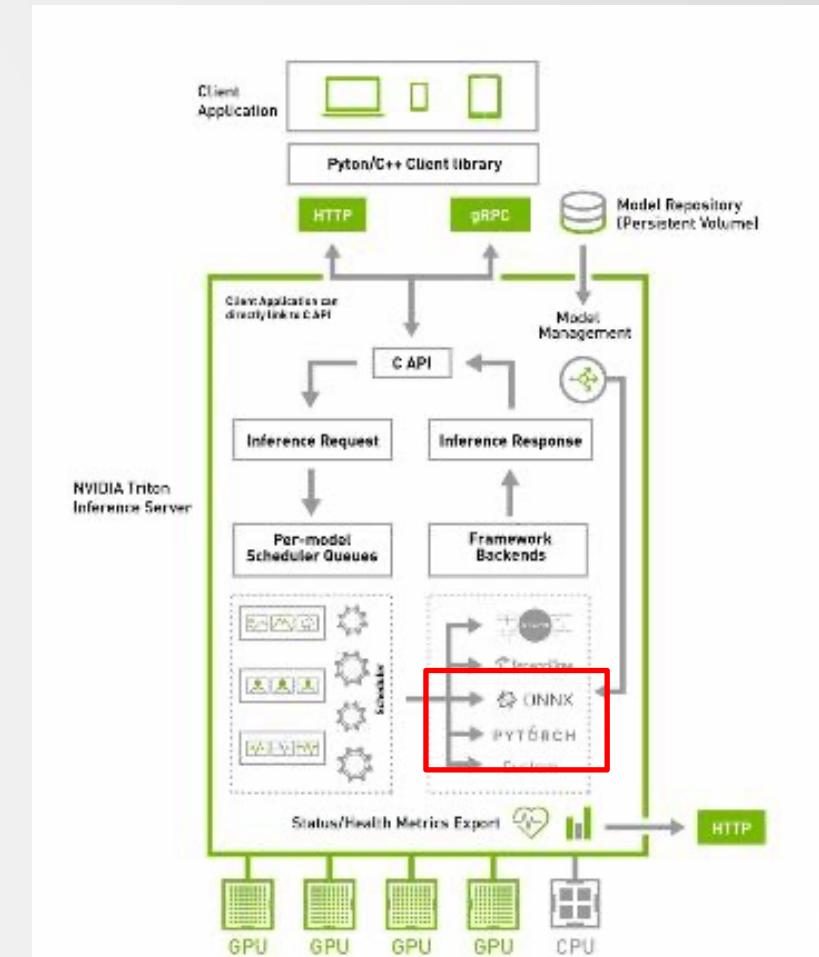
Triton relies on framework backends (Tensorflow, Caffe2, PyTorch) to execute the inference request on CPU

