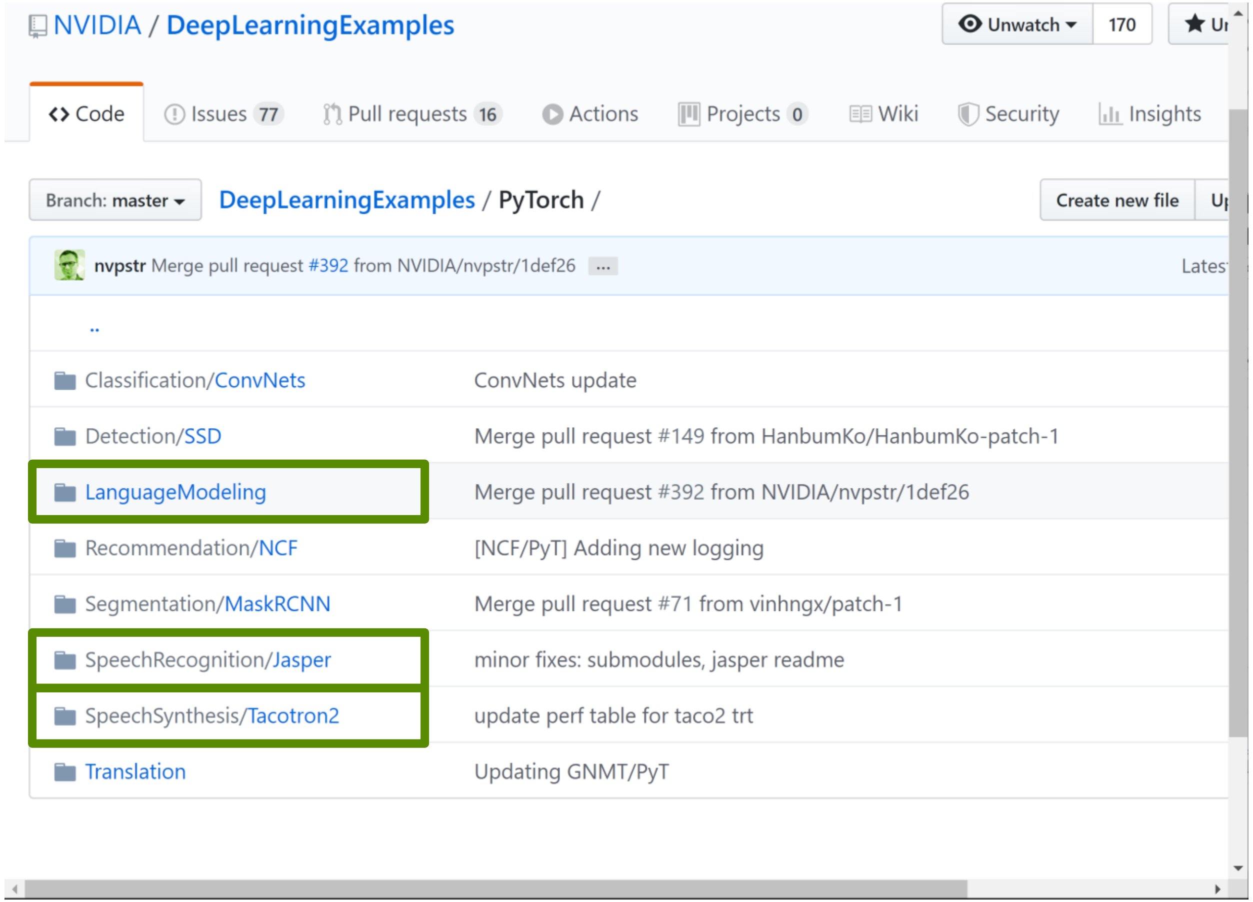
PYTORCH FROM RESEARCH TO PRODUCTION

Grzegorz Karch, GTC 2020

CONVERSATIONAL AI

PyTorch from Research to Production



speech_ai_demo.ipynb

Code | Python 3

Jasper input

RECORD | seconds 1.45

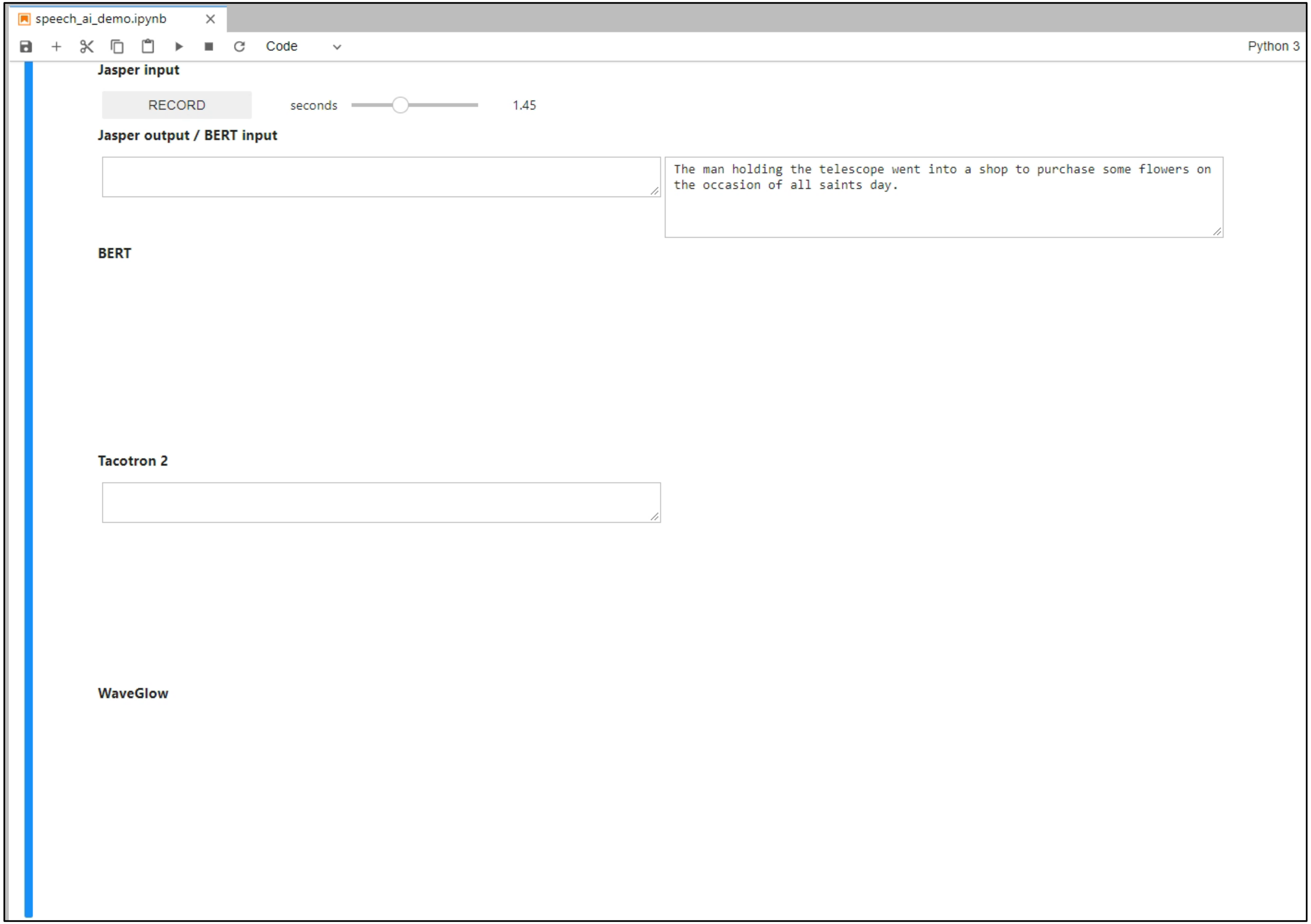
Jasper output / BERT input

```
The man holding the telescope went into a shop to purchase some flowers on
the occasion of all saints day.
```

BERT

Tacotron 2

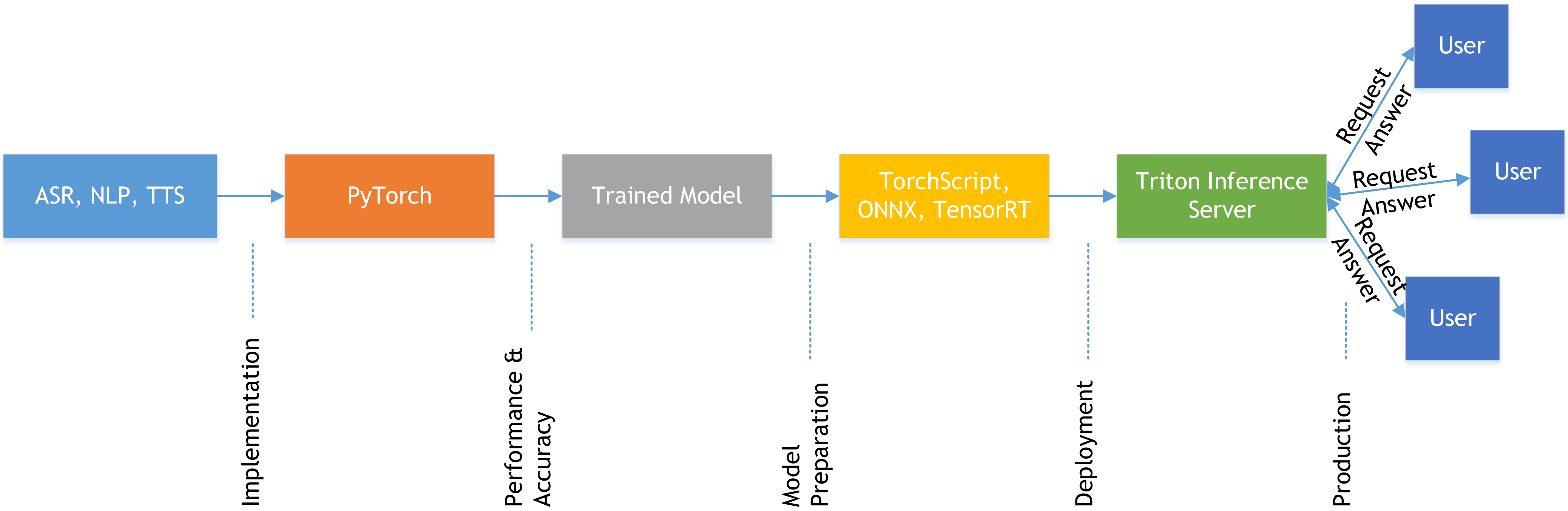
WaveGlow


speech_ai_demo.ipynb
Code
Python 3
Jasper input
RECORD
seconds
1.45
Jasper output / BERT input
The man holding the telescope went into a shop to purchase some flowers on the occasion of all saints day.
BERT
Tacotron 2
WaveGlow

