Mastering Transformers: From Building Blocks to Real-World Applications

Deployment

Prof. Alptekin Temizel

13 Sept 2023

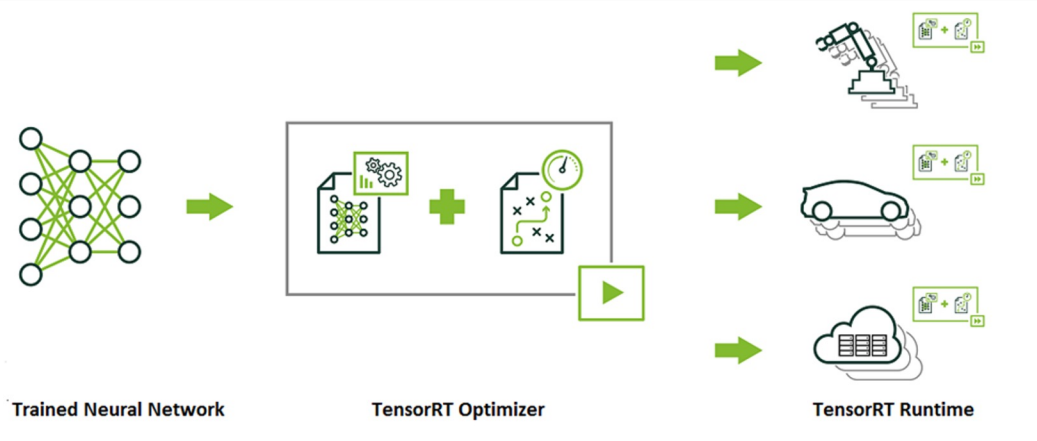
Deployment Frameworks

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

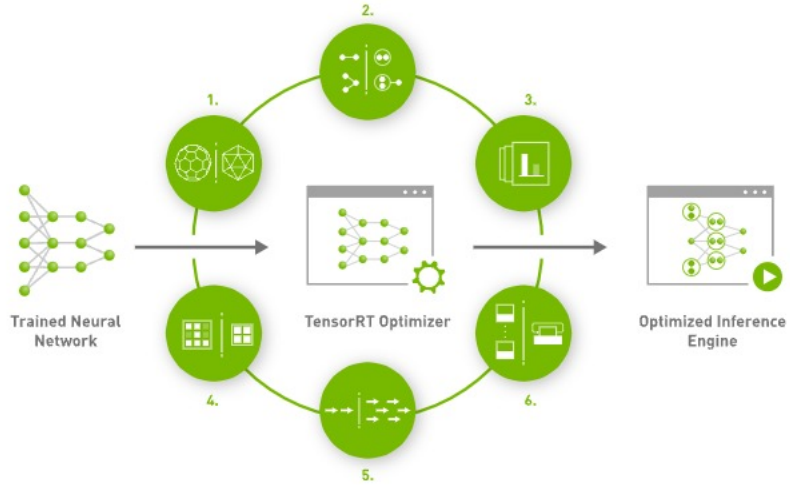
TensorRT

- Inference engine for production deployment of deep learning applications



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

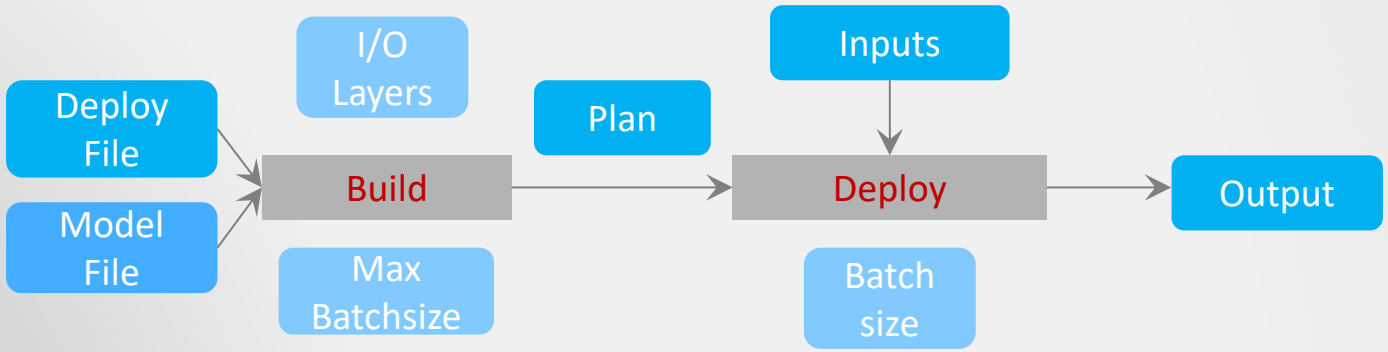
TensorRT Optimizer



1. **Weight & Activation Precision Calibration**
   Maximizes throughput by quantizing models to INT8 while preserving accuracy
2. **Layer & Tensor Fusion**
   Optimizes use of GPU memory and bandwidth by fusing nodes in a kernel
3. **Kernel Auto-Tuning**
   Selects best data layers and algorithms based on target GPU platform
4. **Dynamic Tensor Memory**
   Minimizes memory footprint and re-uses memory for tensors efficiently
5. **Multi-Stream Execution**
   Scalable design to process multiple input streams in parallel
6. **Time Fusion**
   Optimizes recurrent neural networks over time steps with dynamically generated kernels

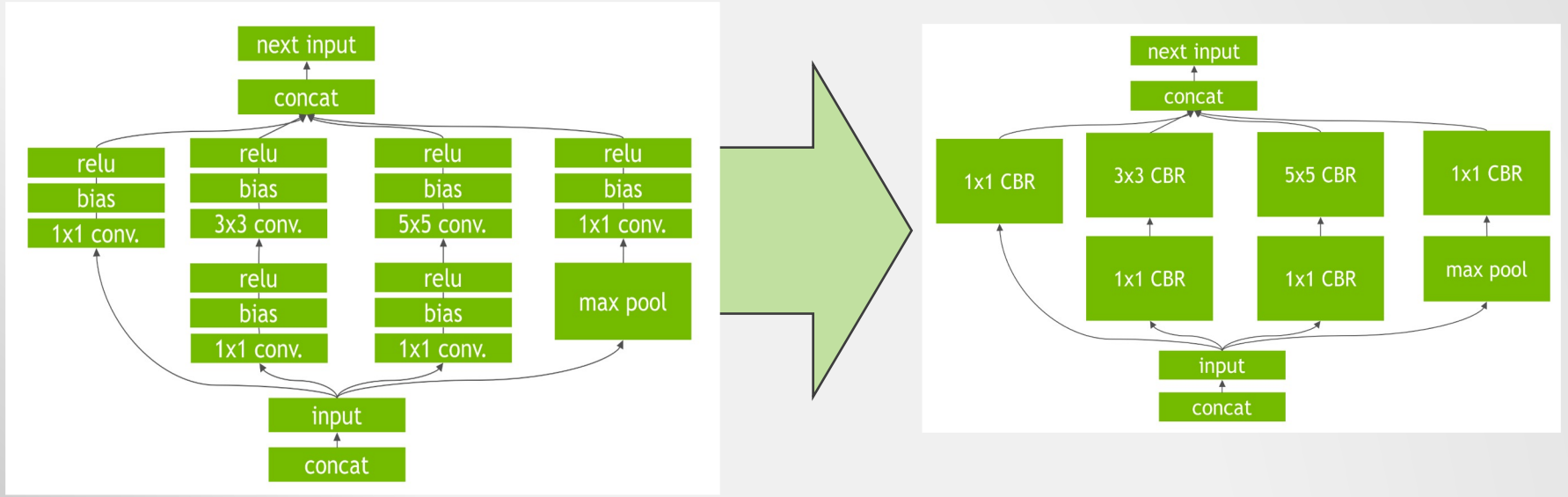
TensorRT RunTime

- **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

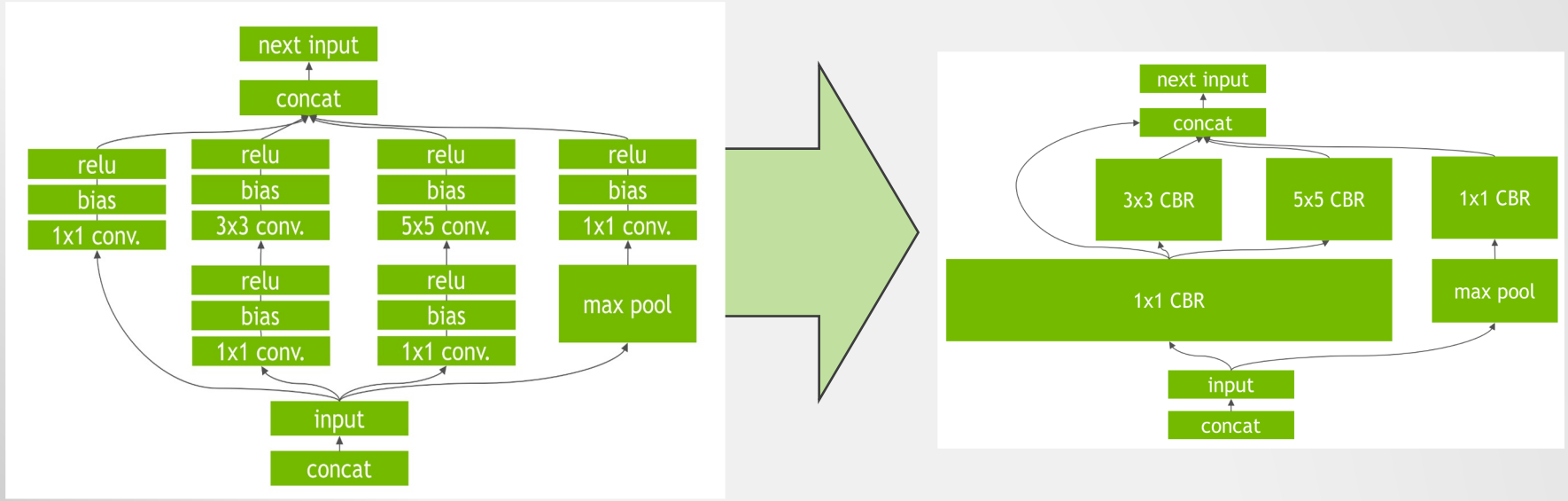


CBR = Convolution, Bias and ReLU

developer.nvidia.com/tensorrt

TensorRT Optimizer

Horizontal Layer Fusion (Layer Aggregation)



