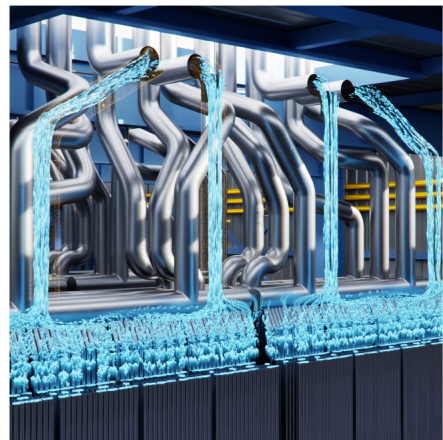


# How to Put AI Models Into Production

A Guide to Accelerated Inference

Michael Wharton and Zhangzhang (ZZ) Si with contributions from Priya Joseph



MANNING

+



# Free Getting Started Kit For AI Inference



Check out this free getting started kit and explore more AI inference resources.

### NVIDIA Triton Inference Server

NVIDIA Triton™ Inference Server, is an open-source inference serving software that helps standardize model deployment and execution and delivers fast and scalable AI in production.

[Get Started](#)

#### What is NVIDIA Triton?

Triton Inference Server, part of the NVIDIA AI platform, accelerates and standardizes AI inference by enabling teams to deploy, run, and scale trained AI models using TensorFlow or PyTorch on CPU-based infrastructure. It provides AI researchers and data scientists the flexibility to choose the right framework for their projects without impacting production deployment. It also helps developers deliver high-performance inference across clouds in a single, unified, and standardized format.

#### Explore the benefits.

- Support for multiple frameworks:** Triton supports all major training and inference frameworks such as TensorFlow, PyTorch, MXNet, FPGAs, ONNX, OpenVINO, and more.
- High-performance inference:** Triton supports all NVIDIA GPUs, including TensorRT, and uses a variety of hardware acceleration techniques to optimize inference performance.
- Designed for DevOps and ML Ops:** Triton supports CI/CD, Kubernetes for orchestration and scaling, and integrates with CI/CD pipelines.
- An integral part of NVIDIA AI:** The NVIDIA AI platform, which includes Triton, gives developers the complete set of tools and algorithms they need to accelerate their AI applications.

### Get Started with Inference

Kick-start your AI inference journey with pre-installed, pre-validated access to NVIDIA Triton and NVIDIA TensorRT—for free.

#### Experience Accelerated Inference

In these hands-on labs, you'll experience fast and scalable AI using NVIDIA Triton™ Inference Server, platform-agnostic inference serving software, and NVIDIA TensorRT™, an SDK for high-performance deep learning inference that includes an inference optimizer and runtime. You'll be able to immediately unlock the benefits of NVIDIA's accelerated computing infrastructure and scale your AI workloads. Choose our labs to get started and ask NVIDIA experts questions about your specific infrastructure.

#### Choose your desired lab to get started.

Deploy an End-to-End AI Pipeline Using a Hello World	Train and Deploy an AI Support Chatbot	Train AI Image Classification Models (Custom Model)	Deploy Fraud Detection Models with NVIDIA Triton
Best for: AI practitioners	Best for: AI practitioners	Best for: AI practitioners	Best for: AI practitioners
<b>What's Included:</b> Products: NVIDIA AI Inference Server, NVIDIA Triton Inference Server, NVIDIA TensorRT, NVIDIA Triton Inference Server	<b>What's Included:</b> Products: NVIDIA AI Inference Server, NVIDIA Triton Inference Server, NVIDIA TensorRT, NVIDIA Triton Inference Server	<b>What's Included:</b> Products: NVIDIA AI Inference Server, NVIDIA Triton Inference Server, NVIDIA TensorRT, NVIDIA Triton Inference Server	<b>What's Included:</b> Products: NVIDIA AI Inference Server, NVIDIA Triton Inference Server, NVIDIA TensorRT, NVIDIA Triton Inference Server

### Optimizing and Serving Models with NVIDIA TensorRT and NVIDIA Triton

By Tonya Verwey, Joe Ridge and Nick Conly

[Download from NGC](#) [Download from GitHub](#)

### Getting Started with NVIDIA Triton

Deploy, run, and scale AI models in production in the cloud, on premises, or at the edge with NVIDIA Triton™.

[Download from NGC](#) [Download from GitHub](#)

### Fast and Scalable AI Model Deployment with NVIDIA Triton Inference Server

By Shankar Chandrasekaran and Mohan Saheli

[Download from NGC](#) [Download from GitHub](#)

### NVIDIA TensorRT

NVIDIA™ TensorRT™, an SDK for high-performance deep learning inference, includes a deep-learning inference optimizer and runtime that delivers low latency and high throughput for inference applications.