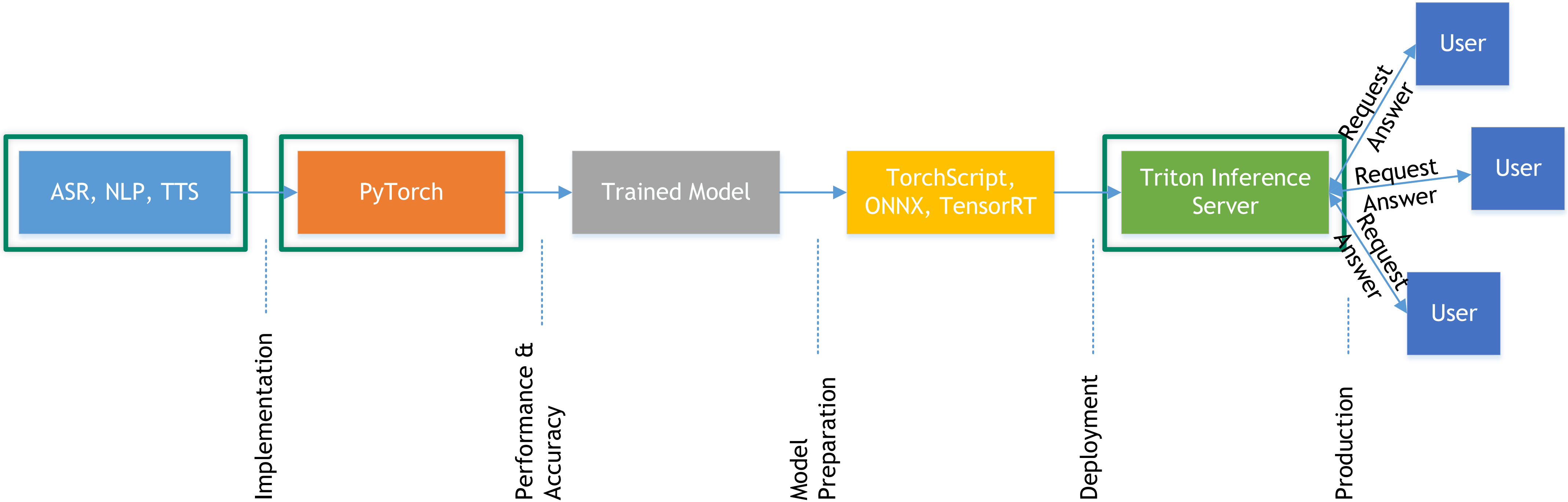
CONVERSATIONAL AI

PyTorch from Research to Production

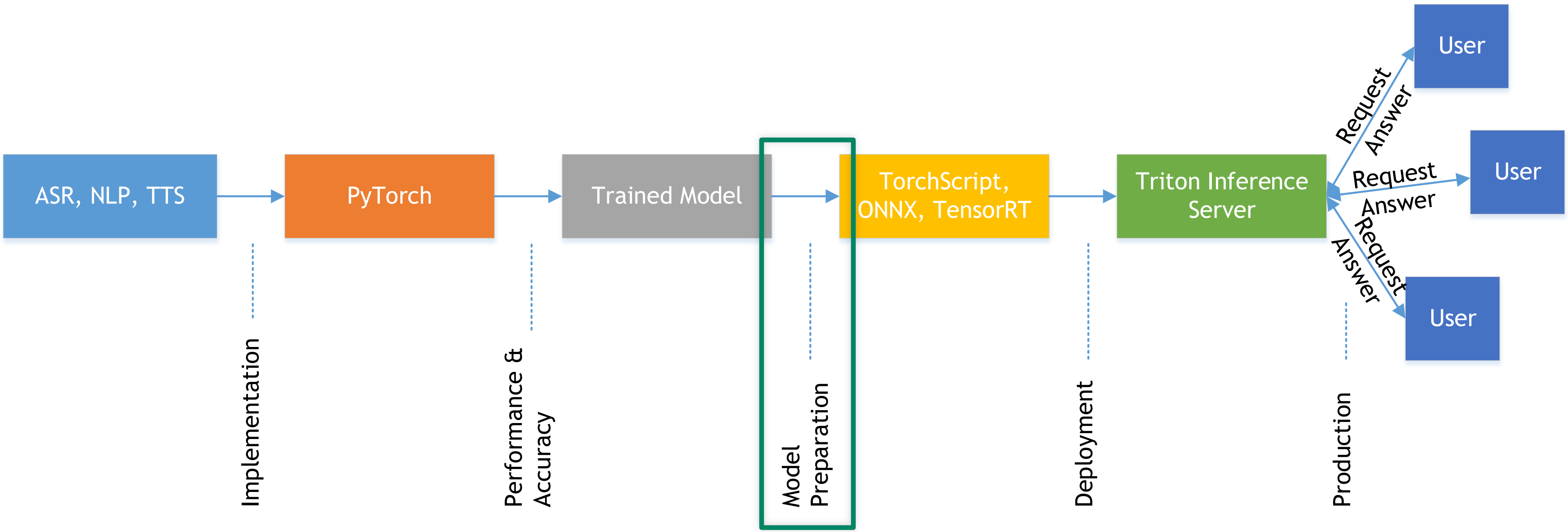
FROM RESEARCH TO PRODUCTION



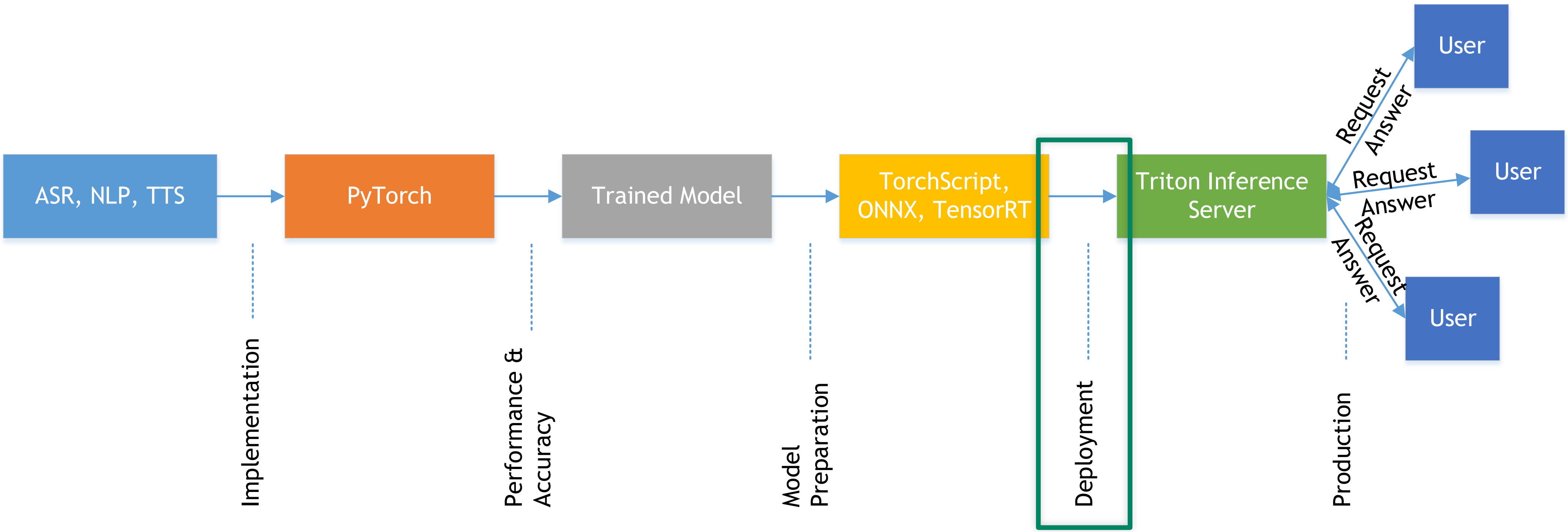
FROM RESEARCH TO PRODUCTION



FROM RESEARCH TO PRODUCTION



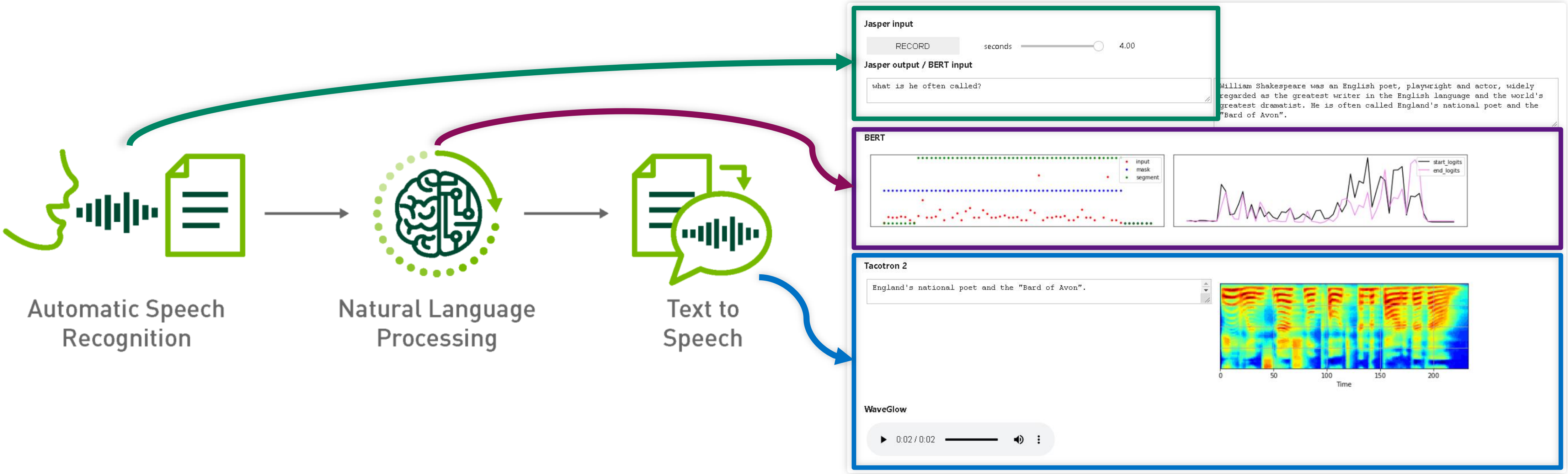
FROM RESEARCH TO PRODUCTION



INGREDIENTS

CONVERSATIONAL AI

PyTorch from Research to Production



CONVERSATIONAL AI

PyTorch from Research to Production

Jasper → BERT → Tacotron2 + WaveGlow

Automatic Speech Recognition

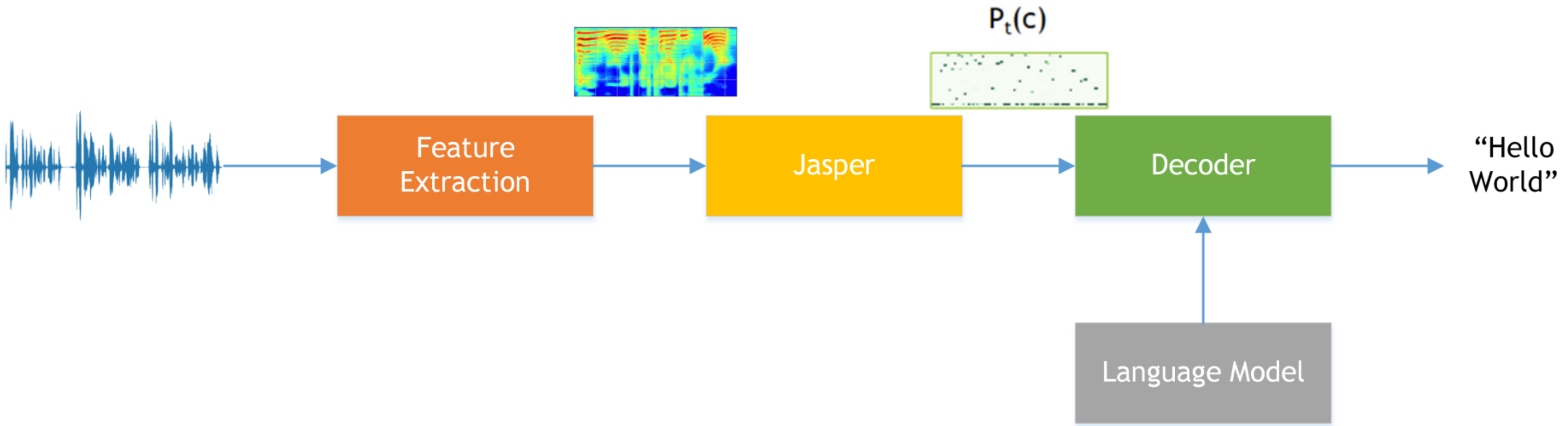
Natural Language Processing

Text to Speech

AUTOMATIC SPEECH RECOGNITION: JASPER

Conversational AI

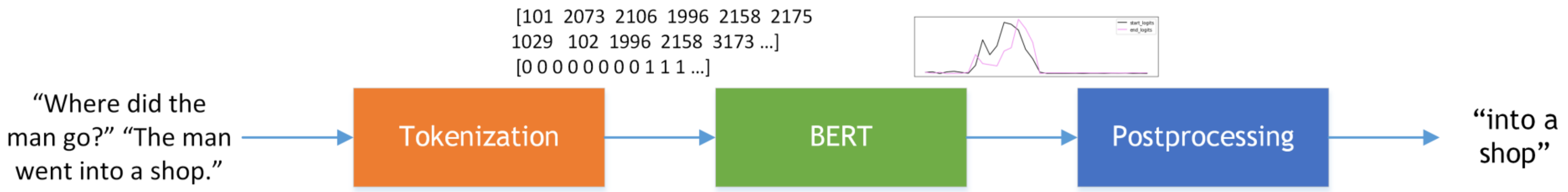
- Jasper [Li et al 2019]: End-to-end convolutional neural acoustic model



NATURAL LANGUAGE PROCESSING: BERT

Conversational AI

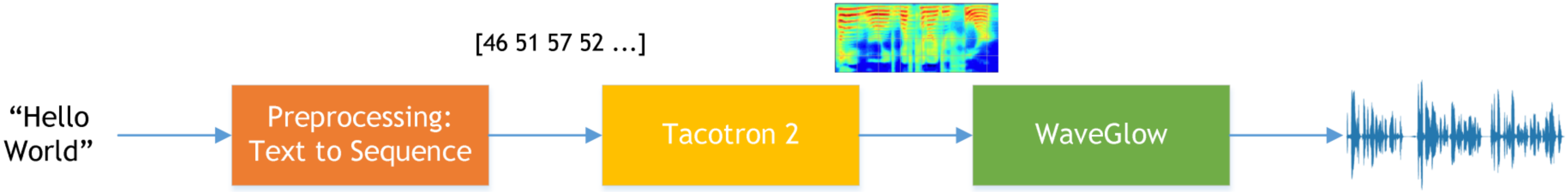
- BERT [Devlin et al 2018]: **B**idirectional **E**ncoder **R**epresentations from **T**ransformers



TEXT TO SPEECH: TACOTRON 2 + WAVEGLOW

Conversational AI

- Tacotron 2 [Shen et al 2018]: autoregressive sequence-to-sequence
- WaveGlow [Prenger et al 2018]: generative flow-based network



PYTORCH

PyTorch from research to production

Open source deep learning platform

- Intuitive API
- First-class Python support
- Gives you great freedom
- Research tool

Problem:

- Hard to deploy

```
class Sequence(nn.Module):
    # ...
    def forward(self, input):
        # ...
        for input_t in input.chunk(input.size(1), dim=1):
            h_t, c_t = self.lstm(input_t, (h_t, c_t))
            output = self.linear(h_t)
            outputs += [output]
        outputs = torch.stack(outputs, 1).squeeze(2)
        return outputs
```

https://github.com/pytorch/examples/blob/master/time_sequence_prediction/train.py

PYTORCH

PyTorch from research to production

Solution:

TorchScript – statically typed subset of Python

- Potent at inference
- „Compiles" the model to a C++ library
- Optimizes model graph, tensor operations
- Limited freedom of development

```
class Sequence(nn.Module):
    # ...
    def forward(self, input):
        # ...
        for input_t in input.chunk(input.size(1), dim=1):
            h_t, c_t = self.lstm(input_t, (h_t, c_t))
            output = self.linear(h_t)
            outputs += [output]
        outputs = torch.stack(outputs, 1).squeeze(2)
        return outputs
```

https://github.com/pytorch/examples/blob/master/time_sequence_prediction/train.py