CBR = Convolution, Bias and ReLU

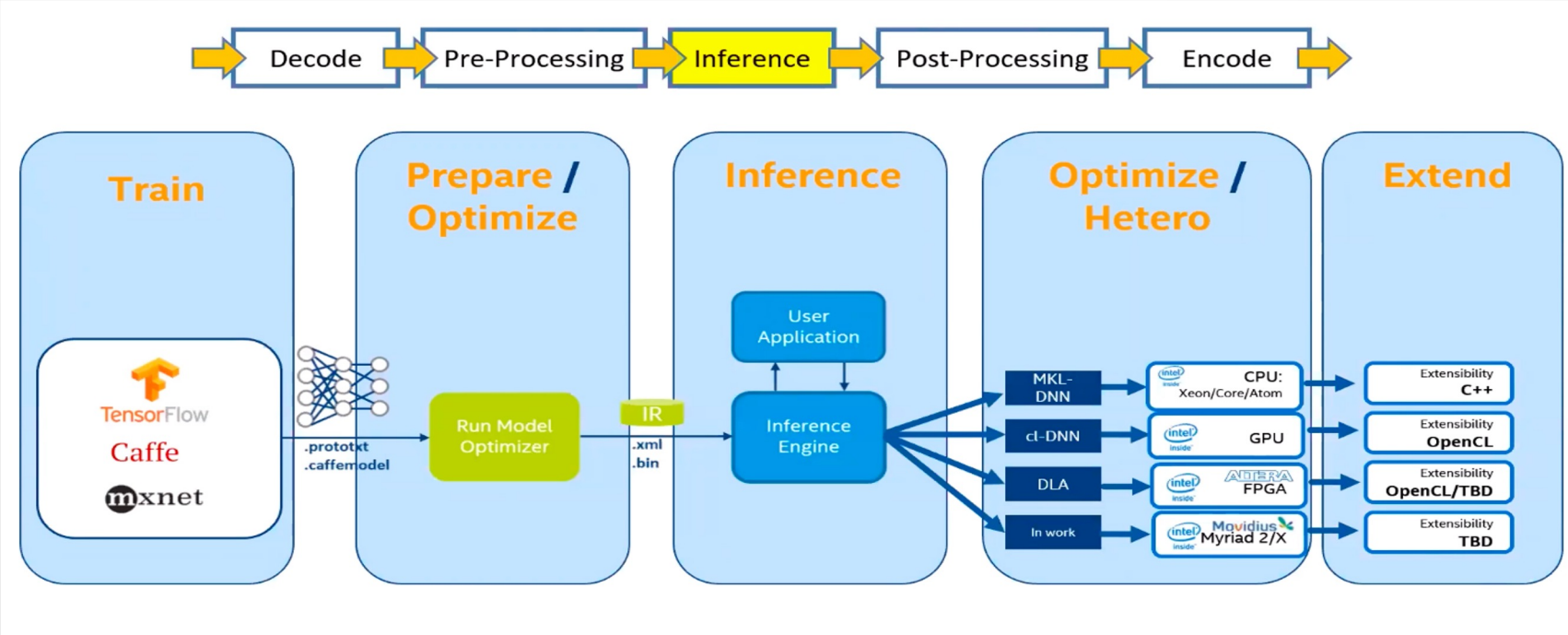
developer.nvidia.com/tensorrt

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.

MediaPipe

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

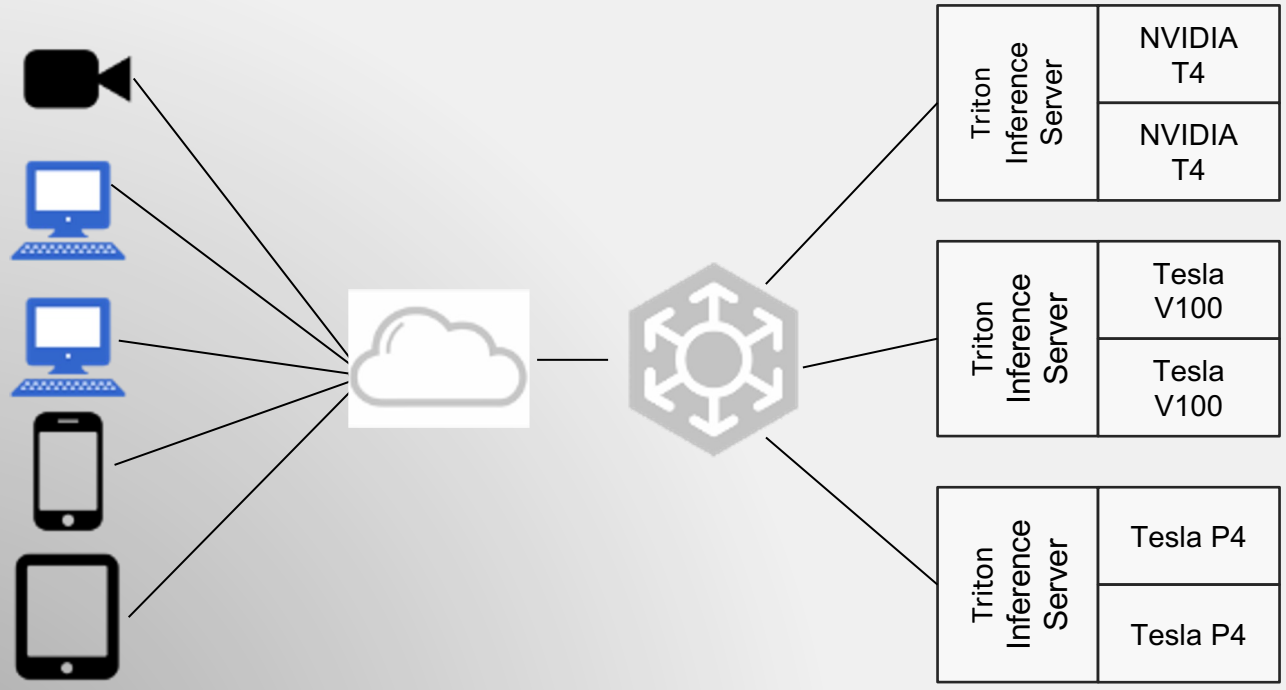
OpenMMLab Detection Toolbox

- 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/mmdetection

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

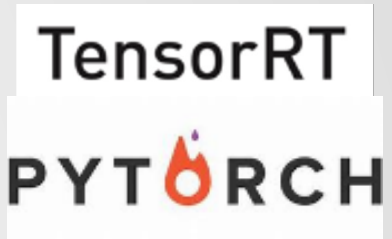
Model Ensemble
Pipeline of one or more models and the connection of input and output tensors between those models (can be used with custom backend)

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



NVIDIA TRITON Inference Server – Concurrent Model Execution

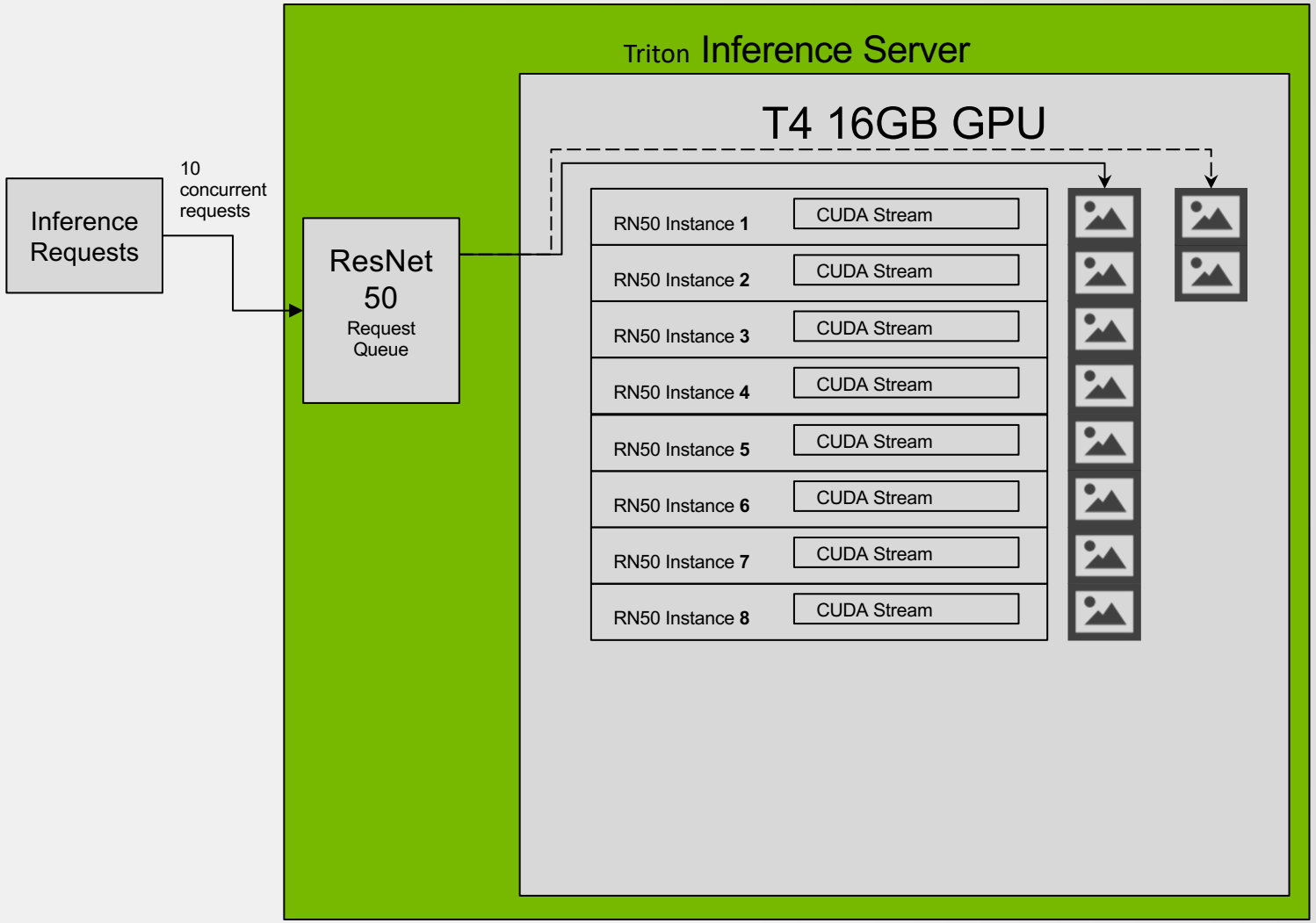
Better Performance and Improved GPU Utilization Through Multiple Model Concurrency