Support for Tensorflow and Caffe2 CPU optimizations using Intel MKL-DNN library

Allows frameworks backends to make use of multiple CPUs and cores

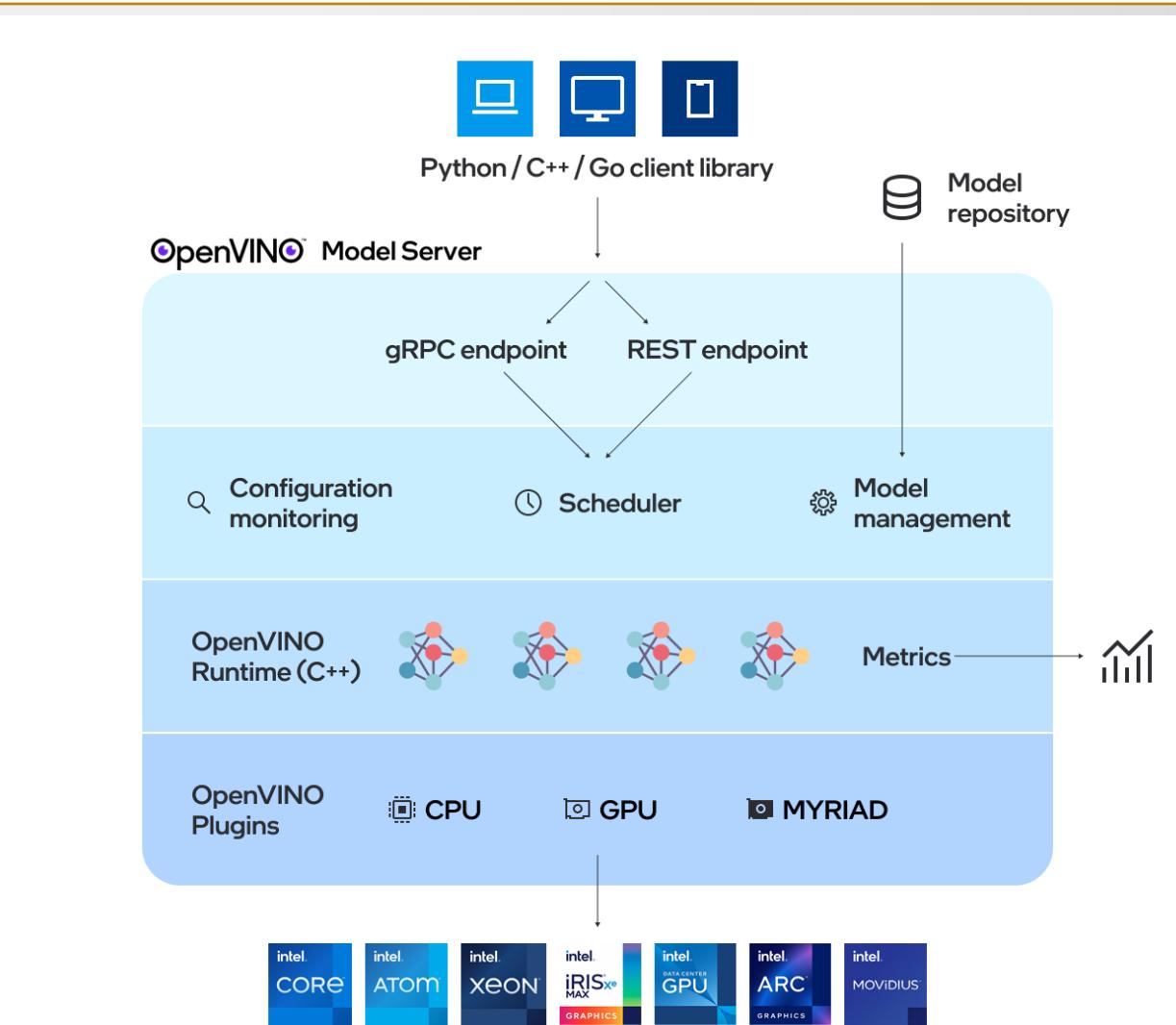
Benefit from Triton features:

- Multiple Model Framework Support
- Dynamic batching
- Custom backend
- Model Ensembling
- Audio Streaming API