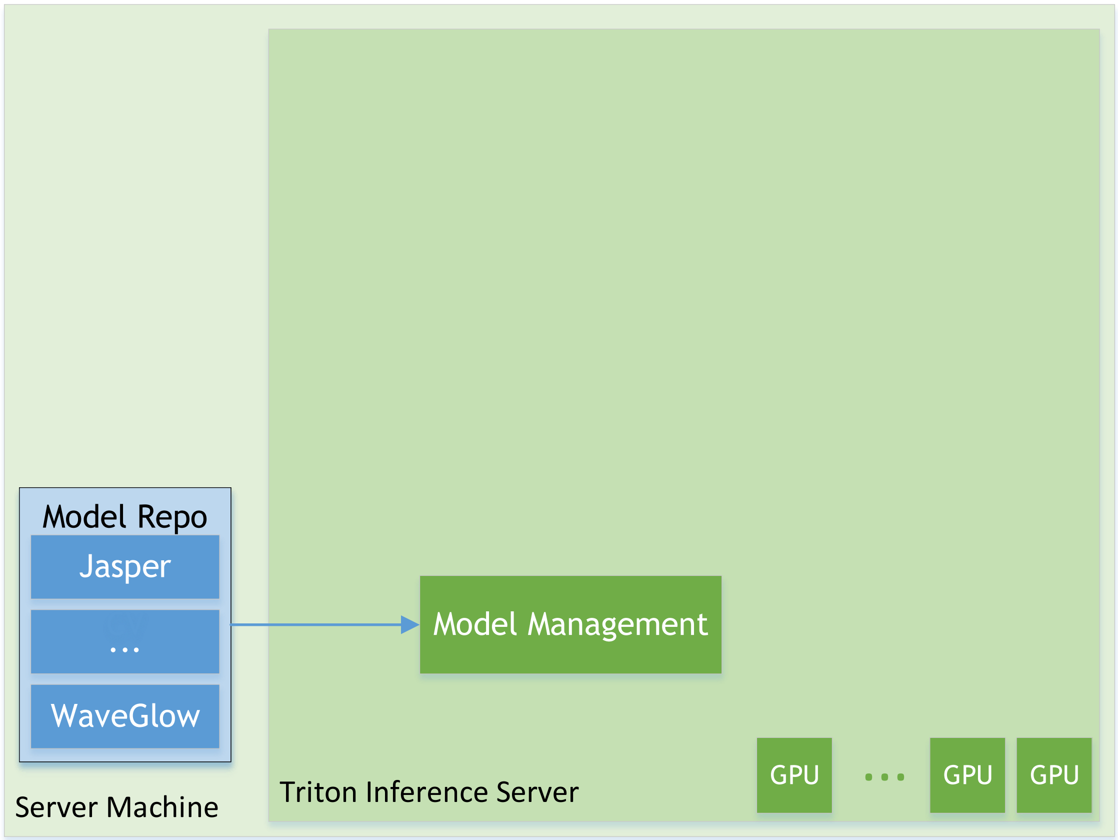
TRITON INFERENCE SERVER

PyTorch from research to production

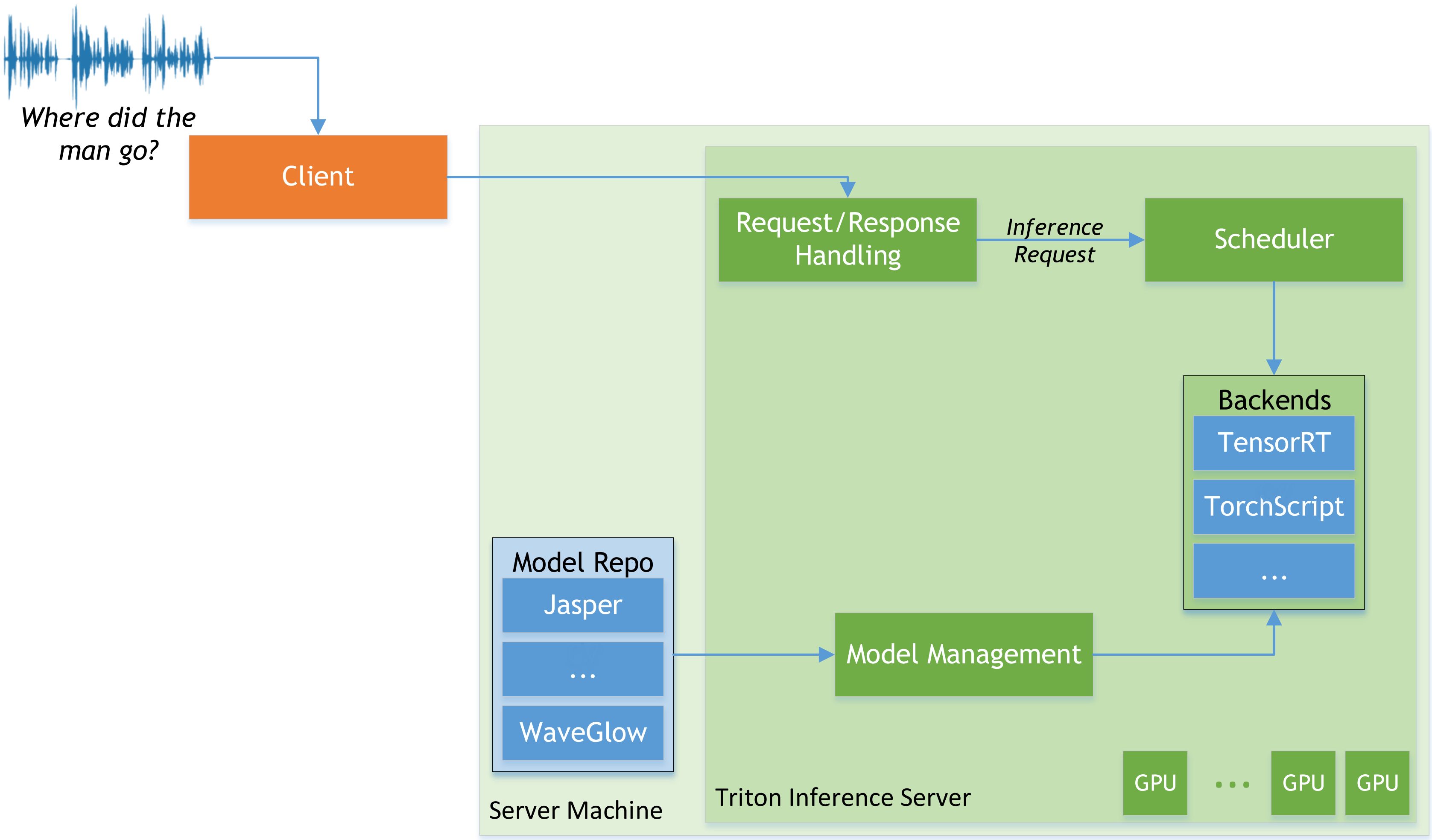
Why Triton Inference Server?

- Supports various backends
- Ensemble support
- Concurrent model execution
- Dynamic scheduling and batching
- Open source https://github.com/NVIDIA/tensorrt-inference-server