Common Scenario 1

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

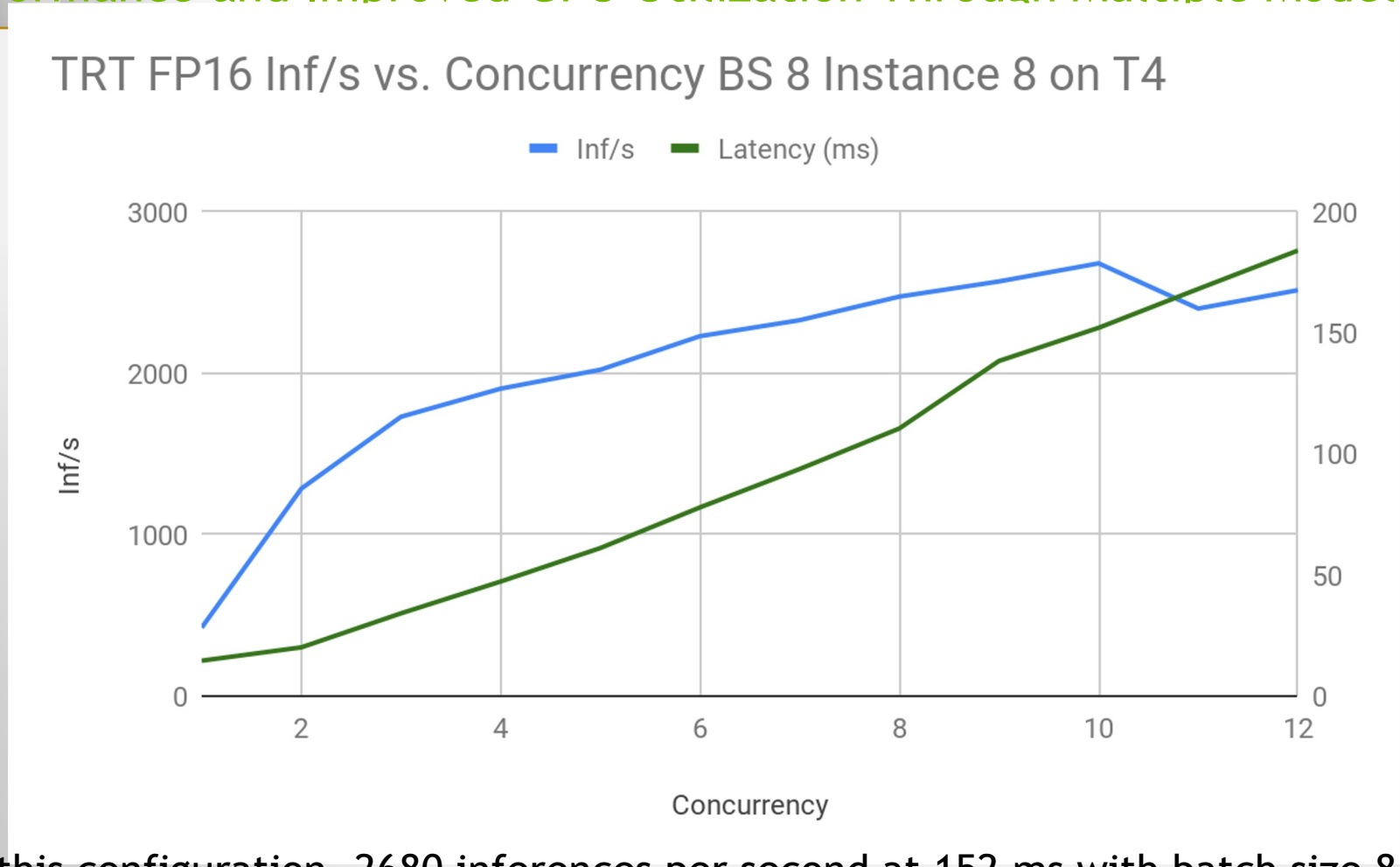
Example:

- 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.

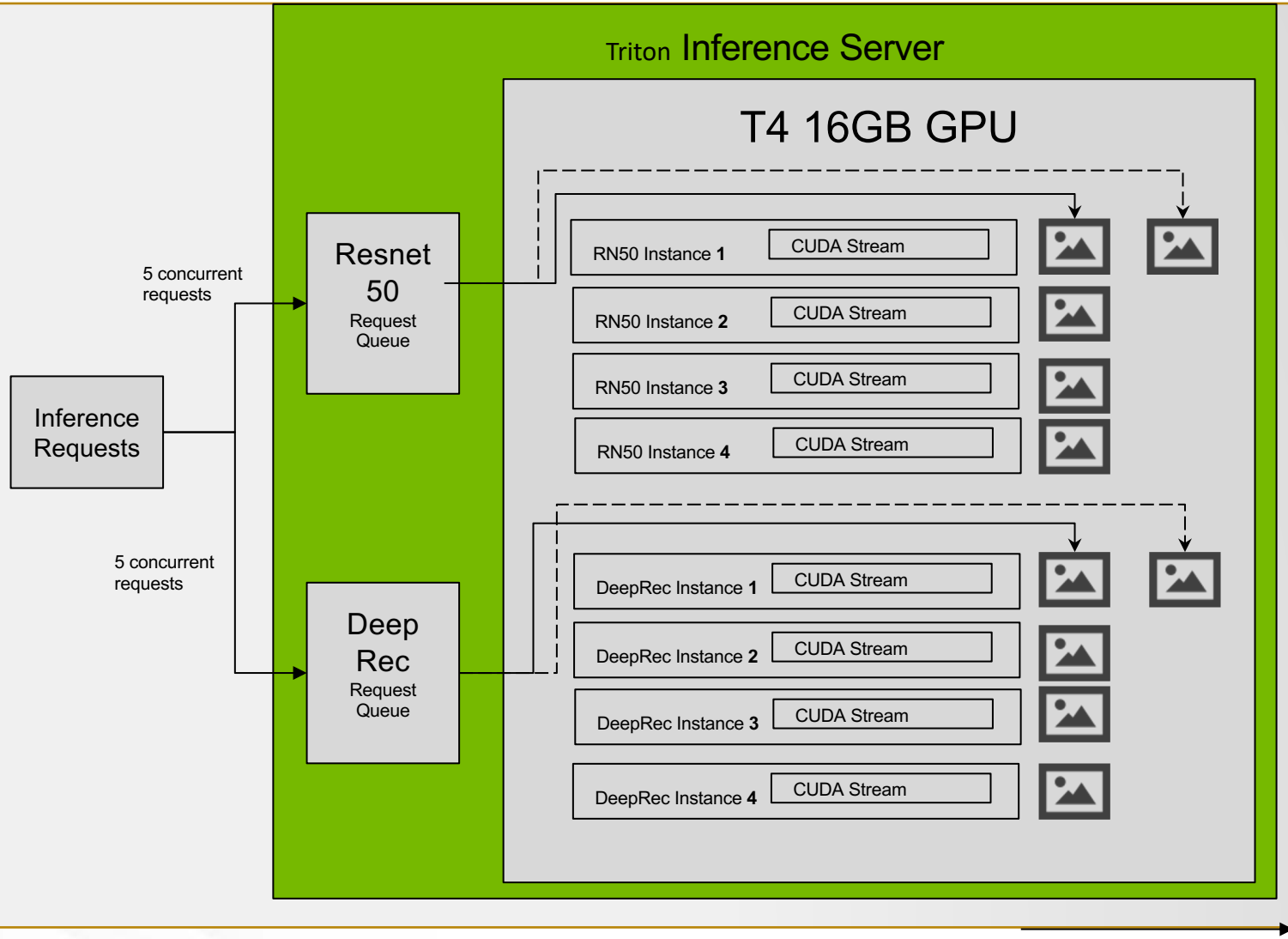
NVIDIA TRITON Inference Server – Concurrent Model Execution

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

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

Before Triton Inference Server - 5,000 FPS

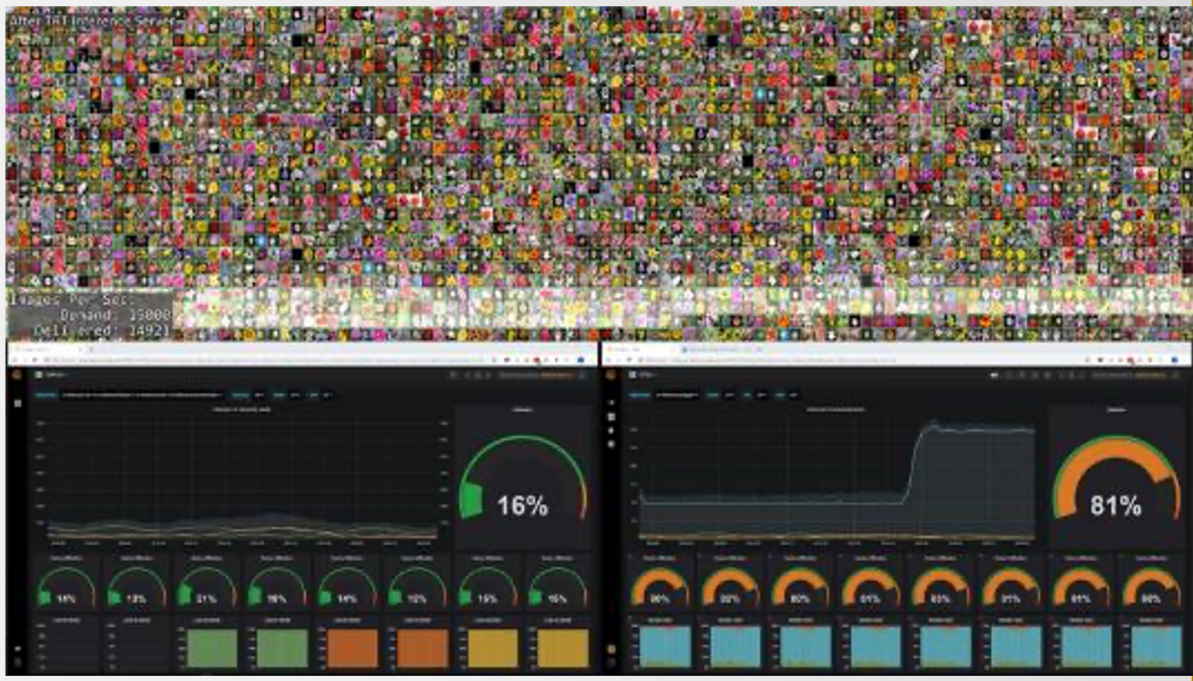
- Spike in requests for blue model
- GPUs running blue model are being fully utilized
- Other GPUs remain underutilized