[Download Now](#) [Get Started](#)

### Speeding Up Deep Learning Inference Using NVIDIA TensorRT (Updated)

By Jack Park, Suresh Chinn, Siddharth Sharma and Reshmi Abhishek

[Download from NGC](#) [Download from GitHub](#)

This post was updated July 20, 2023 to reflect NVIDIA TensorRT 8 updates. NVIDIA TensorRT is an SDK for deep learning inference. TensorRT provides APIs and parsers to import trained models from all major deep learning frameworks. It then generates optimized runtime engines deployable in the

### What Will You Learn?

Advancing your AI project to production can be challenging. Learn how to standardize and optimize your AI inference across multiple model frameworks, different queue types, and diverse CPU and GPU infrastructures with NVIDIA Triton™ Inference Server.

#### Challenges to Deploying AI Models in Production

- Multiple Frameworks:** The need to manage diverse training frameworks, such as TensorFlow, PyTorch, MXNet, and ONNX, across different hardware architectures.
- Model Infrastructure:** Learn the challenges of managing diverse model frameworks across different hardware architectures.
- Scaling Deployment:** The need to manage both on-premise and cloud-based inference workloads across different hardware architectures.
- Diverse Inference Types:** The need to manage both batch and streaming inference workloads across different hardware architectures.

#### Register to Access the Whitepaper

First Name:  Last Name:   
 Business Email Address:   
 Organization/Company Name:   
 Address:  City:   
 Country:  State:   
 Zip:  Phone:   
 I agree to receive emails from NVIDIA:

### TensorRT: WHAT'S NEW

NVIDIA™ TensorRT™ 8.5 includes support for new NVIDIA H100 GPUs and reduced memory consumption for TensorRT optimizer and runtime with CUDA Lazy Loading.

TensorRT 8.5 GA will be available in Q4 2022.

[Download Now](#)

TensorRT 8.4 Highlights:

- New tool to visualize optimized graphs and debug model performance easily. [Learn more](#)
- Reduce engine size by up to 50%, allowing for easier application package distribution.
- New TF-GAT toolkit for improved INT8 accuracy on TensorFlow models. [Learn more](#)

TensorRT is available today in the PyTorch container from the NVIDIA NGC™ catalog.

TensorFlow™ TensorRT is available today in the TensorFlow container from the NGC catalog.

EXPLORE MORE RESOURCES



*How to Put AI Models Into Production*  
*A Guide to Accelerated Inference*


Michael Wharton and Zhangzhang (ZZ) Si  
with contributions from Priya Joseph

©2022 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

- ⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

 Manning Publications Co.  
20 Baldwin Road Technical  
PO Box 761  
Shelter Island, NY 11964

Cover designer: Ben Counsell

ISBN: 9781633438613

# *contents*

---

*foreword iv*

- 1 A brief overview of AI inference 1**
  - 1.1 Key artificial intelligence terminology 3
  - 1.2 Modern inference microservices: How we got here 4
  - 1.3 Inference in classical machine learning 7
  - 1.4 Inference in deep learning 7
  - 1.5 Inference patterns 8
  - 1.6 A modern inference architecture 9
  - 1.7 Challenges and best practices for inference systems 12
  
- 2 AI inference case studies 15**
  - 2.1 Optimizing and scaling solutions with NVIDIA TensorRT and Triton 16
  - 2.2 Inference in natural language processing (NLP) 18
  - 2.3 Inference in computer vision 21
  - 2.4 Inference in recommender systems 22
  - 2.5 Inference in fraud detection 23

# *contents*

---

- 3 **AI inference in practice** 25
  - 3.1 Challenges of inference deployment 25
  - 3.2 Optimize models with TensorRT 28
  - 3.3 When to use TensorRT 29
  - 3.4 Deploy inference with Triton Inference Server 30
  - 3.5 Recipes for different data types 32
  - 3.6 Recipes for complex inference tasks 40
  - 3.7 Deployment process and best practices 44
  - 3.8 Code lab: Deploy inference for reverse image search 51
  
- 4 **The AI inference horizon** 53
  - 4.1 Broad AI adoption 53
  - 4.2 Algorithms 54
  - 4.3 Regulatory environments 55
  - 4.4 Additional trends 56
  - 4.5 Summary 56

# foreword

---

There are two parts to AI machine learning: *training* a model and *using* the model. Training AI machine learning and its subset, neural network-based deep-learning models, has well-established frameworks and processes. It is a focused area in organizations with dedicated data scientists and ML engineers. They are part of a large community of developers that has standard frameworks, tools, and processes to help.

But deploying and using the AI model in production is ad hoc in many organizations. Each team has its own ways. Some are on the public cloud, others run models on-premise or at the edge. The computing infrastructure varies, too. Some teams might run their models on standard CPU-based servers while others demand accelerators like GPUs. Traditional tools and processes used for deploying enterprise applications are not sufficient. The IT, DevOps practitioners, or software developers do not have in-depth knowledge of what it takes to put AI models in production environments.

Successful use and growth of AI in an organization needs understanding of all the facets of AI inference. AI inference uses AI models to make predictions in the production environment. In other words, it deploys the trained model and makes it operational in a product or service. It needs a focused approach with specific hardware and software considerations like that of a web or a database application. Teams need a standardized way to deploy, run, and scale AI models.