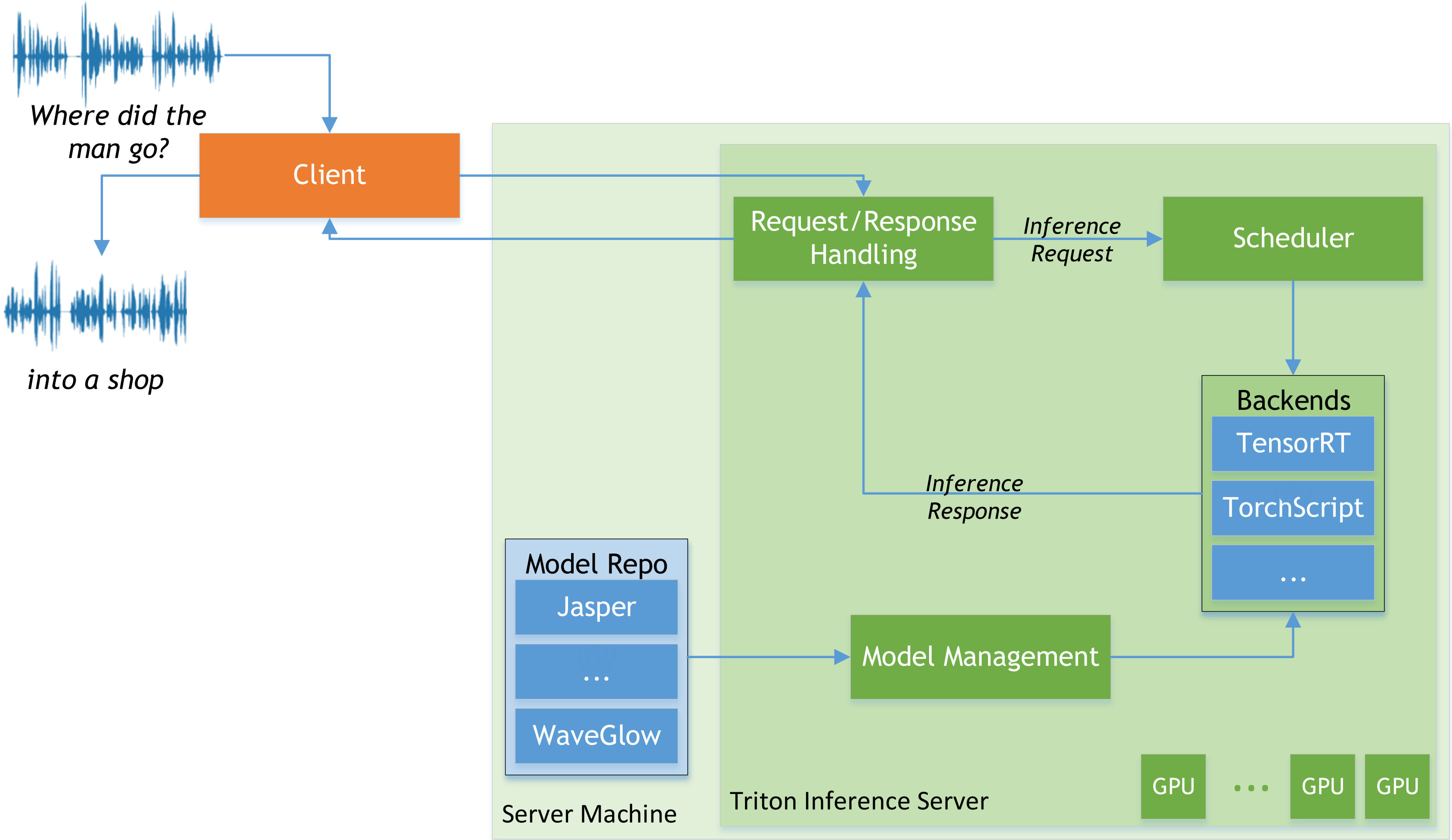
TRITON INFERENCE SERVER



TRITON INFERENCE SERVER



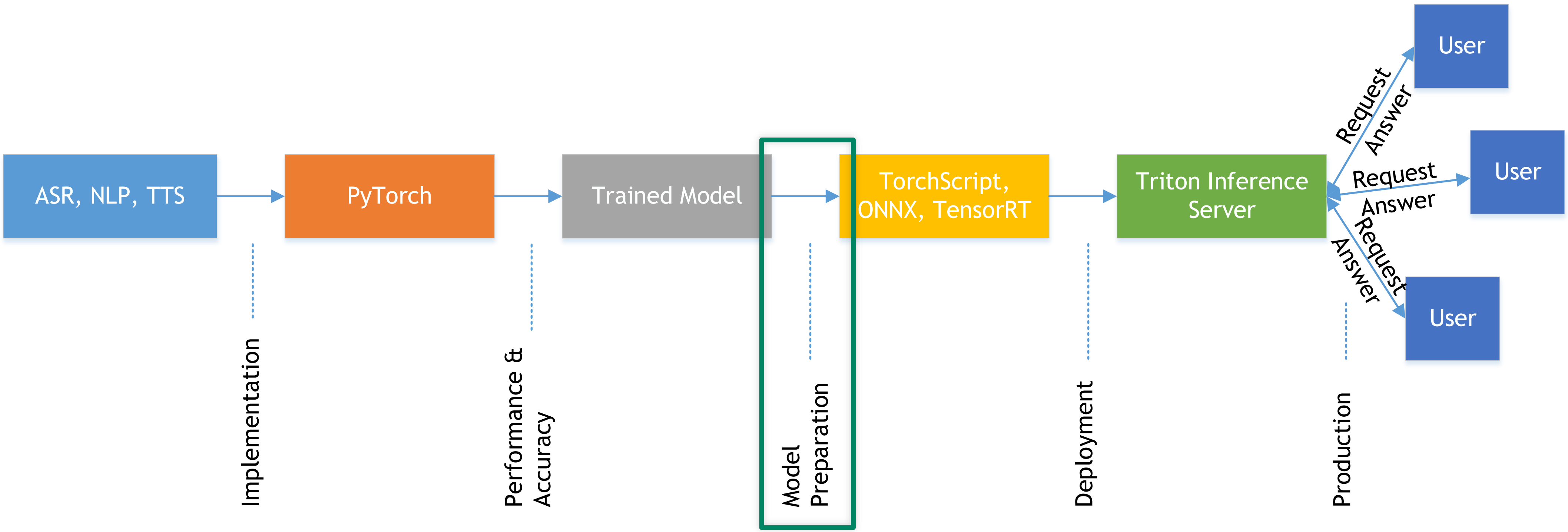
TRITON INFERENCE SERVER



NVIDIA

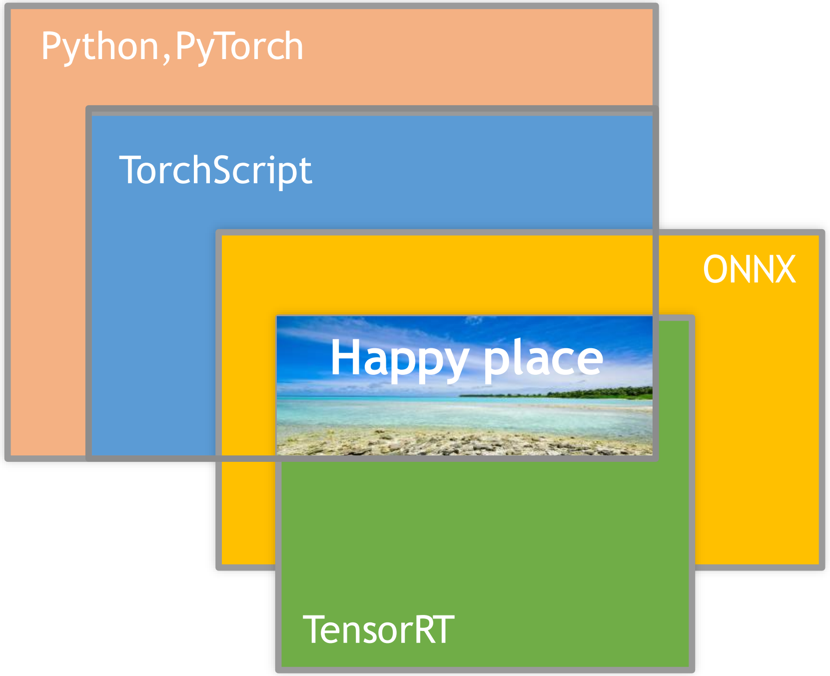
MODEL PREPARATION

FROM RESEARCH TO PRODUCTION



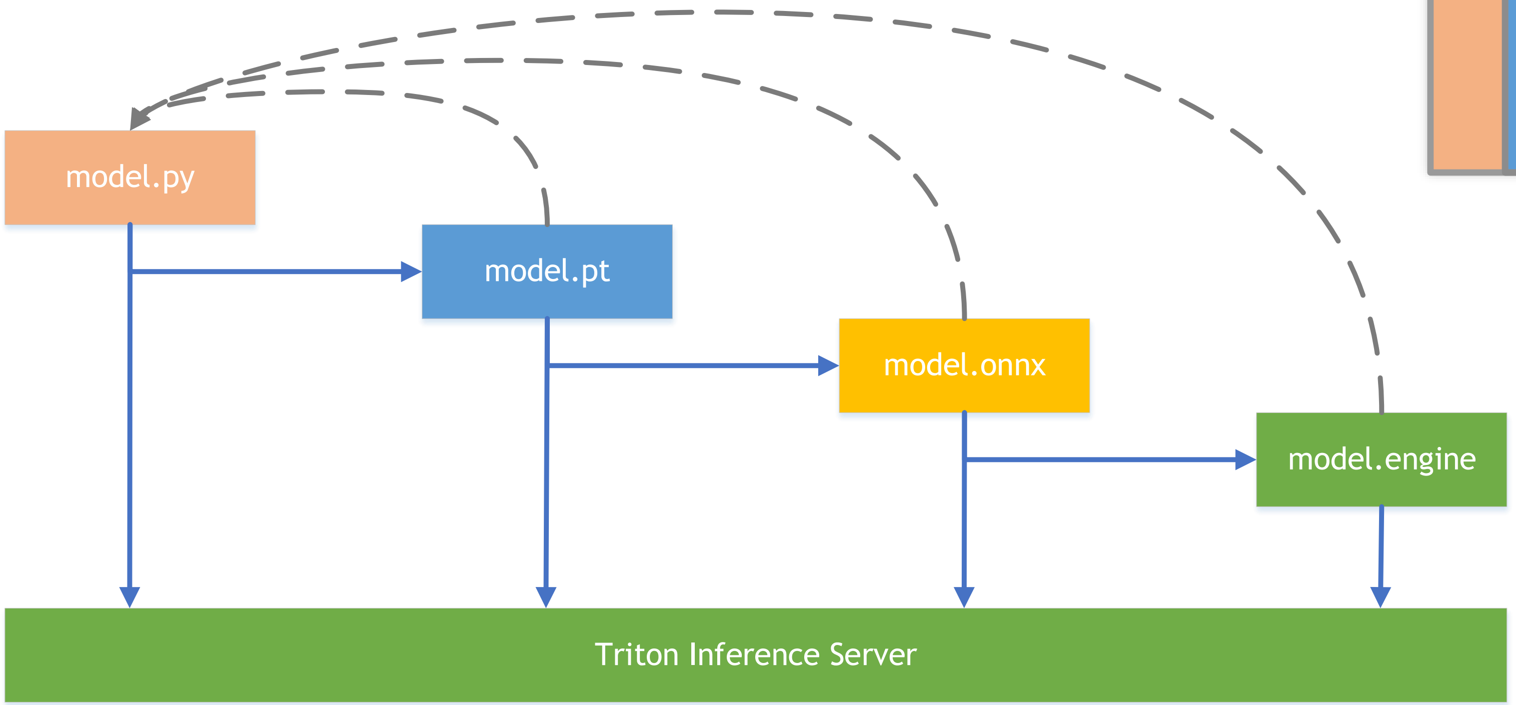
MODEL PREPARATION

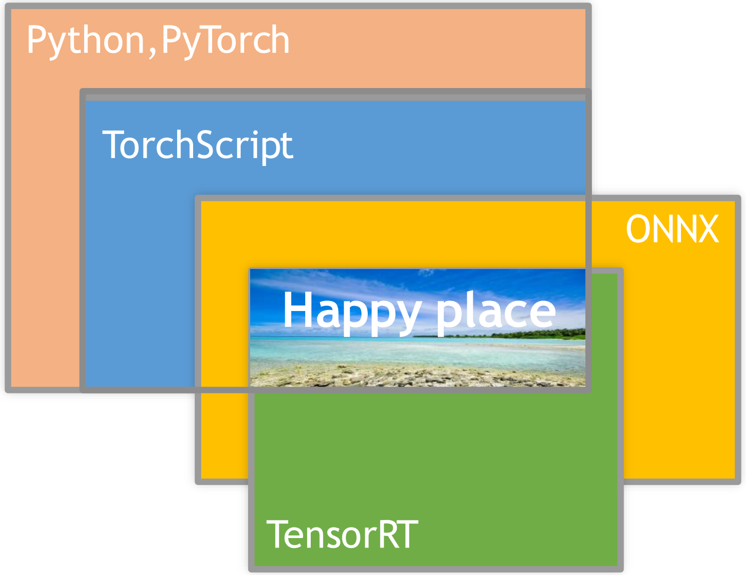
PyTorch from research to production



MODEL PREPARATION

PyTorch from research to production





TRACING VS SCRIPTING

Model Preparation

Tracing (`torch.jit.trace`, `torch.onnx.export`)

- Runs on example input and records operations
- No control-flow

```
torch.onnx.export(model, (mel, z), "waveglow.onnx",
                  input_names=["mel", "z"], output_names=["audio"],
                  dynamic_axes={"mel": {0: "batch_size", 2: "mel_seq"},
                                ...})
```

TRACING VS SCRIPTING

Model Preparation

Tracing (`torch.jit.trace`, `torch.onnx.export`)

- Runs on example input and records operations
- No control-flow

```
torch.onnx.export(model, (mel, z), "waveglow.onnx",
                  input_names=["mel", "z"], output_names=["audio"],
                  dynamic_axes={"mel": {0: "batch_size", 2: "mel_seq"},
                                ...})
```

Scripting (`torch.jit.script`)

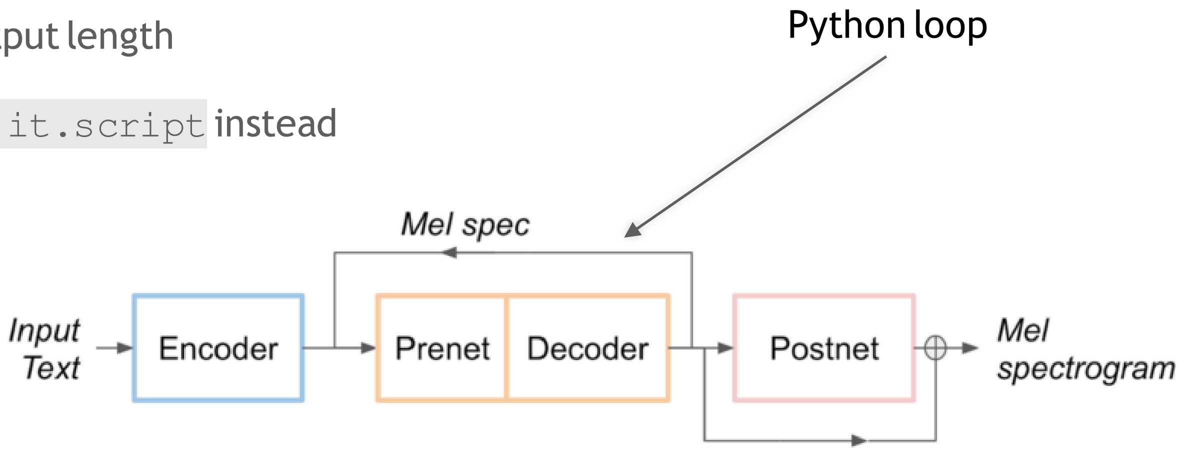
- Analyzes Python source code
- Captures control-flow operations

```
model = Tacotron2(**config)
jitted_model = torch.jit.script(model)
torch.jit.save(jitted_model, "tacotron2.pt")
```

TACOTRON 2

Model Preparation

- Ideally, convert to TensorRT via `torch.onnx.export`
- Problem: would fix output length
- Solution: use `torch.jit.script` instead



Simplified Tacotron 2 architecture