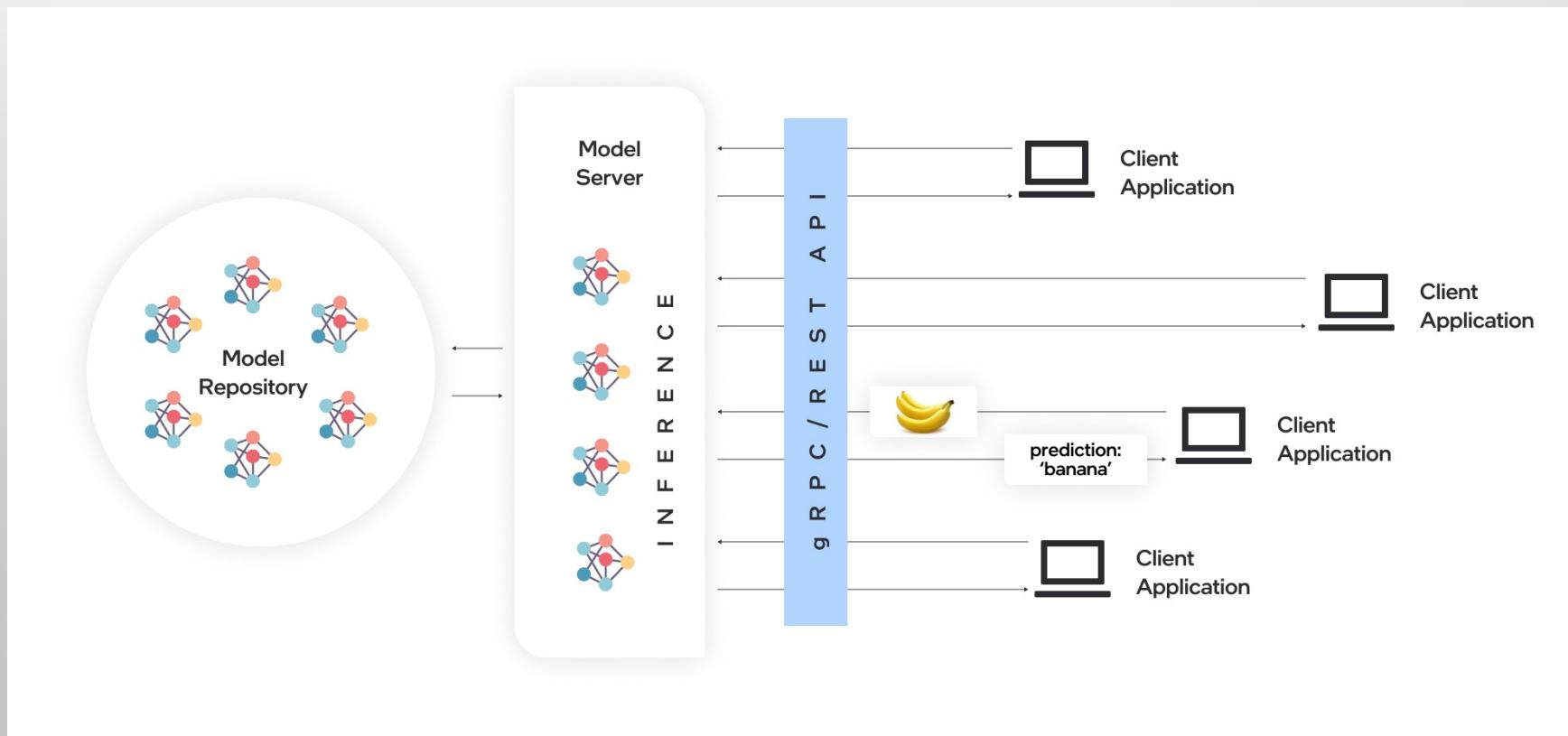
OpenVINO™ Model Server

Inference service is provided via gRPC or REST API, making deploying new algorithms and AI experiments easy.



OpenVINO™ Model Server

Model Server hosts models and makes them accessible to software components over standard network protocols: a client sends a request to the model server, which performs model inference and sends a response back to the client.