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

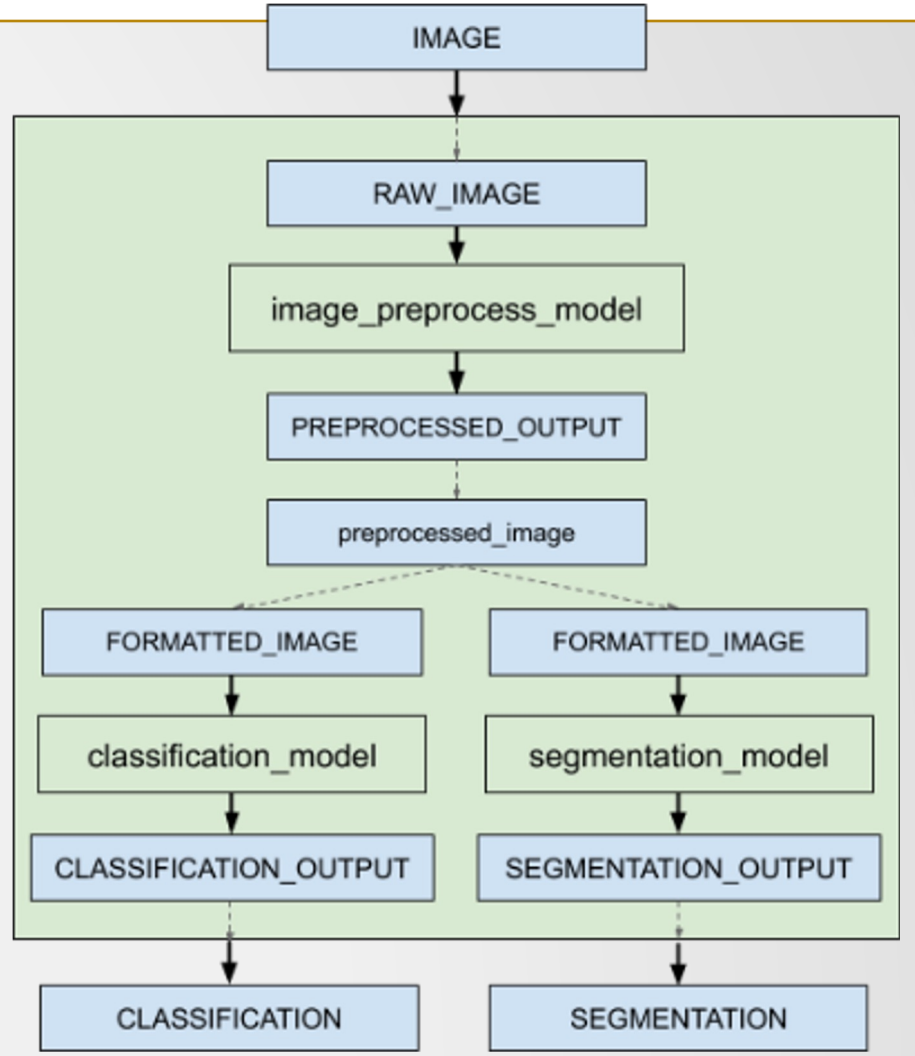
After Triton Inference Server - 15,000 FPS



- 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

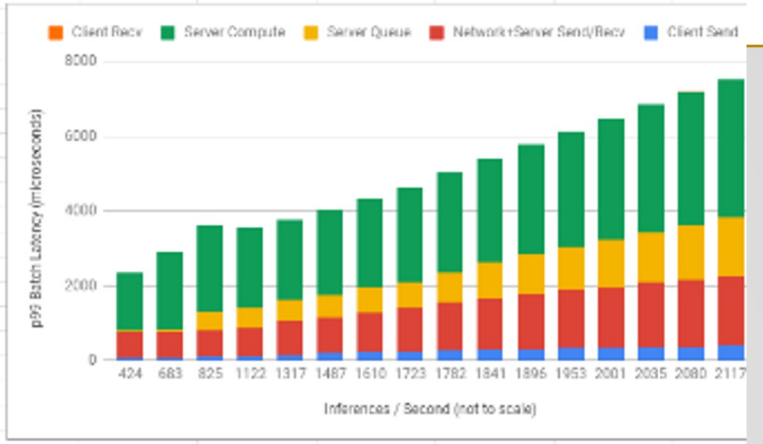
- Pipeline of one or more models and the connection of input and output tensors between those models
- Use for model stitching or data flow of multiple models such as data preprocessing → inference → data post-processing
- Collects the output tensors in each step, provides them as input tensors for other 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



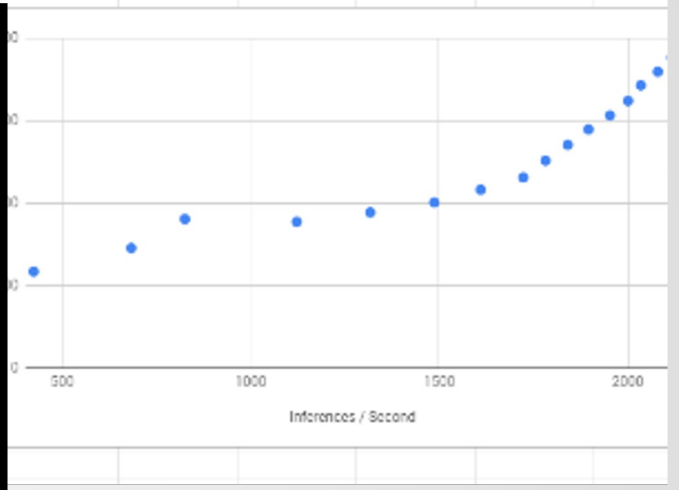
NVIDIA TRITON Inference Server – Perf_Client Tool

- 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

	p99 Batch Latency (microseconds)					
	Client Send	Network+Server Send/Recv	Server Queue	Server Compute	Client Recv	Total
24	75	689	51	1522	6	2343
83	91	696	42	2076	7	2912
25	104	706	508	2293	7	3618
22	126	755	522	2140	7	3550
17	156	909	548	2158	7	3778
87	194	969	601	2247	7	4018
10	224	1060	680	2357	7	4328
23	248	1141	723	2505	7	4624
82	272	1290	797	2668	7	5034
41	289	1352	987	2781	7	5416
96	302	1467	1093	2922	7	5791
53	327	1588	1135	3073	8	6131
01	334	1619	1271	3252	8	6484
35	362	1723	1350	3419	8	6862
80	374	1782	1461	3565	8	7190
17	383	1874	1560	3710	8	7536




```
    Request count: 2187
    Throughput: 729 infer/sec
    Avg latency: 2729 usec (standard deviation 67 usec)
    Avg gRPC time: 2687 usec (marshal 89 usec + response wait 2591 usec + unmarshal 7 usec)
  Server:
    Request count: 2623
    Avg request latency: 1970 usec (overhead 18 usec + queue 38 usec + compute 1914 usec)

Request concurrency: 3
  Pass [1] throughput: 861 infer/sec. Avg latency: 3471 usec (std 1479 usec)
  Pass [2] throughput: 861 infer/sec. Avg latency: 3462 usec (std 1342 usec)
  Pass [3] throughput: 861 infer/sec. Avg latency: 3468 usec (std 1446 usec)
  Client:
    Request count: 2585
    Throughput: 861 infer/sec
    Avg latency: 3468 usec (standard deviation 1446 usec)
    Avg gRPC time: 3440 usec (marshal 98 usec + response wait 3335 usec + unmarshal 7 usec)
  Server:
    Request count: 3095
    Avg request latency: 2701 usec (overhead 15 usec + queue 484 usec + compute 2201 usec)

Request concurrency: 4
  Pass [1] throughput: 918 infer/sec. Avg latency: 4342 usec (std 1251 usec)
  Pass [2] throughput: 894 infer/sec. Avg latency: 4459 usec (std 1392 usec)
  Pass [3] throughput: 909 infer/sec. Avg latency: 4384 usec (std 1271 usec)
  Client:
    Request count: 2720
    Throughput: 909 infer/sec
    Avg latency: 4383 usec (standard deviation 1271 usec)
    Avg gRPC time: 4355 usec (marshal 118 usec + response wait 4230 usec + unmarshal 7 usec)
  Server:
    Request count: 3267
    Avg request latency: 3507 usec (overhead 15 usec + queue 1376 usec + compute 2196 usec)

Inferences/Second vs. Client Average Batch Latency
Concurrency: 1, 418 infer/sec, latency 2370 usec
Concurrency: 2, 729 infer/sec, latency 2729 usec
Concurrency: 3, 861 infer/sec, latency 3468 usec
Concurrency: 4, 909 infer/sec, latency 4383 usec
```



NVIDIA TRITON Inference Server – Perf_Client Tool

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

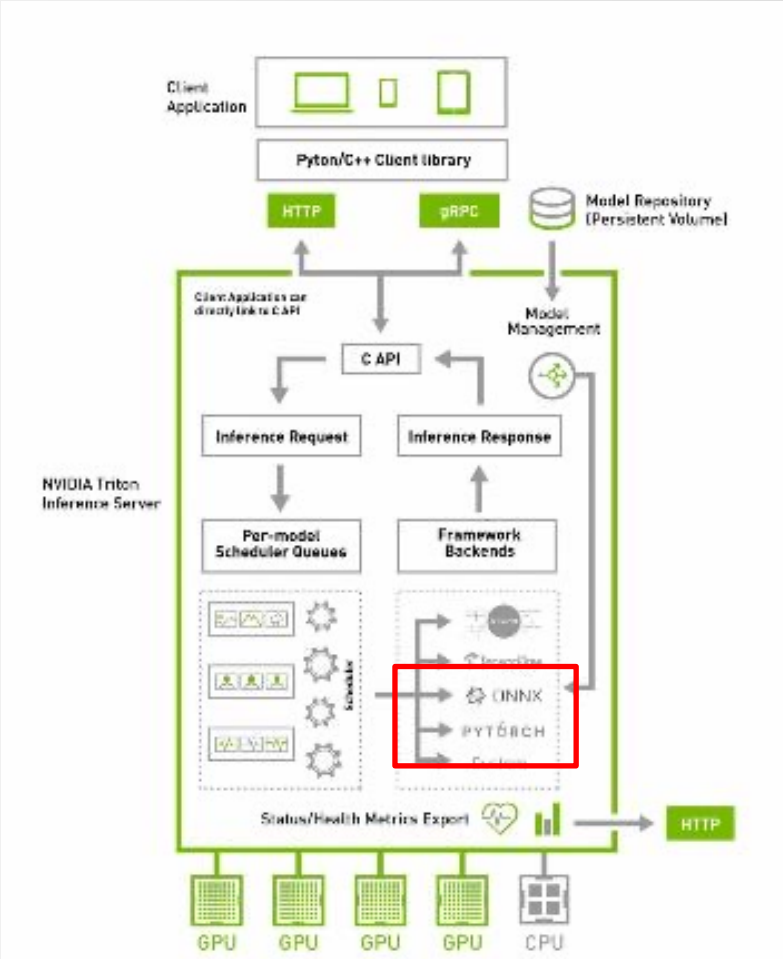
Triton relies on framework backends (Tensorflow, Caffe2, PyTorch) to execute the inference request on CPU