TACOTRON 2 → TORCHSCRIPT

Model Preparation

Problem: `torch.jit.script` will compile the forward method but

- Forward pass ≠ Inference for Tacotron 2
- `forward()` needed by LibTorch backend in Triton

Solution: wrapper classes

```
def get_model(model_config, ..., forward_is_infer=False):
    if forward_is_infer:
        class Tacotron2_forward_is_infer(Tacotron2):
            def forward(self, inputs, input_lengths):
                return self.infer(inputs, input_lengths)
        model = Tacotron2_forward_is_infer(**model_config)
    else:
        model = Tacotron2(**model_config)
```

TACOTRON 2 → TORCHSCRIPT

Model Preparation

Problem: Tensor members cannot be created outside __init__() method:

```
Tried to set nonexistent attribute: attention_hidden. Did you forget to initialize it in __init__()?:
Is it in the documentation? Or can I use register_buffer?
```

Solution: State tensors must be treated as local variables

```
(mel_output, gate_output,
 attention_hidden,attention_cell,...) =
                              self.decode(decoder_input, attention_hidden, attention_cell,
                                          ..., memory, processed_memory, mask)
```

```
model = Tacotron2(**config)
jitted_model = torch.jit.script(model)
torch.jit.save(jitted_model, „tacotron2.pt")
```

WAVEGLOW → ONNX → TENSORRT

Model Preparation

Problem: WaveGlow creates matrix inverses on the fly

```
class Invertible1x1Conv(torch.nn.Module):
    # ...
    def forward(self, z):
        W = self.conv.weight.squeeze()
        if not hasattr(self, 'W_inverse'):
            W_inverse = W.float().inverse()
            self.W_inverse = W_inverse
        z = F.conv1d(z, self.W_inverse, bias=None, stride=1, padding=0)
```