At the recent NVIDIA [GPU Technology Conference \(GTC\)](#), several companies were highlighted to show how they are addressing challenges in inference. For example, NIO, the electric car maker, is building a scalable inference system to deploy hundreds of models to process huge amounts of data from autonomous vehicles. Airtel, the second-largest wireless provider in India, needs a high throughput inference solution to process data from hundreds of thousands of customer support calls every day. Companies like GE Healthcare, Wealthsimple, and Yahoo Japan are looking to streamline and centralize inference deployment across frameworks, computing processors, and devices. Those organizations that are embracing AI need to master both training and inference for a sustained competitive advantage.

AI practitioners, data scientists, ML engineers, IT/DevOps, and others need to look at both today's and tomorrow's requirements for deploying and running models in production applications. They must account for constantly evolving model frameworks with different teams using different frameworks. The computing processors, accelerators, and software platforms are also evolving at an unprecedented pace. The number and type of models are expanding. AI is making its way into many areas of the business. This is a book from AI practitioners to other practitioners, written with the goal of filling that need to share that practical and foundational knowledge.

Michael Wharton, Dr. Zhangzhang Si, and Priya Joseph share practical perspectives obtained from years of experience working at places such as Applied Research Laboratories, Expedia, Amazon Web Services, and more. We also share the feedback that NVIDIA has received from its customers—all in a single place to learn about the different dimensions of AI inference.

This book offers a look at current approaches to current situations, but it also provides the fundamentals you need to address your own situations. The space of AI is evolving every day, creating new needs and new solutions. We can't wait to see what you come up with as you apply these general principles and explore the inference stage of AI. Enjoy the book!

—Shankar Chandrasekaran  
Senior Product Marketing Manager,  
NVIDIA



# A brief overview of AI inference

---

Imagine you are on a team of engineers that recently expended countless resources cleaning data, refining pre-processing pipelines, conducting experiments, and training candidate models for a language translation application. Prior to this work, market research revealed that a competitor had achieved translation accuracies that far surpass those of your product’s legacy approach. Since this realization became clear, the resulting blood, sweat, and tears poured into catching up have finally paid off. Your team reached the project’s goal metrics by using a transformer-based model architecture fine-tuned on your company’s proprietary data set. However, now that you have a proven artificial intelligence (AI) model artifact and a well-established preprocessing pipeline, how can you use it to finally regain a competitive footing for your enterprise?

In order to use your model, it’s imperative to deploy an *AI inference system* into your production application. *AI inference* refers to making queries to an AI model using novel inputs and returning the resulting predictions. In AI inference systems, predictive models—often machine learning models—are packaged in such a way that another service can ask for a prediction given some input data. In chapter 2 we’ll show you how this works with the hypothetical translation app we just described—by deconstructing the very real Microsoft Translator.

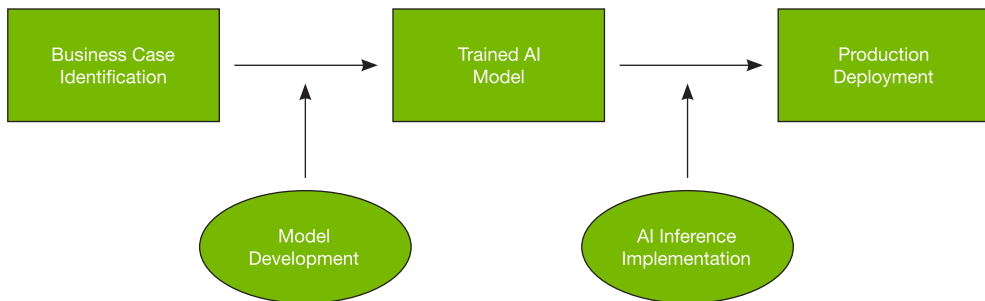
Let’s say you’ve built a system for an e-commerce site that predicts the time it takes to ship items to customers. Given an origin location and a destination location,

context about an item to be shipped (like size and weight), as well as additional meta-data about the journey (whether it takes place over a holiday, for example) the AI model provides an estimate of the overall transit time. The model takes all these as inputs and performs the prediction, also known as inference. The service that performs these inferences is an AI inference system.

In other words, AI inference allows us to transform machine learning models into something of value at scale. It allows us to gain insight from these powerful tools that we've spent untold time, energy, and expense creating.

AI models could, of course, be embedded directly into an application, and that is indeed how it has been done in the past. But given the size and complexity of modern AI models, as well as their compute demands, they lend themselves well to a microservices approach. A simple linear regression model, for instance, could be hard-coded directly into an application; but when model parameters number in the millions (as with AI models), a compartmentalized prediction service is all but inevitable.

Figure 1.1 shows where AI inference implementations fit into a typical development progression.



**Figure 1.1** AI inference systems help simplify the path to production deployment.

Many businesses struggle to cross the bridge between development and production when using machine learning, and this struggle is often due to the multitude of challenging barriers that inference deployments present. Artificial intelligence can be used for a vast array of tasks including autocompletion, forecasting, fraud detection, document scanning, search, and many more. Each of these use cases has a particular set of constraints that bring with them their own unique challenges, though all are readily achievable given the right tools and insights. One problem common to all products and services developed with AI models is the deployment phase.