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

Allows frameworks backends to make use of multiple CPUs and cores

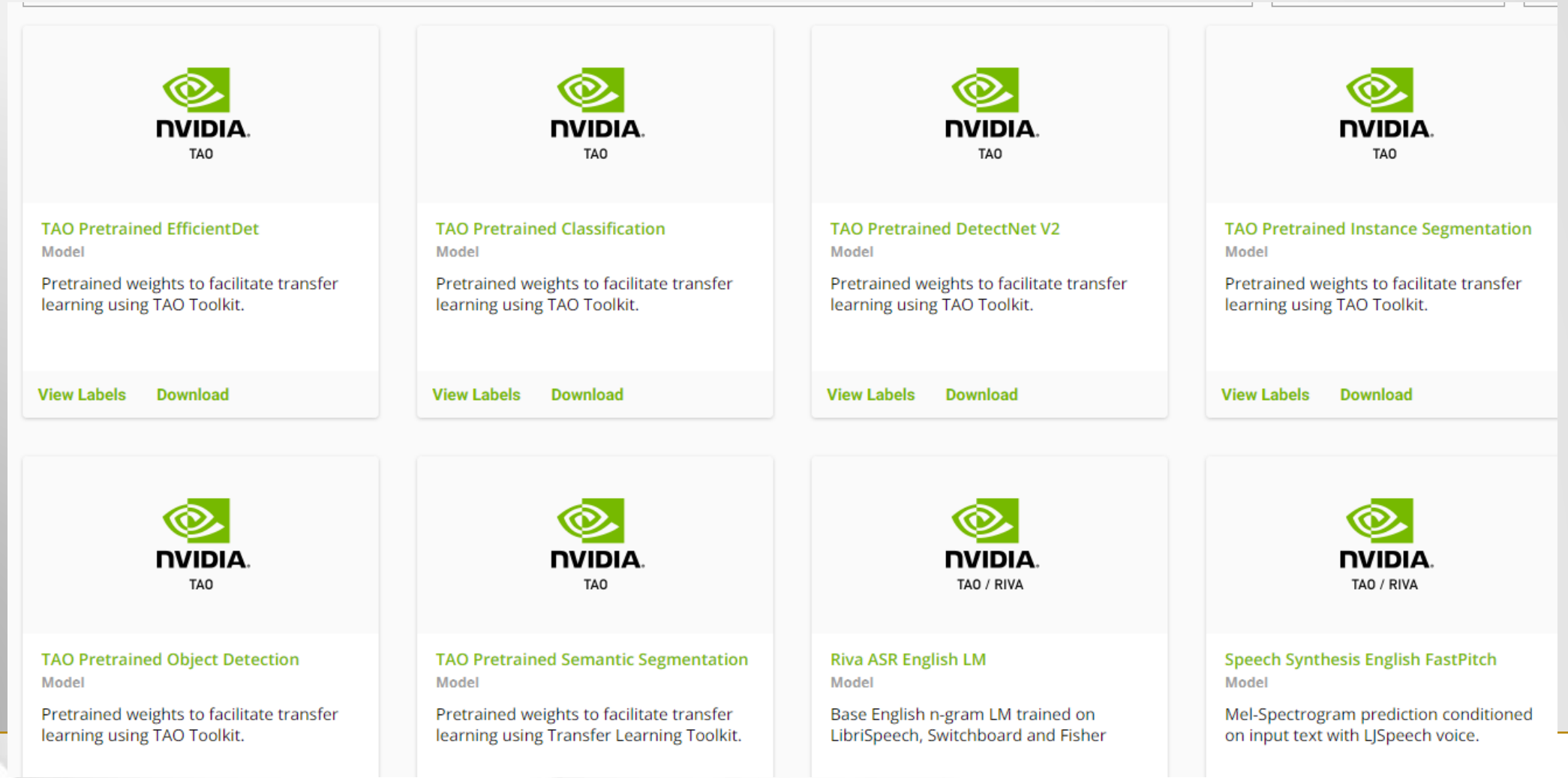
Benefit from Triton features:

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



NVIDIA NGC

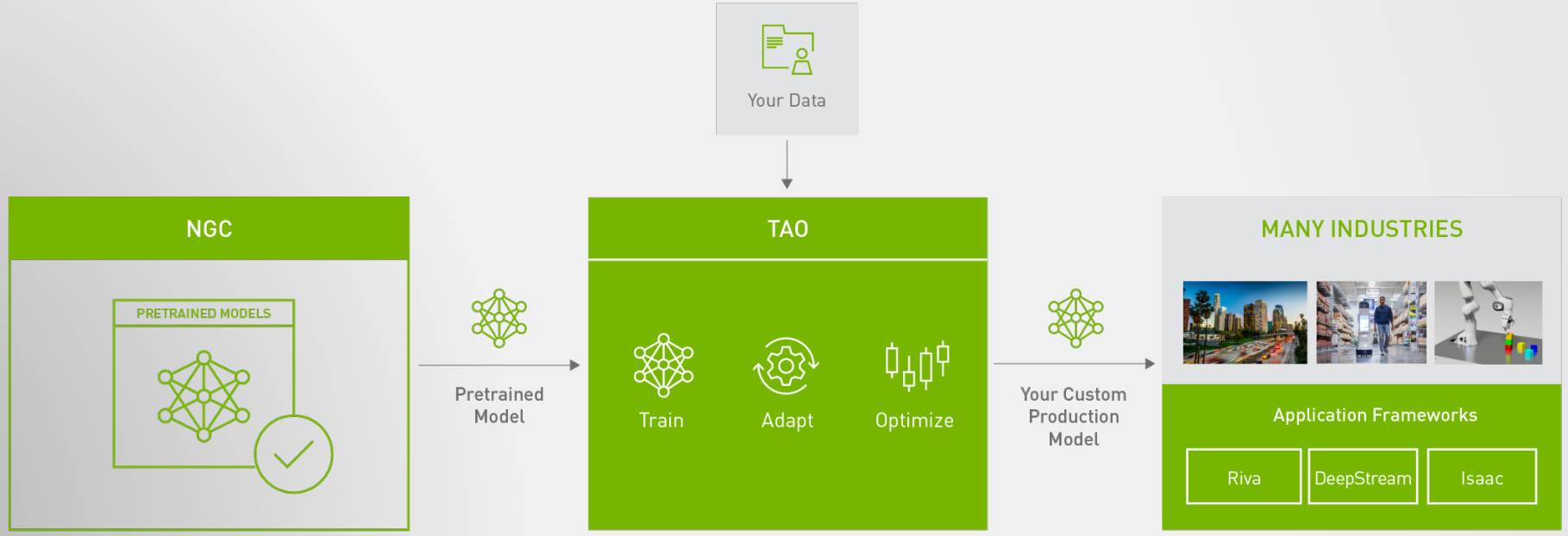
The NGC catalog offers pre-trained models for a variety of common AI tasks that are optimized for NVIDIA Tensor Core GPUs, and can be easily re-trained by updating just a few layers



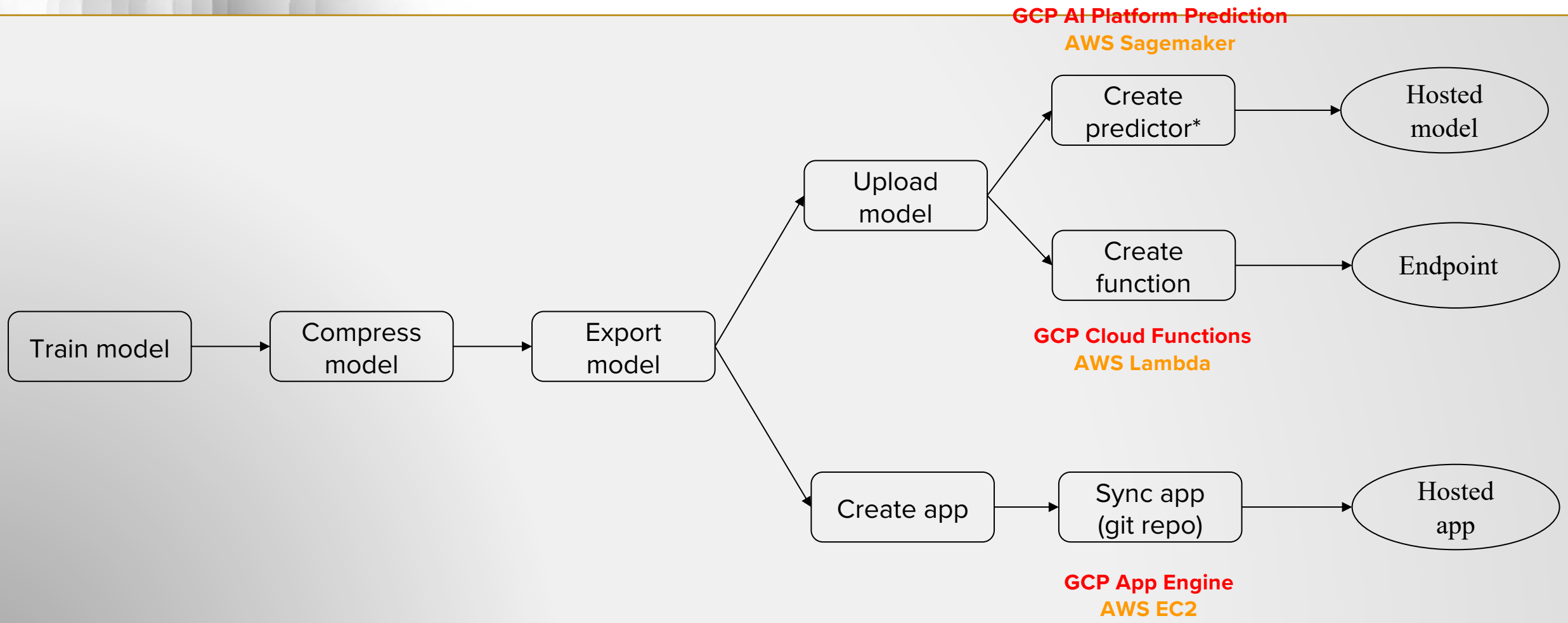
NVIDIA TAO

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)