Solution: Pre-launch WaveGlow inference to initialize inverses

```
waveglow.infer(mel, sigma=args.sigma_infer)
# ...
torch.onnx.export(waveglow, (mel, ...)
```

WAVEGLOW → ONNX → TENSORRT

Model Preparation

Problem: ONNX doesn’t support 1D convolutions

Solution: Convert 1D convs to 2D convs

```
def convert_conv_1d_to_2d(conv1d):
    conv2d = torch.nn.Conv2d(in_channels=conv1d.weight.size(1),
                             out_channels=conv1d.weight.size(0),
                             kernel_size=(conv1d.weight.size(2), 1),
                             ...)
    conv2d.weight.data[:,:,:,0] = conv1d.weight.data
    conv2d.bias.data = conv1d.bias.data
    return conv2d
```

WAVEGLOW → ONNX → TENSORRT

Model Export

WaveGlow PyTorch → ONNX:

```
torch.onnx.export(waveglow, (mel, z), "waveglow.onnx",
                  # ...,
                  input_names=["mel", "z"], output_names=["audio"],
                  dynamic_axes={"mel": {0: "batch_size", 2: "mel_seq"},
                                "z": {0: "batch_size", 2: "z_seq"},
                                "audio": {0: "batch_size", 1: "audio_seq"}})
```

ONNX → TensorRT engine

```
network = builder.create_network(explicit_batch)
with trt.OnnxParser(network, TRT_LOGGER) as parser:
    with open("waveglow.onnx", 'rb') as model:
        parser.parse(model.read())
        engine = builder.build_engine(network, config=config)
```

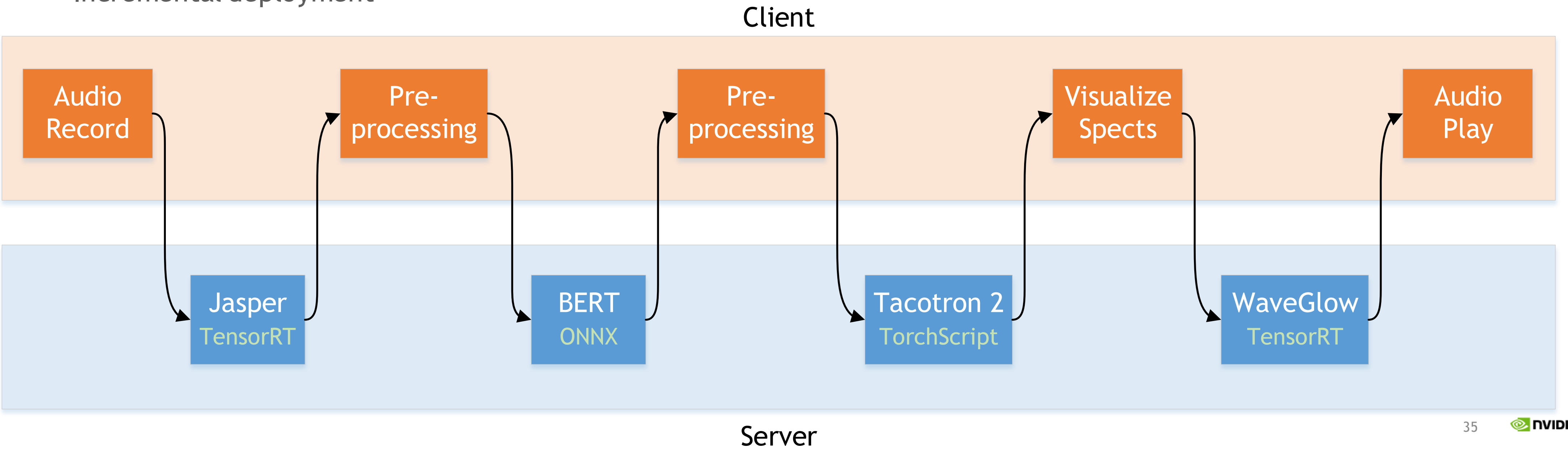
DEPLOYMENT

TENSORRT INFERENCE SERVER: CLIENT-SERVER

Deployment

For deployment process, use client-server communication

- Easy debugging
- Incremental deployment

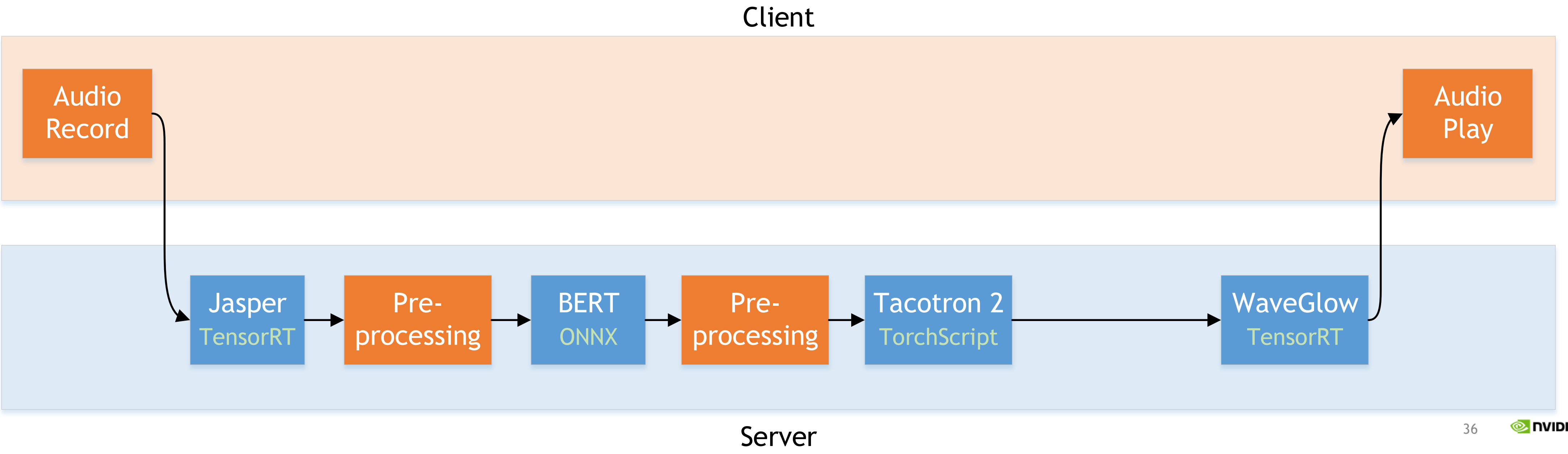


TENSORRT INFERENCE SERVER: CLIENT-SERVER

Deployment

In production create ensemble instead

- Will enable all Triton features (dynamic batching, queuing, better performance, etc.)



DEPLOYMENT

PyTorch from Research to Production

Start TensorRT Inference Server

```
NV_GPU=1 nvidia-docker run -ti --ipc=host --network=host
--rm -p8000:8000 -p8001:8001 \
-v /home/grzegorz/convai/model_repo/:/models \
nvcr.io/nvidia/tensorrtserver:20.01-py3 trtserver \
--model-store=/models
```

```
model_repo
├── bert-trt
│   ├── 1
│   │   └── model.plan
│   └── config.pbtxt
...
├── tacotron2
│   ├── 1
│   │   └── model.pt
│   └── config.pbtxt
└── waveglow-trt
    ├── 1
    │   └── waveglow_fp16.engine
    └── config.pbtxt
```

DEPLOYMENT

PyTorch from Research to Production

Start client

```
docker run -it --rm --network=host speech_ai_client:demo
```

SIMPLE CLIENT – TACOTRON 2

Deployment

config.pbtxt

```
name: "tacotron2"
platform: "pytorch_libtorch"
max_batch_size: 8
input [{
    name: "sequence__0"
    data_type: TYPE_INT64
    dims: [-1]
  },
  {
    name: "input_lengths__1"
    data_type: TYPE_INT64
    dims: [1]
    reshape: { shape: [ ] }
  }]
output [{
    name: "mel_outputs_postnet__0"
    data_type: TYPE_FP32
    dims: [80,-1]
  },
  {
    name: "mel_lengths__1"
    data_type: TYPE_INT32
    dims: [1]
    reshape: { shape: [ ] }
  }]
```

Start client

```
docker run -it --rm --network=host speech_ai_client:demo
```

```
url = 'localhost:8000'

infer_ctx_tacotron2 = InferContext(url, 0, 'tacotron2', -1)

sequence = np.array(input, dtype=np.int64)
input_lengths = np.array(len(input), dtype=np.int64)

input_dict = {'sequence__0':(sequence,),
              'input_lengths__1':(input_lengths,)}
output_dict = {'mel_outputs_postnet__0':InferContext.ResultFormat.RAW,
               'mel_lengths__1':InferContext.ResultFormat.RAW}

result = infer_ctx_tacotron2.run(input_dict, output_dict, 1)
```

client.py

DEMO

Jasper input

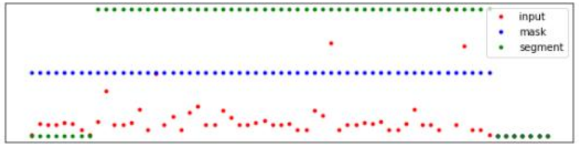
RECORD seconds 4.00

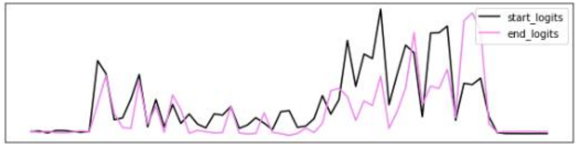
Jasper output / BERT input

```
what is he often called?
```

```
William Shakespeare was an English poet, playwright and actor, widely
regarded as the greatest writer in the English language and the world's
greatest dramatist. He is often called England's national poet and the
"Bard of Avon".
```

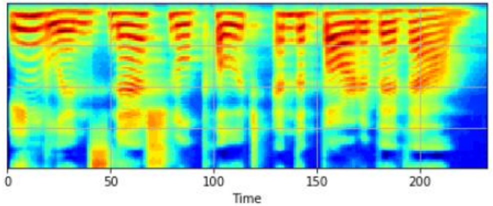
BERT





Tacotron 2

```
England's national poet and the "Bard of Avon".
```



WaveGlow

0:02 / 0:02

DEMO

Try it yourself!

https://github.com/NVIDIA/DeepLearningExamples/PyTorch/SpeechSynthesis/Tacotron2/notebooks/conversationalai