The task of solving this “last-mile problem”—the problem of operationalizing your prediction artifacts—is the primary focus of this text. We articulate and address many of these difficulties in the hope of making inference systems more easily adopted. This

report should function as a primer to inform more thoughtful engineering and business decisions on the topic. We won't teach you how to build your own inference system, but we will prepare you to build it, or to manage a team that is building it. After reading this, you should be able to speak intelligently and make sound decisions about various types of inference, common inference tooling, as well as impacts in a range of industries.

We start this work with a thorough introduction to AI inference, including key terminology and a look at the historical context that has led to the current approaches. In chapter 2, we present an array of use-case examples from various industries to give you an idea of the wide-ranging applications of inference systems. Both the challenges and the solutions to those challenges are instructive. Then, in chapter 3, we will get into the engineering details of inference systems and see how they are built. Finally, in chapter 4, we will discuss industry trends and describe some market signals that indicate where the AI industry may be headed.

Through practical explanations and a series of case studies, we aim to acquaint you with all the foundational knowledge you'll need to see how and why an inference system may make sense for your projects. We hope that this concise guide becomes a reference to come back to over time, as you become more familiar with inference systems.

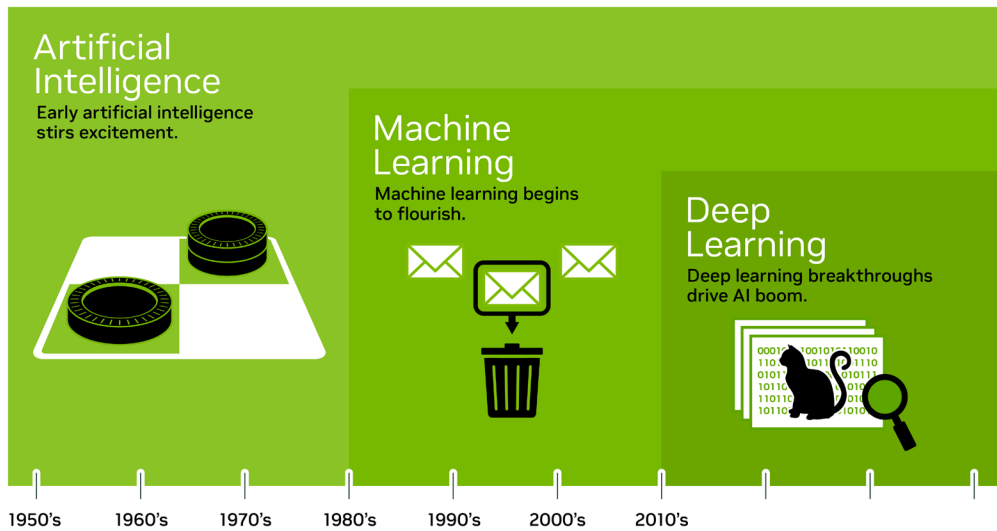
Let's start with a quick look at the historical context for contemporary approaches to AI inference.

## 1.1 Key artificial intelligence terminology

In order to understand practical inference in classical machine learning, developers, data scientists, and business leaders need to be on the same page about the terminology. Though common and familiar, the terms around AI are often conflated or misused. Though somewhat subtle, the distinctions between them help provide language we can use to discuss all the various applications leveraging such powerful and significant technology.

Let's take a look at the key terms (figure 1.2):

- *Artificial intelligence*: Systems designed to perform tasks otherwise performed by humans.
- *Machine learning*: The practice of training systems on data (as opposed to programming them with rule sets) to make inferences.
- *Deep learning*: A subset of machine learning where the trained systems are multi-layered (i.e., "deep"), contain progressively more complex learned representations of the input at higher depths, and are highly parameterized.



**Figure 1.2** The relationship between artificial intelligence, machine learning, and deep learning.

Given these definitions, it is true to say that deep learning is a subfield of machine learning, and that machine learning is a form of artificial intelligence. “Classical machine learning” is often referred to as the subset of machine learning techniques that preceded deep learning. These techniques are frequently much less complex, much less parameterized, and therefore require much less computation to make novel predictions.