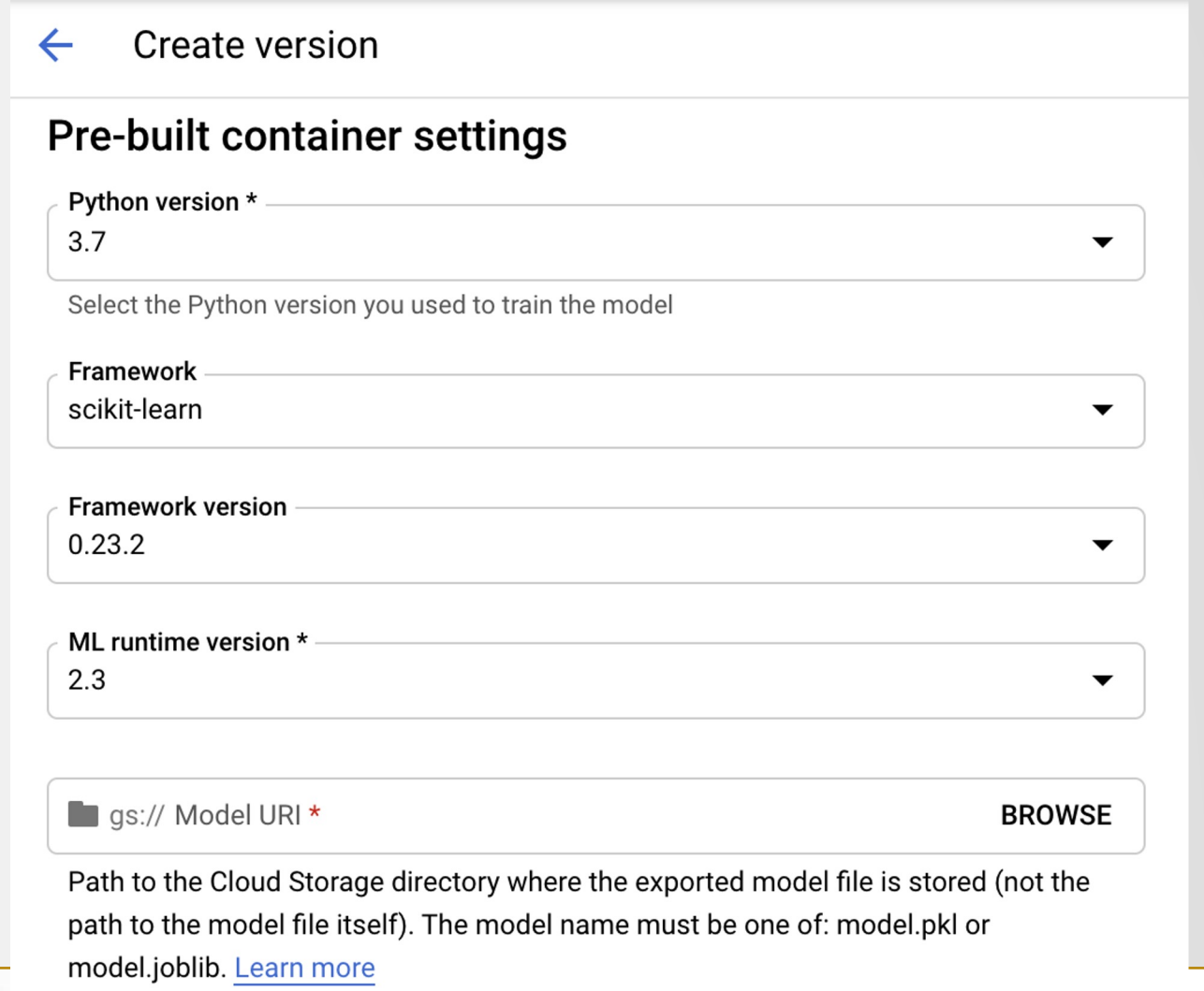


Export a model

- 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



GCP AI Platform: hosted model

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

Output looks like this

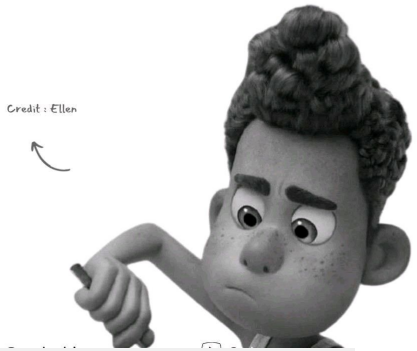
```
{
  "name": "projects/[YOUR-PROJECT-ID]/operations/create_[YOUR-MODEL-NAME]_[YOUR-VERSION-NAME]-[TIMESTAMP]",
  "metadata": {
    "@type": "type.googleapis.com/google.cloud.ml.v1.OperationMetadata",
    "createTime": "2018-07-07T02:51:50Z",
    "operationType": "CREATE_VERSION",
    "modelName": "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