NVIDIA NGC

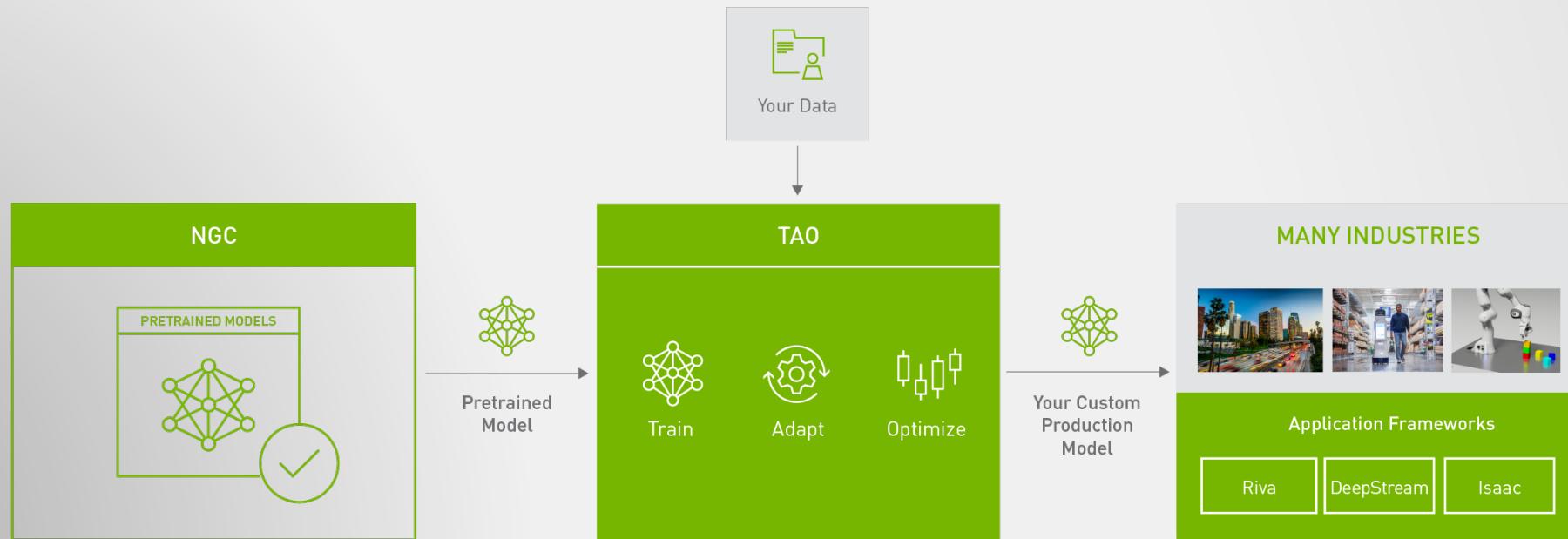
The NGC catalog offers pre-trained models for a variety of common AI tasks that are optimized for NVIDIA Tensor Core GPUs, and can be easily re-trained by updating just a few layers

 TAO	 TAO	 TAO	 TAO
TAO Pretrained EfficientDet Model Pretrained weights to facilitate transfer learning using TAO Toolkit. View Labels Download	TAO Pretrained Classification Model Pretrained weights to facilitate transfer learning using TAO Toolkit. View Labels Download	TAO Pretrained DetectNet V2 Model Pretrained weights to facilitate transfer learning using TAO Toolkit. View Labels Download	TAO Pretrained Instance Segmentation Model Pretrained weights to facilitate transfer learning using TAO Toolkit. View Labels Download
 TAO	 TAO	 TAO / RIVA	 TAO / RIVA
TAO Pretrained Object Detection Model Pretrained weights to facilitate transfer learning using TAO Toolkit.	TAO Pretrained Semantic Segmentation Model Pretrained weights to facilitate transfer learning using Transfer Learning Toolkit.	Riva ASR English LM Model Base English n-gram LM trained on LibriSpeech, Switchboard and Fisher	Speech Synthesis English FastPitch Model Mel-Spectrogram prediction conditioned on input text with LJSpeech voice.