DEPLOYER

Pytorch from Research to Production

- Contains boilerplate code for export to ONNX/TorchScript tracing and scripting
- Generates model based on provided data input
- Test correctness against original PyTorch model
- Available on GitHub

```
python deployer.py --ts-trace --triton-model-name=<name> -- --checkpoint=<path to your checkpoint>
```

DEPLOYING

PyTorch from Research to Production

NeMo: toolkit for defining and building models for Conversational AI applications

https://nvidia.github.io/NeMo/

Jarvis: comprehensive workflow to build, train and deploy AI systems

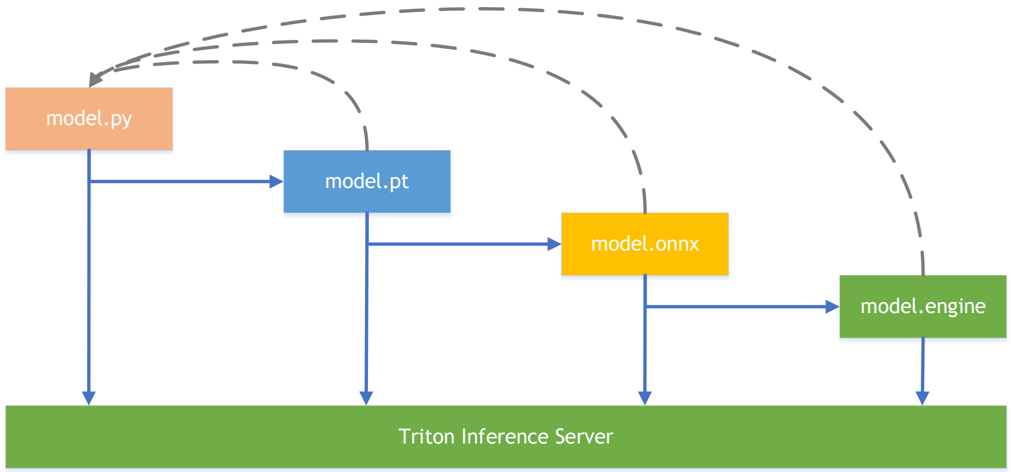
https://developer.nvidia.com/nvidia-jarvis

CONCLUSION

PyTorch from Research to Production

Lessons learned

- Plan ahead
- Continuously check the exports
- Triton allows incremental deployment
- Engage! PyTorch, Triton, ONNX forums/issues

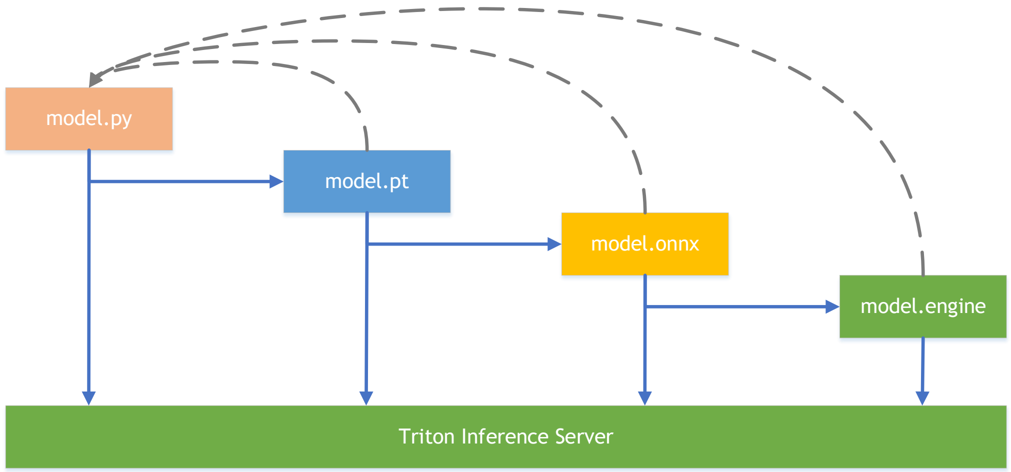


CONCLUSION

PyTorch from Research to Production

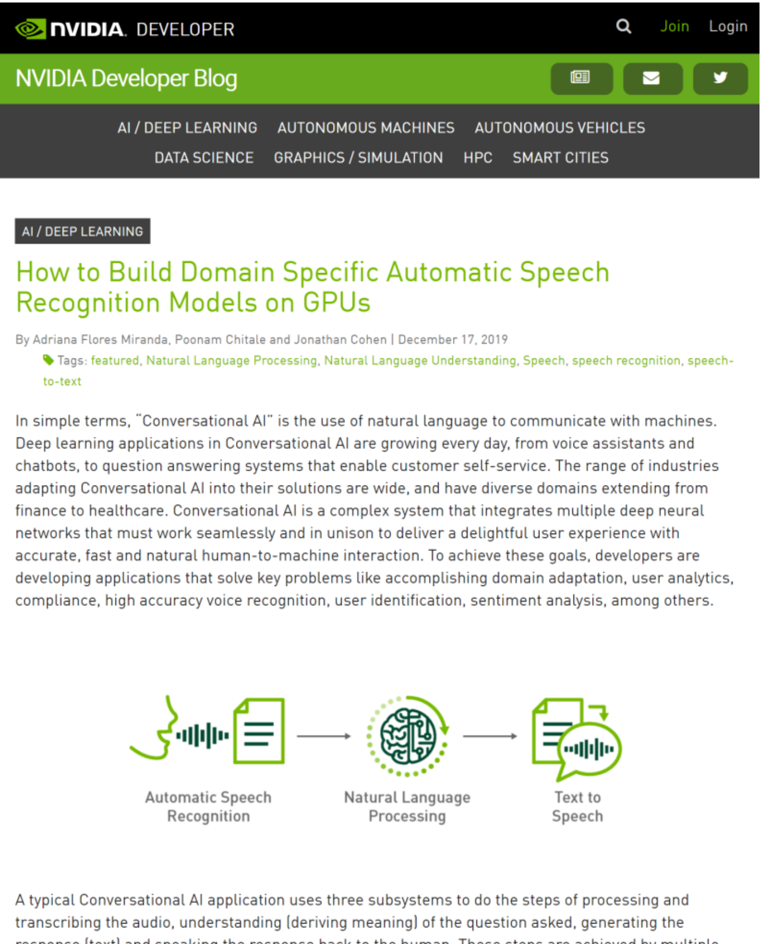
Summary – Conversational AI

- Already great results
- In future – use ensemble, reduce communication
- Extend the demos



RESOURCES

PyTorch from Research to Production



NVIDIA DEVELOPER Join Login

NVIDIA Developer Blog

AI / DEEP LEARNING AUTONOMOUS MACHINES AUTONOMOUS VEHICLES DATA SCIENCE GRAPHICS / SIMULATION HPC SMART CITIES

AI / DEEP LEARNING

How to Build Domain Specific Automatic Speech Recognition Models on GPUs

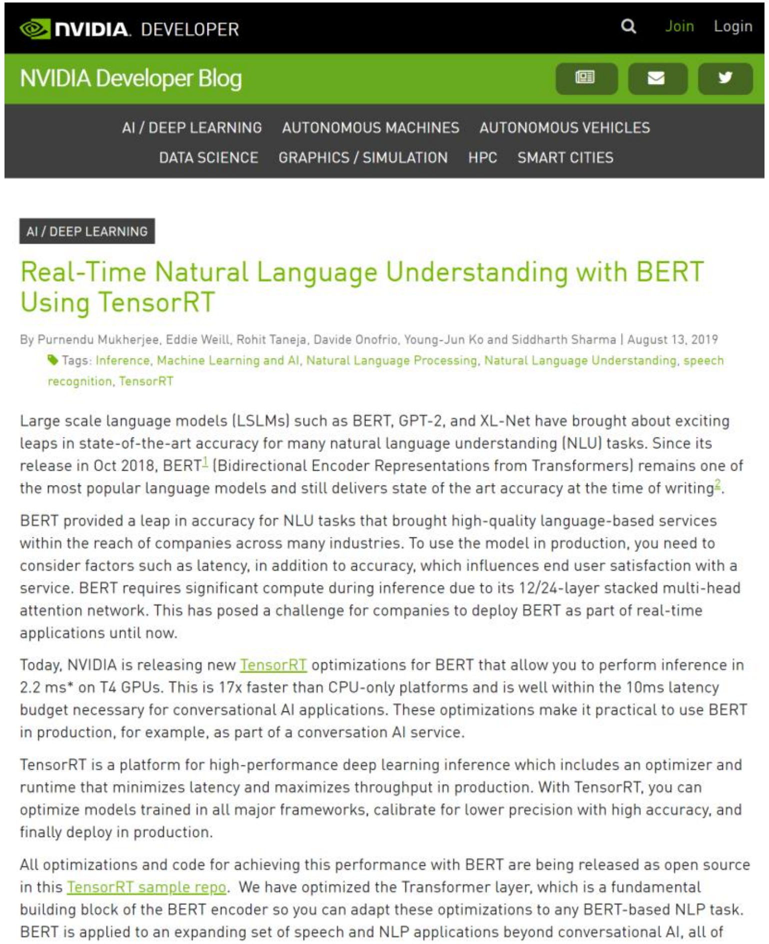
By Adriana Flores Miranda, Poonam Chitale and Jonathan Cohen | December 17, 2019

Tags: featured, Natural Language Processing, Natural Language Understanding, Speech, speech recognition, speech-to-text

In simple terms, "Conversational AI" is the use of natural language to communicate with machines. Deep learning applications in Conversational AI are growing every day, from voice assistants and chatbots, to question answering systems that enable customer self-service. The range of industries adapting Conversational AI into their solutions are wide, and have diverse domains extending from finance to healthcare. Conversational AI is a complex system that integrates multiple deep neural networks that must work seamlessly and in unison to deliver a delightful user experience with accurate, fast and natural human-to-machine interaction. To achieve these goals, developers are developing applications that solve key problems like accomplishing domain adaptation, user analytics, compliance, high accuracy voice recognition, user identification, sentiment analysis, among others.