```python
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']
```

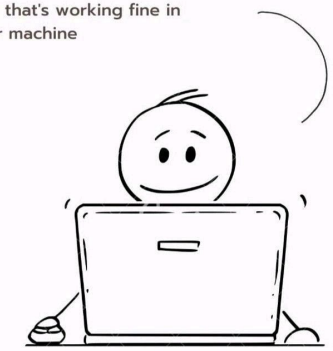
WHAT IS A DOCKER ?



Development

Lets say You created an Application

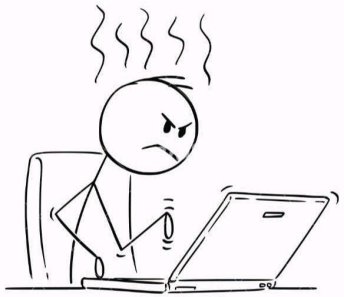
And that's working fine in your machine



Production

But in Production it doesn't work properly

Developers experince it a lot



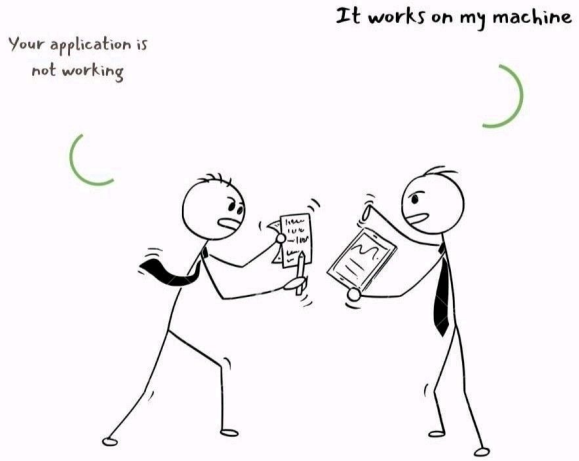
Source: https://www.linkedin.com/in/stevenouri/

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



DOCKER

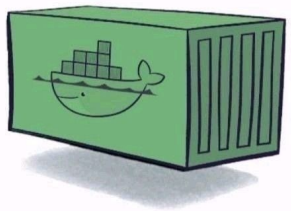
We need a standardized way to package the application with its **dependencies** and deploy it on any environment.

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

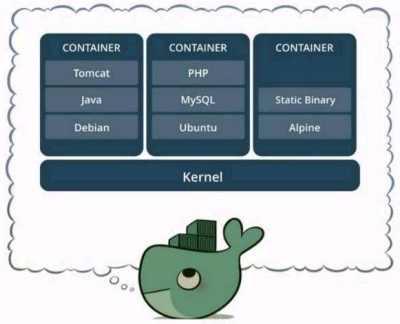
Source: https://www.linkedin.com/in/stevenouri/

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

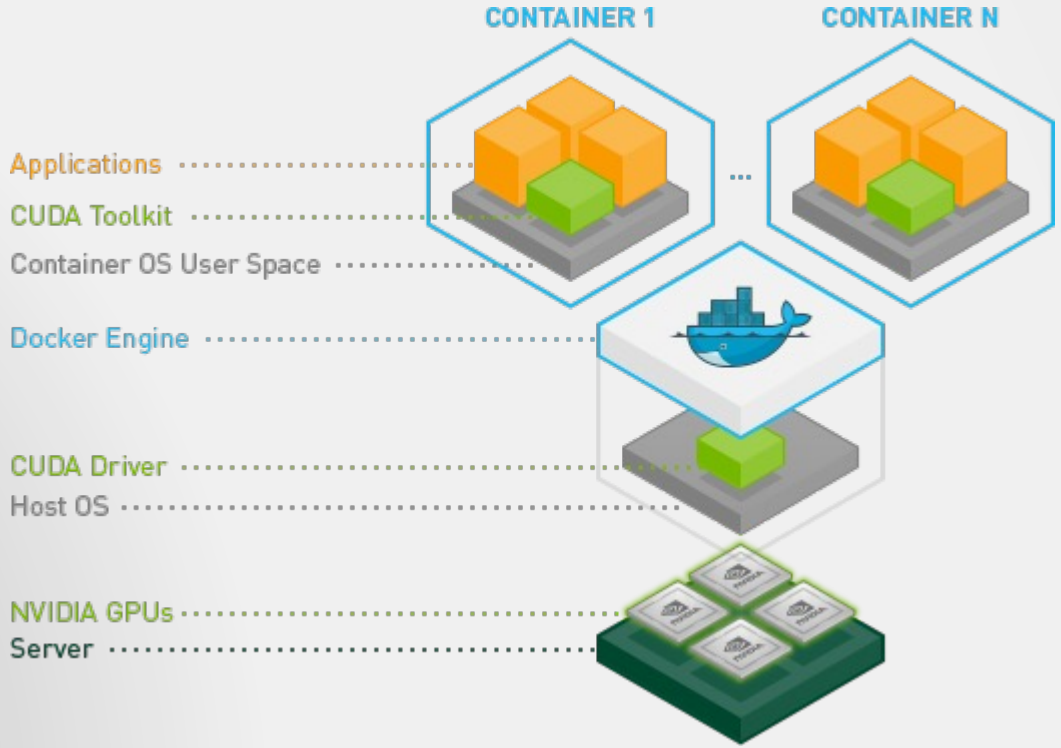
Source: https://www.linkedin.com/in/stevenouri/

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,:

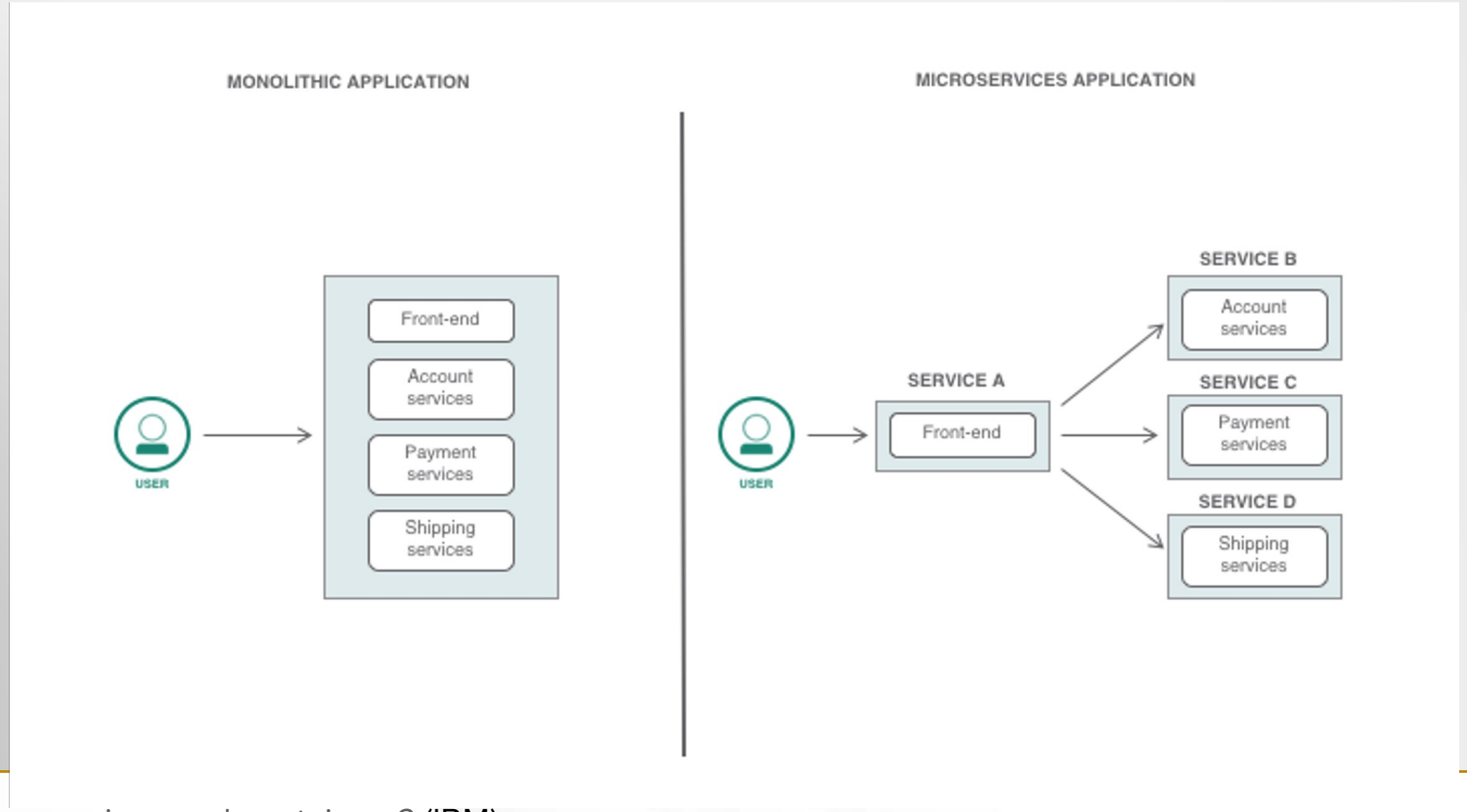
- Reproducible builds
- Ease of deployment
- Isolation of individual devices
- Run across heterogeneous driver/toolkit environments
- Requires only the NVIDIA driver to be installed
- Enables "fire and forget" GPU applications
- Facilitate collaboration

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

Container:

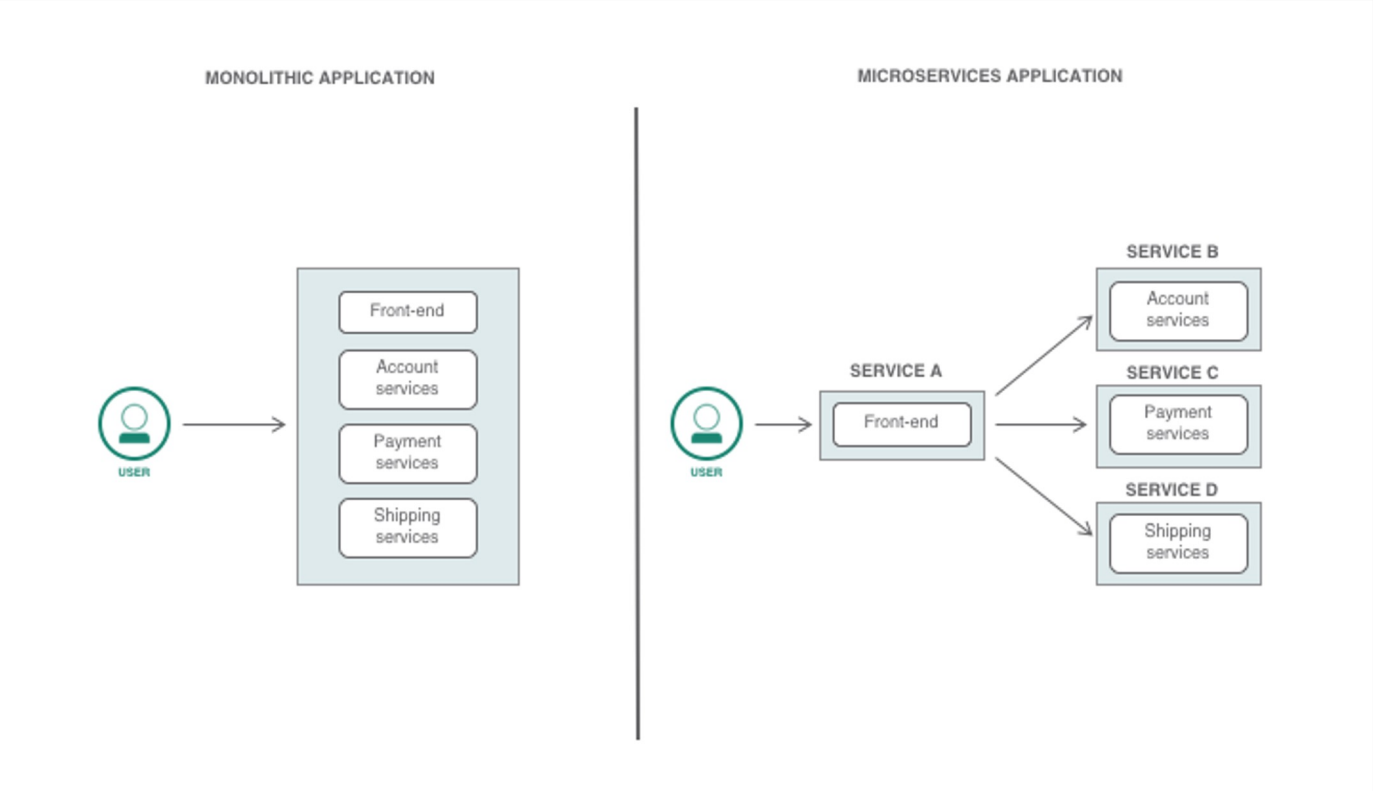
- Monolith: all application logics in one instance
- Microservices:
  - break application logics into smaller services
  - each service in its own container



Why should you use microservices and containers? (IBM)

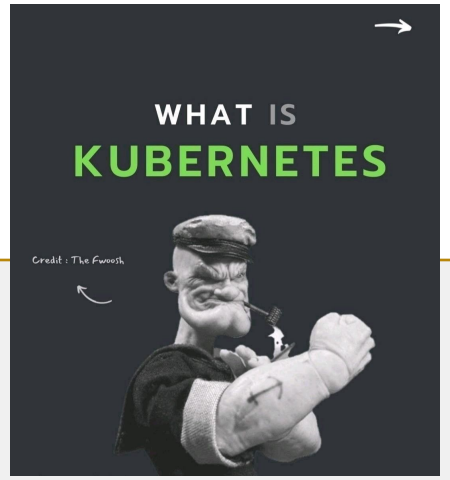
Container

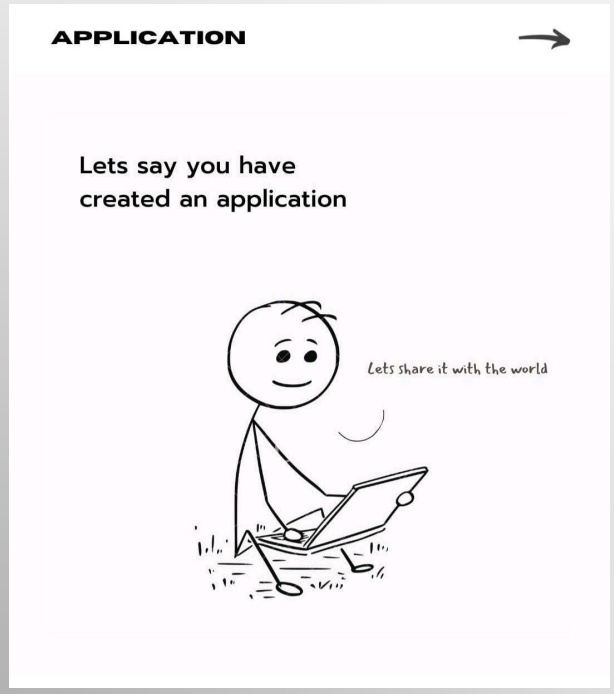
- **Reduced complexity**: each developer works on a smaller codebase
- **Faster development cycle**: easier review process
- **Flexible stack**: different microservices can use different technology stacks

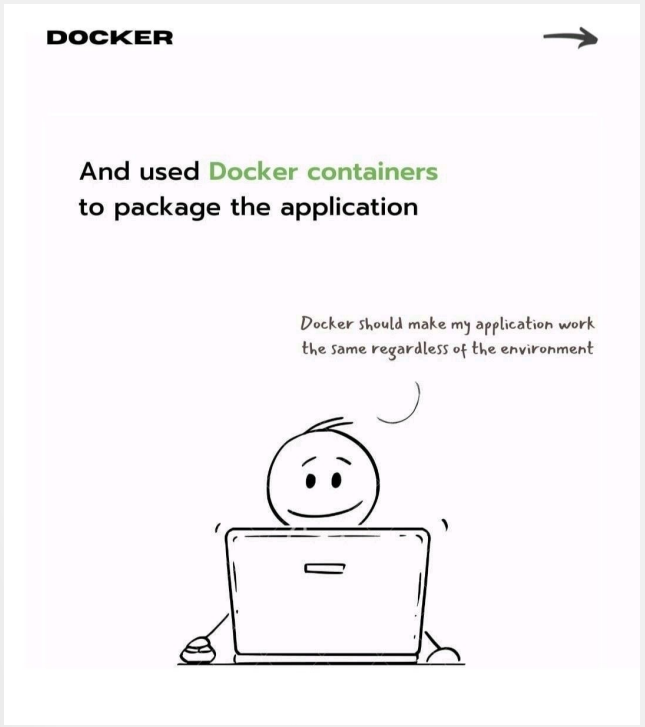


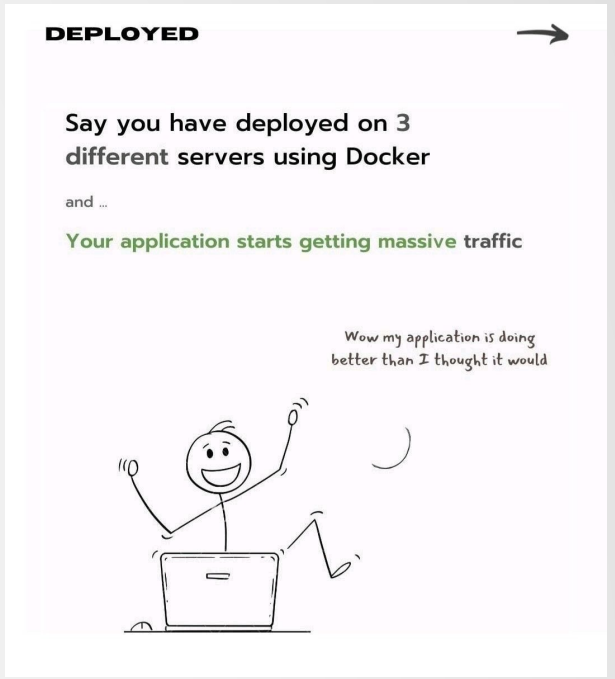
Why should you use microservices and containers? (IBM)

Kubernetes









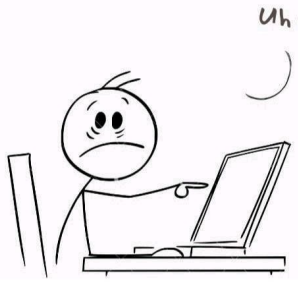
Source: https://www.linkedin.com/in/stevenouri/

Kubernetes

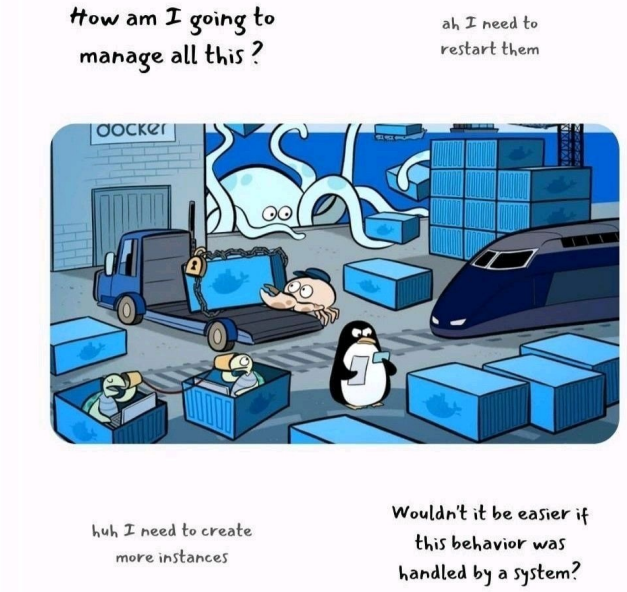
SCALING

Now you need to scale up fast; how will you go from 3 servers to 40 servers that you may require?