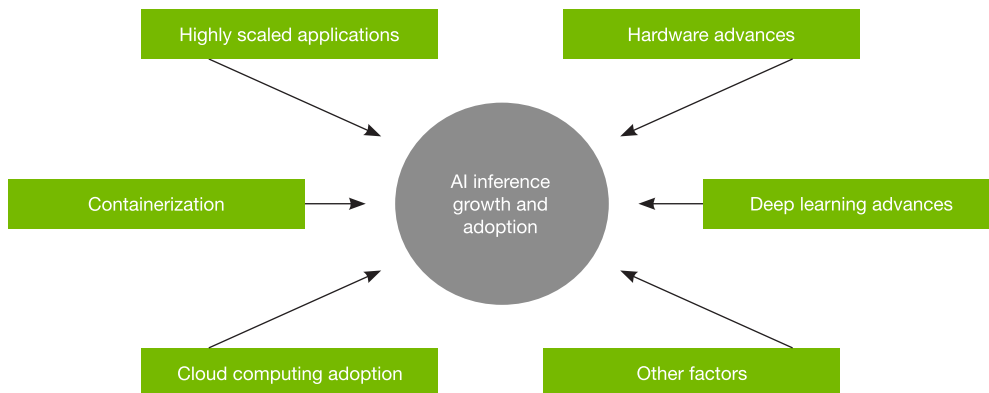
## 1.2 *Modern inference microservices: How we got here*

AI is becoming such a crucial part of the world we live in that a baseline understanding of AI is a necessary prerequisite for navigating modern business contexts. It has become a critical component to the banking, retail, real estate, healthcare, manufacturing, and advertising industries, among many others. AI algorithms can detect fraud, predict ad performance, uncover nascent cases of cancer, and discover and suggest products that you didn’t know you needed. It’s certain that AI is here to stay.

Inference deployment does the work of operationalizing AI models in every one of these settings, without exception. Without inference, it would be impossible to put these systems to good use. Furthermore, unless carefully optimized for their unique prediction tasks, the underlying models often impose a computational burden that’s cost-prohibitive. Though tricky at times, if the right tools, pipelines, and processes are made available to all stakeholders involved (that is, customers, development teams, product owners, and business leaders) inference solutions can provide strong

competitive business advantages. It is our expectation that this text serves as a reference to enable just that.

Many historical developments have set the stage for the modern approach to inference. Advances in hardware and software, as well as norms adopted by the industry, have shaped the best practices landscape over time. Chief among them are highly scaled applications, hardware advances, the advent of cloud computing, a focus on containerization, advances in deep learning, and ever-evolving use cases. Let's look into each of these factors (summarized in figure 1.3) in detail.



**Figure 1.3** Some factors that affect the AI inference landscape.

*Highly scaled applications:* The demand for scale and efficiency has only increased in recent decades as edge device capabilities, network bandwidth, and global connectivity have improved in kind. For example, American Express makes around 8 billion AI-powered decisions annually to mitigate fraud on more than \$1 trillion in transactions. The latency, volume, and compute requirements associated with such a scale necessitates that we rethink traditional infrastructure paradigms.

*Hardware advances:* Interest in parallel processing has been an ongoing feature of computing since the 1950s. In the last decade in particular, deep-learning models that require highly parallelized computation have been widely adopted. This has in effect acted as a forcing function for advancement. Graphical processing units (GPUs) and a variety of application-specific integrated circuits (ASICs) have enabled scaled, low-latency model training and prediction. The feedback loop between highly demanding prediction tasks (like AI inference serving) and the parallel compute they require continues to this day.

*The advent of cloud computing:* In the traditional on-premise (“on-prem”) server paradigm, compute resources were effectively fixed for the owner’s business use cases. In many cases, traditional infrastructure design would have required designing to the

maximum demand a system might encounter, which can be cost-prohibitive. However, modern cloud computing has enabled not only incredible scale with a relatively low complexity cost, but also flexibility in resource usage. In the most extreme case, serverless computing enables dynamic allocation of resources to meet demand “on-the-fly”. In general, scaled inference has become more cost-effective and accessible to a wider developer audience, which sets the stage for a thriving industry surrounding AI inference in the cloud.

*A focus on containerization:* Containers are portable versions of applications that for the most part are unconcerned with the environment that they are run on. Developing them not only enables flexibility in the choice of runtime hardware, but also reproducibility and standardization. Furthermore, it’s much easier to establish standardized application interfaces, which can in turn make deploying inference a much easier task. Application containers are in essence a standard layer of abstraction that sit on top of the environment itself, whereas historically, environment specifications and install scripting would have been managed by the developer at deployment time. Since containers have become standard in virtually all enterprise applications, it has laid a foundation for more standardization in the AI/ML world.

*Advances in deep learning:* Perhaps the most salient point is that deep learning has demonstrated tremendous business value for a wide variety of use cases in the last decade. In 2012, a groundbreaking convolutional neural network architecture named [AlexNet](#) drastically improved state-of-the-art performance on the popular ImageNet image classification benchmark, besting the previous record holder by nearly 11 percentage points on top-5 error. (Note: “Top-5 error” refers to how often the correct image classification is in the model’s top 5 ranked predictions.) Around this same period, major advances in computer vision became commonplace. Since then, neural network-based computer vision architectures have proven to be highly important in many business settings, and similar advances have been achieved in natural language processing (NLP), reinforcement learning, and other domains. Typical model architectures require millions of computations to make a single prediction, and in extreme cases, large language models (“LLMs”) like [OpenAI’s GPT-3](#) require billions. Though not all production-grade predictive models use deep learning, this context heavily influenced the modern inference landscape by making high volumes of parallel computation routine.