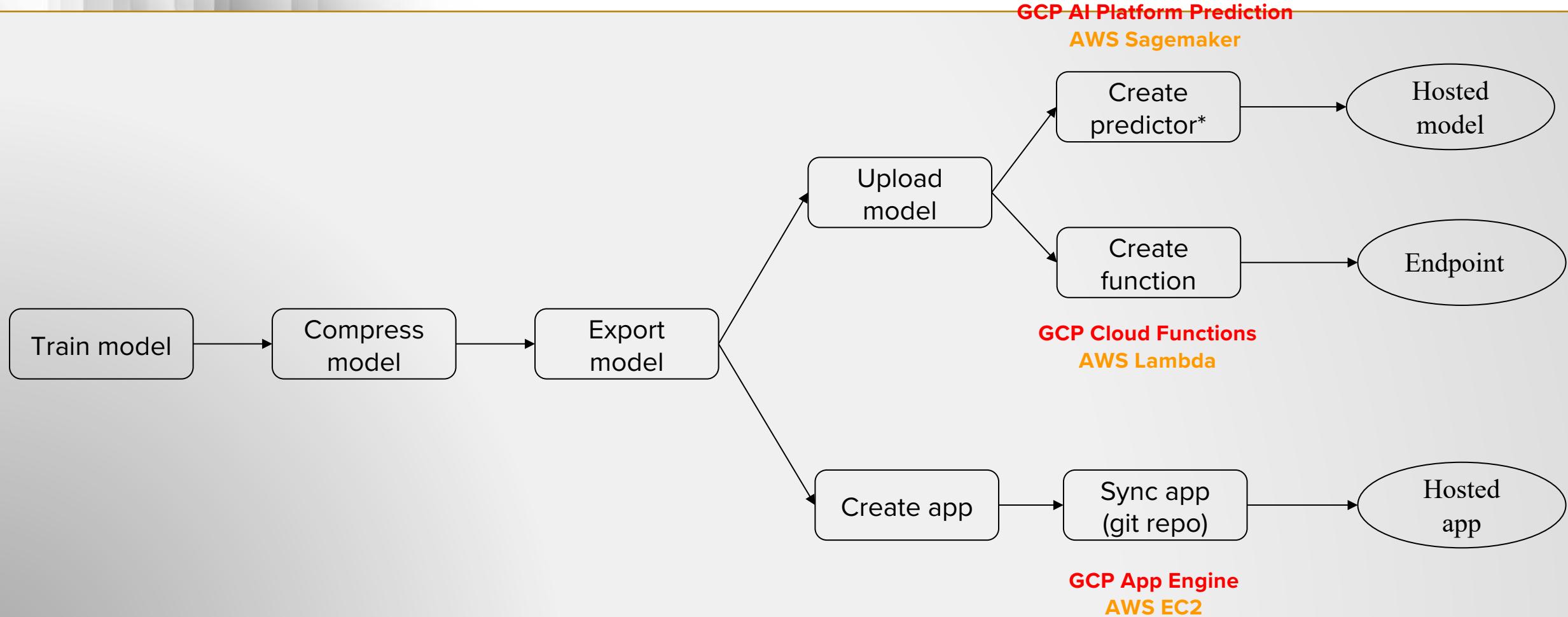
NVIDIA TAO

TAO (Train-Adapt-Optimize) is an AI-model-adaptation framework that lets enterprise application developers fine-tune pretrained models with custom data to quickly produce highly accurate models.

- 1 Choose from NVIDIA's library of pretrained models
- 2 Quickly train, adapt, and optimize models to your unique application
- 3 Integrate your customized models into your application and deploy



Google Cloud Platform (GCP) and Amazon Web Services (AWS)



Export a model

- Turn models into binary formats
 - scikit-learn, XGBoost -> joblib, pickle
 - TensorFlow -> .save()
 - PyTorch -> .save()

GCP AI Platform: hosted model

1. Upload model to cloud storage
2. Create a version for your model

[←](#) Create version

Pre-built container settings

Python version *

3.7

Select the Python version you used to train the model

Framework

scikit-learn

Framework version

0.23.2

ML runtime version *

2.3

 gs:// Model URI *

BROWSE

Path to the Cloud Storage directory where the exported model file is stored (not the path to the model file itself). The model name must be one of: model.pkl or model.joblib. [Learn more](#)

GCP AI Platform: hosted model

1. Upload model to cloud storage
2. Create a version for your model

Output looks like
this

```
{  
  "name": "projects/[YOUR-PROJECT-ID]/operations/create_[YOUR-MODEL-NAME]_[YOUR-VERSION-NAME]-[TIMESTAMP]",  
  "metadata": {  
    "@type": "type.googleapis.com/google.cloud.ml.v1.OperationMetadata",  
    "createTime": "2018-07-07T02:51:50Z",  
    "operationType": "CREATE_VERSION",  
    "modelName": "projects/[YOUR-PROJECT-ID]/models/[YOUR-MODEL-NAME]",  
    "version": {  
      "name": "projects/[YOUR-PROJECT-ID]/models/[YOUR-MODEL-NAME]/versions/[YOUR-VERSION-NAME]",  
      "deploymentUri": "gs://your_bucket_name",  
      "createTime": "2018-07-07T02:51:49Z",  
      "runtimeVersion": "2.3",  
      "framework": "[YOUR_FRAMEWORK_NAME]",  
      "machineType": "[YOUR_MACHINE_TYPE]",  
      "pythonVersion": "3.7"  
    }  
  }  
}
```