How to decide which container should go where? Monitor all containers ? & make sure they restart if they die?



OUT OF CONTROL



KUBERNETES

This is where Kubernetes comes into play

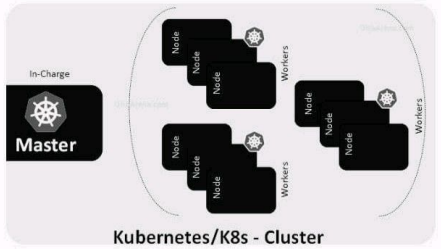
Kubernetes (aka k8s or "kube") is an open source **container orchestration** platform that automates deploying, managing, and scaling containerized applications.

Credit: Steve Nouri, https://www.linkedin.com/in/stevenouri/

Kubernetes

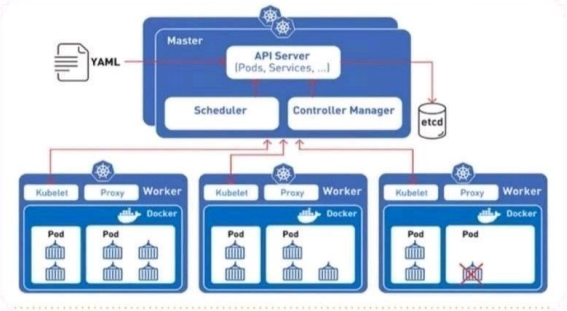
HOW IT WORKS ?

A Kubernetes cluster consists of a set of worker machines, called **nodes**, that run containerized applications



Every cluster has at least one **worker node**. Hence if a node fails, your application will still be accessible from the other nodes as in a cluster, **multiple** nodes are grouped.

ARCHITECTURE

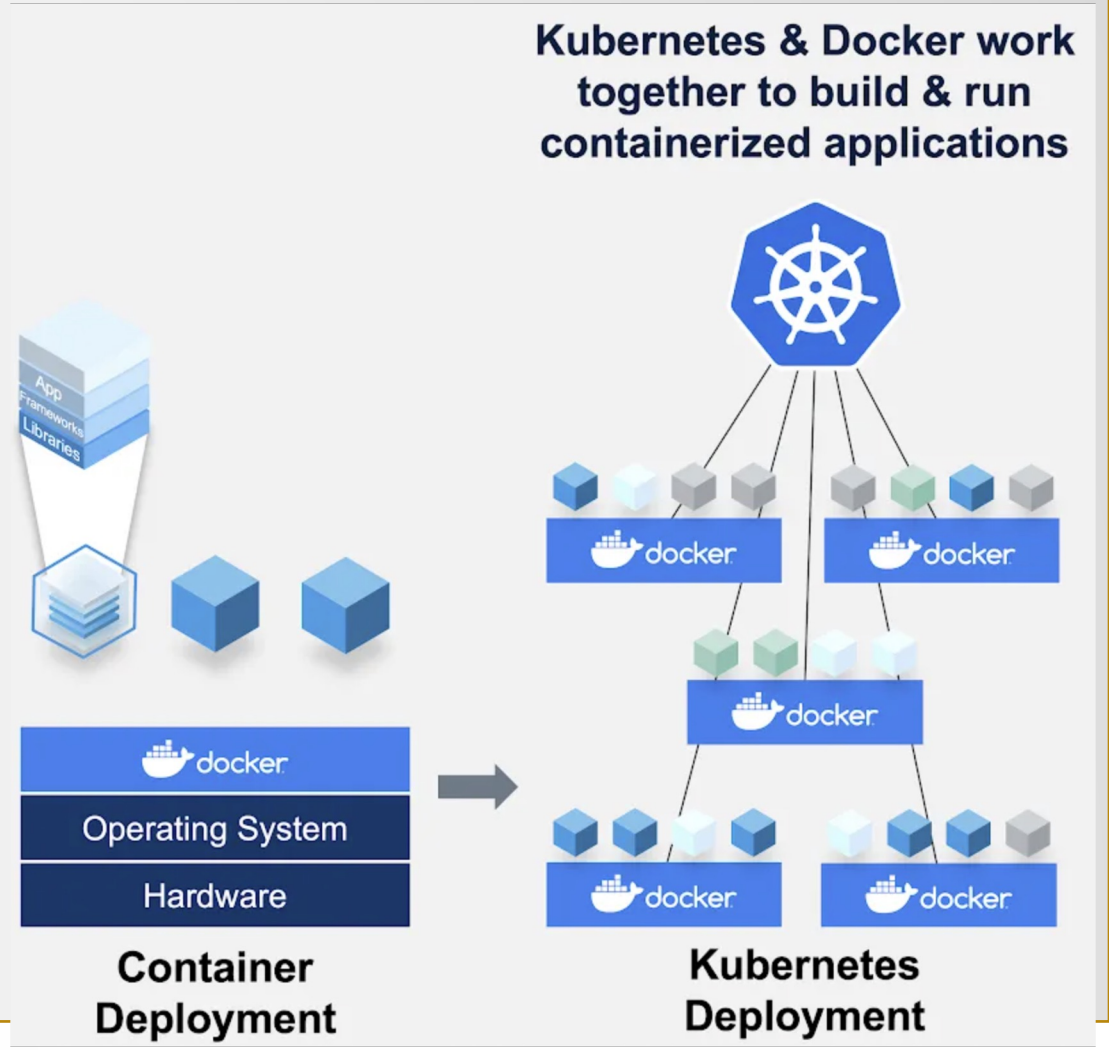


Every node contains a container runtime, **Kubelet** (for starting, stopping, and managing individual containers by requests from the Kubernetes control plane), and **kube-proxy** (for networking and load balancing).

Credit: Steve Nouri, https://www.linkedin.com/in/stevenouri/

Kubernetes

- How to manage multiple containers?
- If a container goes down, how to restart/replace it?
- How to allocate resources between containers?
- Security access
- …

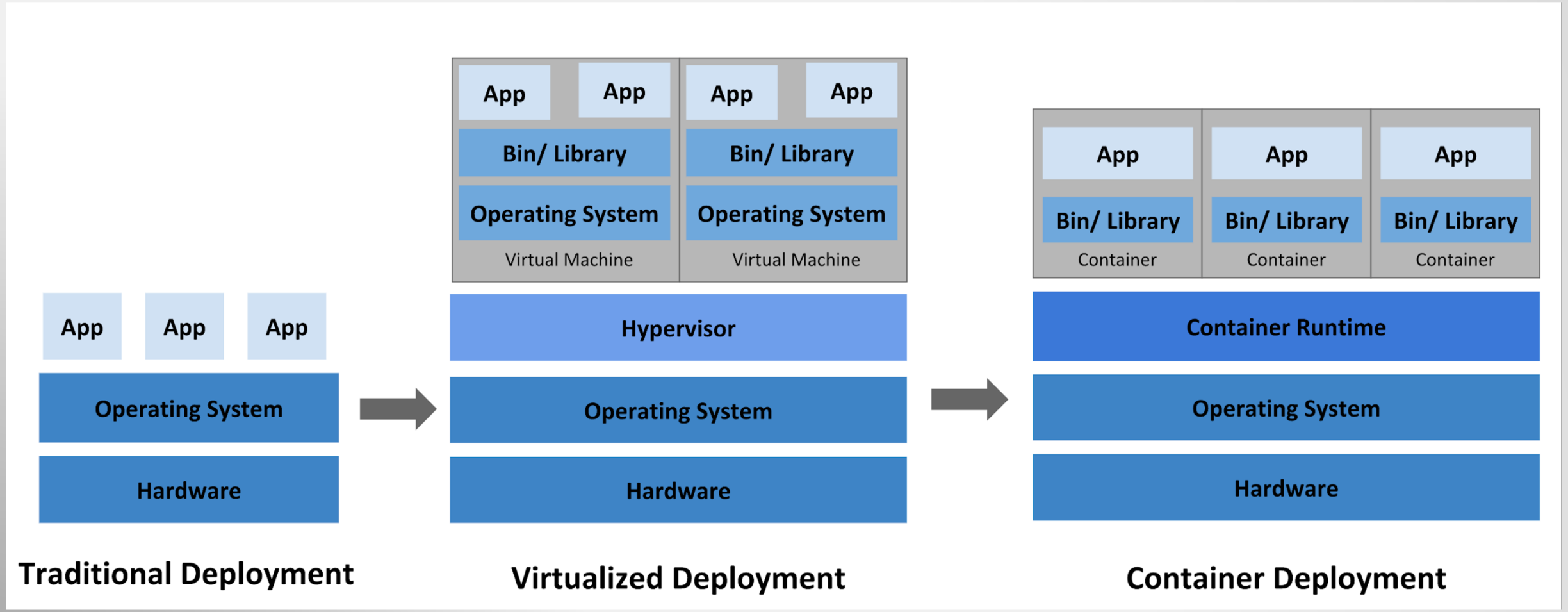


Top Questions Answered: Docker and Kubernetes? I Thought You Were Competitors! (Docker blog, 2019)

Kubeflow

- An open, community driven project to make it easy to deploy and manage an ML stack on Kubernetes
- Includes services to create and manage interactive Jupyter notebooks.
- TensorFlow training job operator can be used to handle distributed TensorFlow training jobs on various cluster sizes.
- TensorFlow Serving container can be used export trained TensorFlow models to Kubernetes.
- NVIDIA Triton Inference Server can be used for maximized GPU utilization, auto-scaling, traffic load balancing and redundancy/failover metrics.

Virtualizatio



https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

VMs vs. co

Virtual machine	**Container**
Full virtualization	OS-level virtualization
Simulates unmodified “guest” OS	Creates isolated user space on same OS
Can run any OS (e.g. Windows on Linux)	Can only accommodate different distributions of same OS
Heavy	Lightweight
Startup time: minutes	Startup time: seconds