*Ever-evolving use cases:* With the popularization of AI and machine learning, statistical models of all shapes and sizes have shown business viability at scale. Some examples include:

- Spell checking
- Search engines
- Document translation
- Audio transcription

- Automated visual inspection
- Fraud detection
- Text-to-speech with realistic voices
- Anomaly detection
- Content personalization and recommender systems
- Forecasting
- Price prediction
- And many more ...

In essence, because so many use cases have proven business utility when adapted for use with artificial intelligence, it's imperative that business workflows adopt predictive modeling to remain competitive. A properly configured inference engine lives at the heart of such a workflow.

### 1.3 Inference in classical machine learning

Consider an example where a model is required to predict price on real estate listing data. Let's assume that many categorical features (e.g., square footage, number of bathrooms, kitchen counter finish types) exist and that the number of encoded input features is 100. For a simple linear regression model, like those used in classical machine learning, the number of parameters would be 101. One hundred multiplications and one sum operation are required to generate a single prediction.

Now consider a more modern multi-layer perceptron (MLP) network (the most basic type of deep neural network) with three hidden layers of sizes 75, 50, and 25. Such a model would have over *12 thousand parameters* and would therefore require orders of magnitude more computation to generate a single inference (i.e., to predict price on a *single* real estate listing). In many scenarios, it is preferable to pay the cost of the additional compute in exchange for a model performance boost.

The big takeaway from this simple example is that classical machine-learning algorithms (like k-nearest neighbors, decision trees, and logistic regression models) by and large can be deployed at scale using generic and standard hardware like CPUs. The process of scaling infrastructure to meet demand and latency requirements is then a fairly straightforward exercise. There are certainly cases where classical ML models benefit from GPU acceleration, but deep-learning models almost always require a more powerful compute solution. Because of the stark contrast between the computational demands of classical ML and deep-learning models, the typical software and hardware requirements for each vary substantially.

### 1.4 Inference in deep learning

Deep learning architectures span a wide variety of designs, configurations, and data modalities. However, despite this diversity, nearly all of them benefit from highly parallelized computation in latency-constrained applications. These powerful models have parameter counts that can range from thousands to hundreds of billions, depending

on the model architecture. When a use case demands a deep-learning inference (e.g., real-time video frame analysis), model optimization and specialized hardware are all but unavoidable.

Computer vision (CV) and natural language processing (NLP) models, in particular, are the most “compute-hungry” of all applications. However, because of the incredible value they provide, we are forced to engineer solutions that allow for widespread utilization of these models.

A very common and highly performant NLP model called [BERT](#), for instance, houses 110 million parameters in the base configuration. A typical computer that primarily leverages CPUs for computation may require more than one minute to generate a single inference, so one must turn to accelerated computing to improve latency and make business cases viable. For example, NVIDIA’s graphical processing unit (GPU) offerings are well-equipped for the task of generating parallel compute-intensive inferences, and solutions like the open-source [NVIDIA Triton Inference Server](#) enable scale across high-demand and dynamic systems. Computer vision (CV) and natural language processing (NLP) models in particular are very well-suited for use with these high-performance infrastructure components, because they often have the highest per-inference compute cost. We will go into more detail on related applications in subsequent chapters.

## 1.5 Inference patterns

Three different methodologies, or patterns, exist for computing inference in any appreciable volume: real-time, batch, and streaming. Recognizing each of these scenarios and defining appropriate requirements are the first steps to good infrastructure design.

*Real-time inference* refers to inference computed as data is ingested. “Real time” implies that the server must perform at a low latency, and although there is no standard latency requirement, it is typically assumed to be less than a few seconds. An example here is a recommender system that produces content recommendations based on user input along with browsing history. The inferences must be generated during the page load time and therefore are real-time. Table 1.1 outlines some typical latency requirements for real-time performance in various applications.

**Table 1.1** Typical latency requirements for real-time inference, by application.

Application	Real-time latency requirement (ascending)
Fraud detection	1.5 milliseconds
Digital ad bidding	10 milliseconds
Image search	0.15 seconds
Speech recognition	0.3 seconds
Chatbot	2 seconds



*Batch inference*, on the other hand, refers to the case where inferences are made on samples in groups, or “batches”. The latency requirement in these scenarios is often much more relaxed, typically on the order of hours or days, and therefore infrastructure with a lower availability or throughput can be employed. For instance, consider an online store which produces a weekly personalized newsletter that includes content recommendations. Even though the model used in this scenario could be exactly the same as the one in the previous example, the need for serving inferences is infrequent. Batch processing would therefore be a more appropriate pattern.

*Streaming inference* is used when data must be processed continuously. Such a paradigm becomes important in self-driving car applications, for example, where continuous data from the driving environment informs insight and control. Generating inference without interruption is paramount to the safety of the vehicle occupants. An important feature of continuous inference systems is that the rate of inference must exceed the rate of sample generation, or otherwise make inferences on down-sampled data. Without sufficient compute performance, the inference system can’t process the data stream as quickly as the data are generated. If an object tracking model, for instance, processes video at 20 frames per-second (FPS), yet the camera operates at 30 FPS, one must either improve prediction latency or predict with a lower overall throughput.