GCP AI Platform

1. Train model
2. Deploy a model & version
3. Make online/batch predictions

```
import googleapiclient.discovery

def predict_json(project, model, instances, version=None):
    """Send json data to a deployed model for prediction.

    Args:
        project (str): project where the AI Platform Prediction Model is deployed.
        model (str): model name.
        instances ([[float]]): List of input instances, where each input
            instance is a list of floats.
        version: str, version of the model to target.

    Returns:
        Mapping[str: any]: dictionary of prediction results defined by the
            model.

    """
    # Create the AI Platform Prediction service object.
    # To authenticate set the environment variable
    # GOOGLE_APPLICATION_CREDENTIALS=<path_to_service_account_file>
    service = googleapiclient.discovery.build('ml', 'v1')
    name = 'projects/{}/models/{}'.format(project, model)

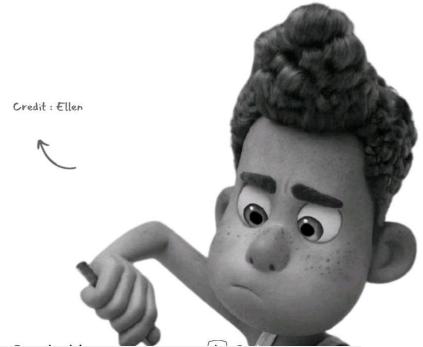
    if version is not None:
        name += '/versions/{}'.format(version)

    response = service.projects().predict(
        name=name,
        body={'instances': instances}
    ).execute()

    if 'error' in response:
        raise RuntimeError(response['error'])

    return response['predictions']
```

WHAT IS A DOCKER ?

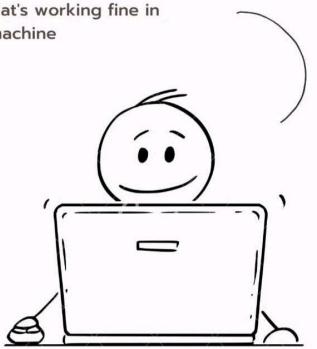


Credit : Ellen

Development

Lets say You created
an Application

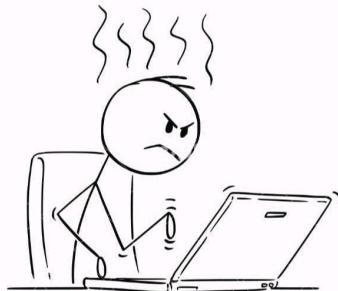
And that's working fine in
your machine



Production

But in Production it
doesn't work properly

Developers experince it a lot



The Reason could be due to :

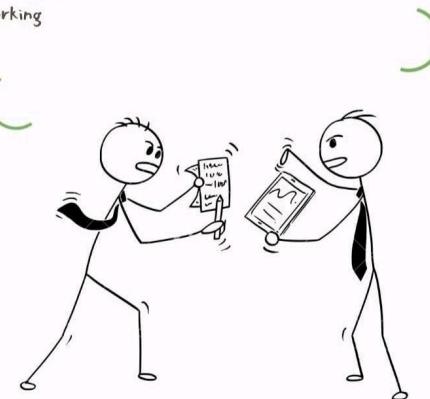
- Dependencies
- Libraries and versions
- Framework
- OS Level features
- Microservices

That the developers machine has but
not there in the **production environment**

That is when the Developer's
famous words are spoken

Your application is
not working

It works on my machine



DOCKER

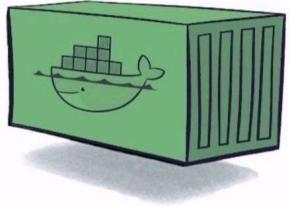
We need a standardized way to package
the application with its **dependencies** and
deploy it on any environment.



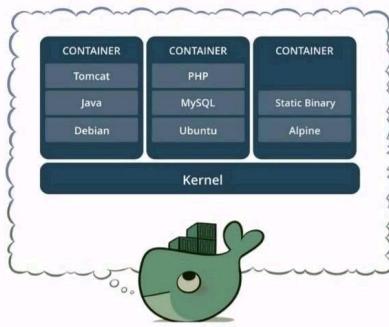
Docker is a tool designed to make it easier
to create, deploy, and run applications by
using containers.