Automatic Speech Recognition
Natural Language Processing
Text to Speech

A typical Conversational AI application uses three subsystems to do the steps of processing and transcribing the audio, understanding (deriving meaning) of the question asked, generating the



NVIDIA DEVELOPER Join Login

NVIDIA Developer Blog

AI / DEEP LEARNING AUTONOMOUS MACHINES AUTONOMOUS VEHICLES DATA SCIENCE GRAPHICS / SIMULATION HPC SMART CITIES

AI / DEEP LEARNING

Real-Time Natural Language Understanding with BERT Using TensorRT

By Purnendu Mukherjee, Eddie Weill, Rohit Taneja, Davide Onofrio, Young-Jun Ko and Siddharth Sharma | August 13, 2019

Tags: Inference, Machine Learning and AI, Natural Language Processing, Natural Language Understanding, speech recognition, TensorRT

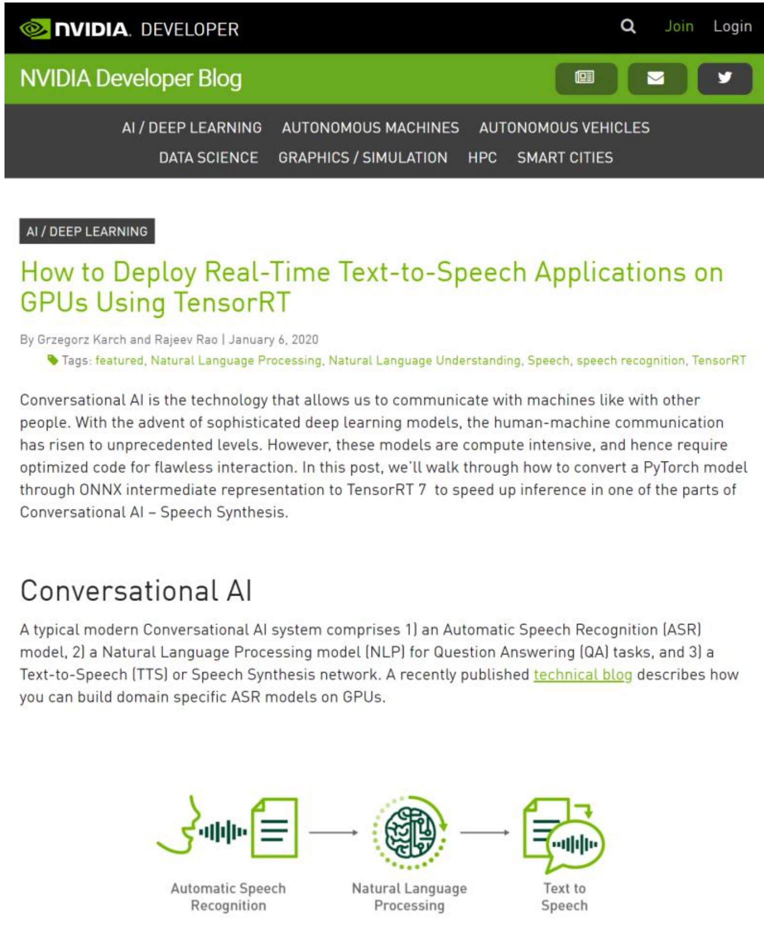
Large scale language models (LSLMs) such as BERT, GPT-2, and XL-Net have brought about exciting leaps in state-of-the-art accuracy for many natural language understanding (NLU) tasks. Since its release in Oct 2018, BERT[1] (Bidirectional Encoder Representations from Transformers) remains one of the most popular language models and still delivers state of the art accuracy at the time of writing[2].

BERT provided a leap in accuracy for NLU tasks that brought high-quality language-based services within the reach of companies across many industries. To use the model in production, you need to consider factors such as latency, in addition to accuracy, which influences end user satisfaction with a service. BERT requires significant compute during inference due to its 12/24-layer stacked multi-head attention network. This has posed a challenge for companies to deploy BERT as part of real-time applications until now.

Today, NVIDIA is releasing new TensorRT optimizations for BERT that allow you to perform inference in 2.2 ms* on T4 GPUs. This is 17x faster than CPU-only platforms and is well within the 10ms latency budget necessary for conversational AI applications. These optimizations make it practical to use BERT in production, for example, as part of a conversation AI service.

TensorRT is a platform for high-performance deep learning inference which includes an optimizer and runtime that minimizes latency and maximizes throughput in production. With TensorRT, you can optimize models trained in all major frameworks, calibrate for lower precision with high accuracy, and finally deploy in production.

All optimizations and code for achieving this performance with BERT are being released as open source in this TensorRT sample repo. We have optimized the Transformer layer, which is a fundamental building block of the BERT encoder so you can adapt these optimizations to any BERT-based NLP task. BERT is applied to an expanding set of speech and NLP applications beyond conversational AI, all of



NVIDIA DEVELOPER Join Login

NVIDIA Developer Blog

AI / DEEP LEARNING AUTONOMOUS MACHINES AUTONOMOUS VEHICLES DATA SCIENCE GRAPHICS / SIMULATION HPC SMART CITIES

AI / DEEP LEARNING

How to Deploy Real-Time Text-to-Speech Applications on GPUs Using TensorRT

By Grzegorz Karch and Rajeev Rao | January 6, 2020

Tags: featured, Natural Language Processing, Natural Language Understanding, Speech, speech recognition, TensorRT

Conversational AI is the technology that allows us to communicate with machines like with other people. With the advent of sophisticated deep learning models, the human-machine communication has risen to unprecedented levels. However, these models are compute intensive, and hence require optimized code for flawless interaction. In this post, we'll walk through how to convert a PyTorch model through ONNX intermediate representation to TensorRT 7 to speed up inference in one of the parts of Conversational AI – Speech Synthesis.

Conversational AI

A typical modern Conversational AI system comprises 1) an Automatic Speech Recognition (ASR) model, 2) a Natural Language Processing model (NLP) for Question Answering (QA) tasks, and 3) a Text-to-Speech (TTS) or Speech Synthesis network. A recently published technical blog describes how you can build domain specific ASR models on GPUs.

Automatic Speech Recognition
Natural Language Processing
Text to Speech

REFERENCES

PyTorch form Research to Production

[Shen et al 2018] "Natural TTS Synthesis by Conditioning WaveNet on Mel Spectrogram Predictions" Jonathan Shen, Ruoming Pang, Ron J. Weiss, Mike

[Prenger et al 2018] "WaveGlow: A Flow-based Generative Network for Speech Synthesis" Ryan Prenger, Rafael Valle, Bryan Catanzaro

[Li et al 2019] "Jasper: An End-to-End Convolutional Neural Acoustic Model" Jason Li1, Vitaly Lavrukhin, Boris Ginsburg, Ryan Leary, Oleksii Kuchaiev, Jonathan M. Cohen, Huyen Nguyen, Ravi Teja Gadde

[Devlin et al 2018] "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova

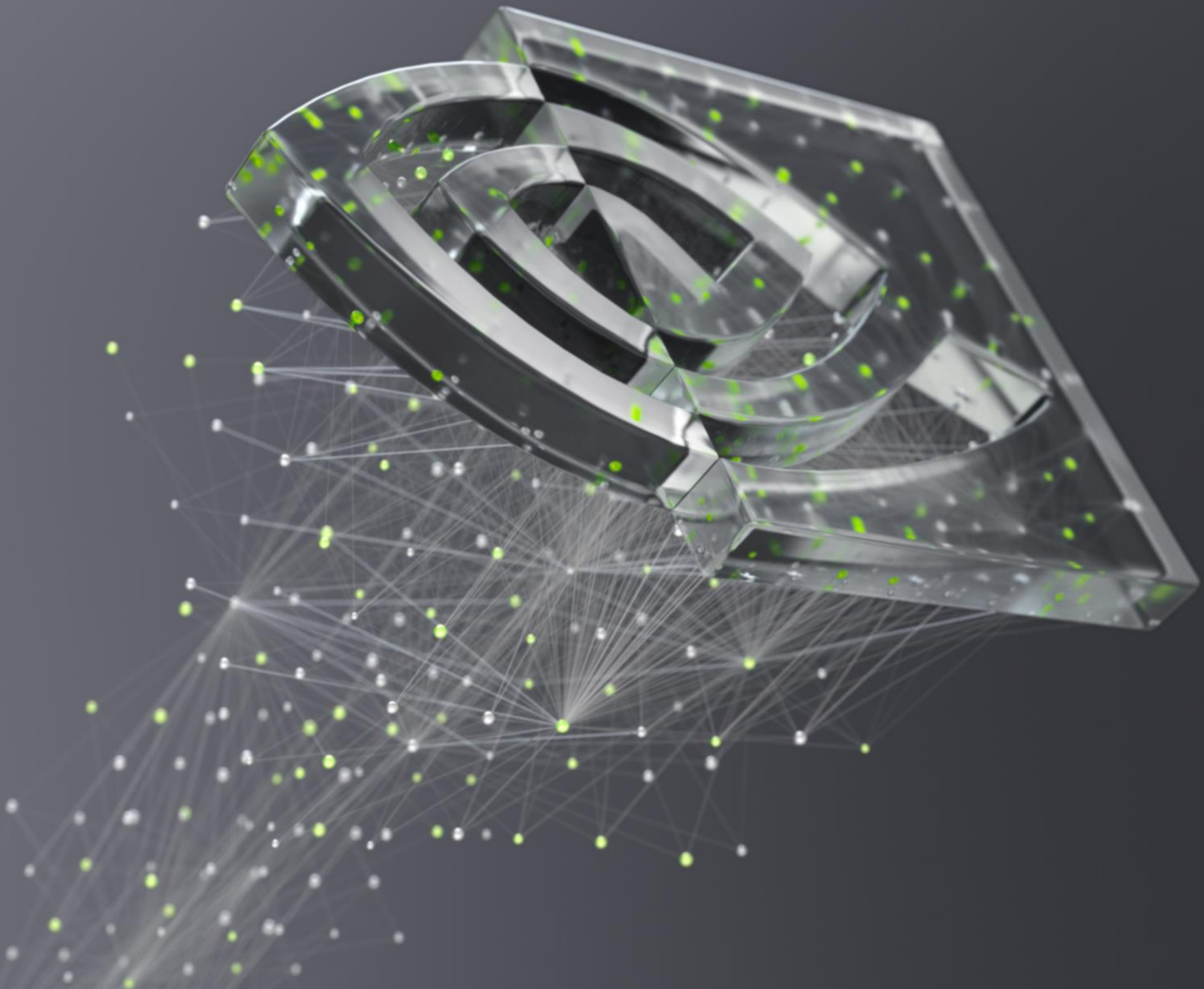
TensorRT Inference Server https://docs.nvidia.com/deeplearning/sdk/tensorrt-inference-server-guide/docs/

TensorRT https://docs.nvidia.com/deeplearning/sdk/tensorrt-developer-guide/index.html

ONNX https://onnx.ai/

PyTorch https://pytorch.org/

TorchScript https://pytorch.org/docs/stable/jit.html



NVIDIA