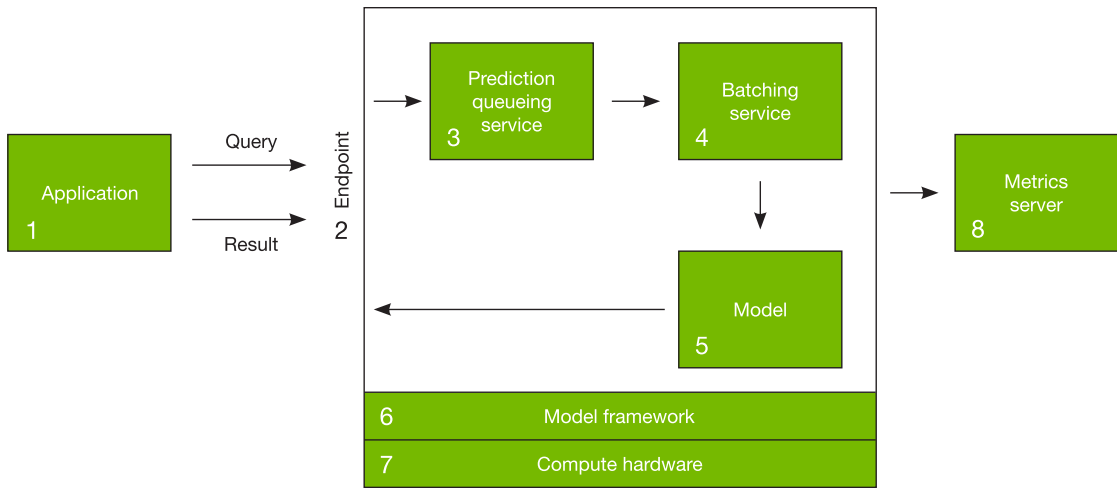
You may also encounter two additional terms in the inference context: *online* and *offline*. In an online inference setting, throughput within a specified latency budget is paramount. Conversely, offline predictions are not as latency-constrained and therefore afford more flexibility in the deployment infrastructure and configuration. The online and offline patterns are analogous to the real-time and batch patterns, respectively, so you may see these terms used somewhat interchangeably.

Each pattern brings with it a host of hardware and software implications, where each design path may differ wildly. It is therefore important to understand all three and intuit when one may be more relevant than the others. Practical examples for each will be detailed in subsequent sections.

## 1.6 A modern inference architecture

Unfortunately, there is no one single system architecture to tackle all possible inference use cases. Because the pace of iteration is so high when it comes to modern deep learning, hardware, and tooling solutions, what works especially well today may well need some revision in the near future. That said, some approaches afford more flexibility and broad applicability than others, in addition to meeting the unique demands of the task at hand. In this section we present you with one such architecture.

Regardless of the inference pattern applicable to your use case (real-time, batch, or streaming), the generic architecture depicted in figure 1.4 is a more-or-less universal solution that should cover most use cases.



**Figure 1.4** A basic system architecture for an inference server.

Let’s walk through this diagram, looking at each component that contributes to the inference system’s performance and reliability. (Note: The number next to each component in this list corresponds to the numbered component in the diagram.)

### 1.6.1 *Application (1)*

This is the place where the inferences are ultimately requested and consumed. Whether making content recommendations, inventory forecasts, or something else entirely, minimizing the complexity and overhead of the inference system is crucial to maximizing application development time.

### 1.6.2 *Endpoint (2)*

Inference servers generally use either standard HTTP or gRPC protocols. In networked architectures, the application will make requests to the inference service through some sort of API, where responses containing prediction payloads follow a contract defined by the development team. These predictions are then used downstream to inform decisions and generate further insight.

### 1.6.3 *Prediction queueing service (3)*

Queueing enables an inference system to accommodate groups of requests that exceed the system throughput capacity, among other benefits. If a group of requests all come in at once, for instance, inferences can be made as the compute hardware allows, while staging incomplete requests until resources become available. Whether ephemeral or persistent, model availability at prediction time may not always be sufficient to meet bursts of demand. The queueing service makes continued system reliability possible.

### 1.6.4 **Batching service (4)**

Though ultimately optional, having a service within the inference system that performs batching is critical for optimal compute utilization and latency. Performing effective batching is made complicated by the constraints of the compute hardware, but solutions like NVIDIA's Triton Inference Server can operate effectively in the background without much configuration. Having some flavor of a batching service can mean the difference between meeting requirements and not meeting them. In a hypothetical case where an e-commerce website makes product recommendations for thousands of customers each second, a batching service is crucial to meet demand. This approach also allows for more efficient utilization of accelerated compute hardware.