How Does Docker Work?

Docker packages an application and all its dependencies in a virtual container that can run on any Linux server.



Each container runs as an isolated process in the user space and take up less space than regular VMs due to their layered architecture.



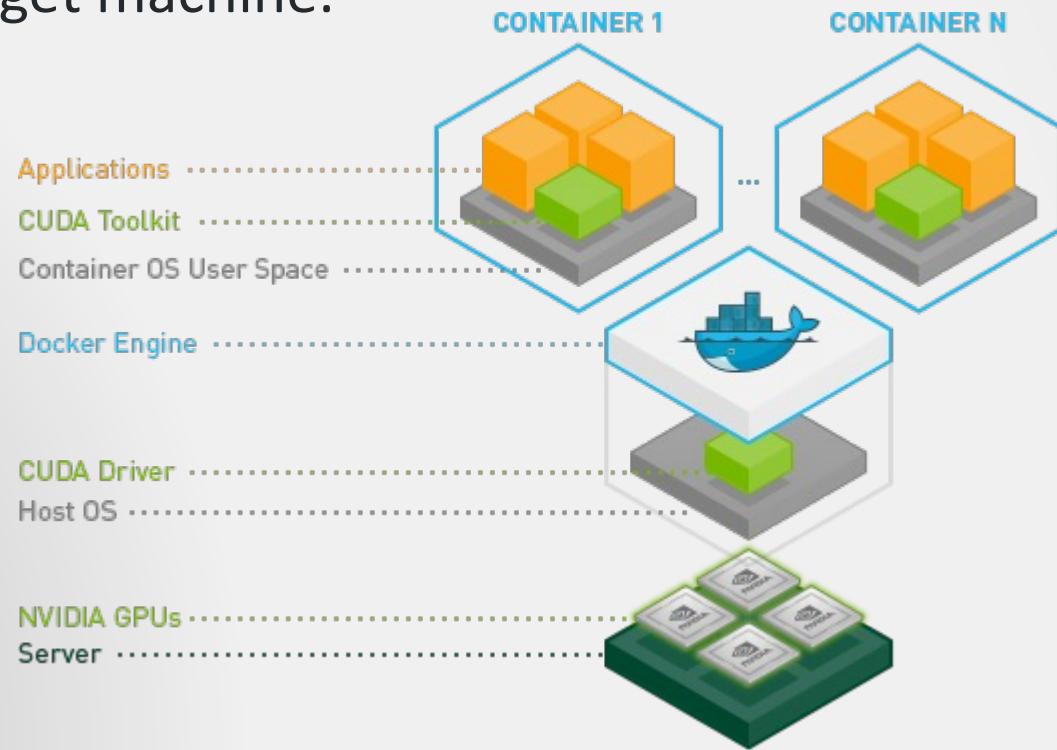
So it will always work the same regardless of its environment

Docker containers

- ❑ Docker® containers are often used to seamlessly deploy CPU-based applications on multiple machines.
- ❑ Containers are both hardware-agnostic and platform-agnostic.
- ❑ This is obviously not the case when using NVIDIA GPUs since it is using specialized hardware and it requires the installation of the NVIDIA driver.
- ❑ As a result, Docker Engine does not natively support NVIDIA GPUs with containers.

NVIDIA Docker

- ❑ To make the Docker images portable while still leveraging NVIDIA GPUs, nvidia-docker makes the images agnostic of the NVIDIA driver.
- ❑ The required character devices and driver files are mounted when starting the container on the target machine.



GPU containerization

Containerizing GPU applications provides several benefits,:

- ❑ Reproducible builds
- ❑ Ease of deployment
- ❑ Isolation of individual devices
- ❑ Run across heterogeneous driver/toolkit environments
- ❑ Requires only the NVIDIA driver to be installed
- ❑ Enables "fire and forget" GPU applications
- ❑ Facilitate collaboration

Container registry

- A container registry is collection of repositories, used to store container images for Kubernetes, DevOps, and container-based application development.
- Single place to manage docker images
 - NVIDIA NGC Containers
 - Amazon Elastic Container Registry
 - Google Container Registry
 - Microsoft Azure Container Registry