### 1.6.5 **Model (5)**

After a batch of requests has been created, the inference itself must be generated. Whether the “model” in this case is a machine-learning model, a logistic-regression model, or even a simple set of heuristics is unimportant. The only requirement of this “model” is that input data are transformed into predictions. Note that pre-processing (e.g., image normalizing) and post-processing (e.g., converting class numbers to text labels) pipelines are often present as well, but in this diagram, they are assumed to be part of the model itself. Said succinctly, the model is the core piece of intellectual property that drives the inference server.

### 1.6.6 **Model framework (6)**

The model framework does the work of translating parameterized model definitions into instructions on your hardware of choice. [TensorRT](#), [PyTorch](#), and [TensorFlow](#) are all commonly used frameworks that fit within virtually any inference server context and are often chosen based on developer familiarity. It's important to note that some frameworks are used for development, deployment, or some combination of the two. PyTorch, for instance, is commonly used to both train models and deploy them, whereas a format like TensorRT is primarily intended for deploying already-trained models for inference.

Performance benchmarking is crucial for critical applications, as the relative performance may vary substantially between frameworks and architectures. The backend choice plays a key role in the success of any system, especially since multiple models may be relevant to a given application.

### 1.6.7 **Compute hardware (7)**

The compute hardware does the raw computation to generate inferences. As discussed previously, many different options exist for this component, but the majority of deployments use some combination of CPUs and GPUs. All common backend frameworks (e.g., TensorRT, [ONNX](#), PyTorch, or TensorFlow) support flexible computation on either. If you're using NVIDIA's [Triton Inference Server](#) for your inference deployment, the Triton Model Analyzer enables you to test your models on a variety of

compute solutions in order to optimize performance and cost when making this key decision.

### **1.6.8 Metrics server (8)**

Last, deploying a metrics server is crucial for server health management and performance administration. Common metrics may include statistics on inference latency, hardware utilization, request volume, prediction accuracy (when calculable), and many others.

Though other components may exist (e.g., a load balancer), the previously discussed core pieces are germane to the success of any production inference server. It's also worth noting that additional tooling and harnessing around the inference workflow lives outside the serving context, especially in relation to the model training pipeline (e.g., a model registry). Thus, the aforementioned architecture, though general and performant, should be considered a bare minimum for the successful development, deployment, and management of statistical prediction pipelines.

## **1.7 Challenges and best practices for inference systems**

The complexities and design levers for inference infrastructure are nearly as diverse as the use cases they enable. Many common themes emerge, however, and in this section, we'll look at some of the top challenges and approaches to addressing them. Understanding this list of considerations (by no means exhaustive) is crucial to making sound design decisions for inference applications.

The best practices presented here are meant to serve as a concise checklist of the major guidelines, rather than all the details one might need to implement each of them. Inference and MLOps tooling is typically well-documented, so there should be a multitude of resources covering the tool stack you select. The concepts described here should be general enough, however, that a valid solution will exist in each case.

### **1.7.1 Model management and orchestration**

Model management and orchestration refers to the processes and tools utilized to manage predictive model artifacts. There are many tools on the market that exist to meet this need (MLFlow, Weights & Biases, and Neptune.ai, to name a few), but all serve a similar purpose at their core. Models are core assets in machine learning applications, and like software development, model development is a highly iterative process that requires some level of governance to ease the transition to deployment as well as enable reproducible results.

Some best practices include:

- Version your models and associate unique version numbers with your deployments.
- Store in a format that is compatible with your deployment infrastructure.
  - Examples: TensorFlow, TFLite, PyTorch, ONNX, TensorRT

- Package ancillary artifacts with each model to make the model generation process reproducible.
- Ensure your model orchestration platform is highly available to your deployment environment and team.

### 1.7.2 Model optimization

Many techniques exist for optimizing model performance in production. As a general rule, higher compute, memory, and cost efficiencies come at the expense of model performance metrics (e.g., accuracy, F1 score, RMSE). This may not always be true, but it should be a reliable heuristic in the vast majority of cases. Techniques such as network pruning, quantization, and mixed precision inference exist to help minimize unnecessary computation, but there is always a balance to strike when using these techniques. Optimization is a very active research topic, especially as it relates to the model training process, and many advances on this front should be expected in the coming years. Though it's possible to skip the model optimization step altogether, the resulting inefficiencies may impact latency, user experience, and recurring costs of the eventual production system.

Some best practices include:

- Identify and exceed a realistic target for your model performance metric. Any margin above this metric leaves room for optimization.
- Consider both memory and latency requirements when choosing an optimization strategy.
- Evaluate optimized models in the same way the parent models are evaluated for performance.
- If necessary, explore multiple model formats, as compute efficiency may differ from framework to framework.

### 1.7.3 Model framework selection

Each model framework has its own advantages and disadvantages. Some are optimized for particular hardware, while others are adapted for a more intuitive experimentation interface. It is frequently the case that the skillsets of the development team and pre-existing codebases make the primary framework choice a foregone conclusion, but it's important to note that the development and deployment frameworks are not necessarily required to be the same.

TensorRT, Tensorflow, PyTorch, Keras, and Scikit-Learn are all commonly used on modern deployments. When choosing a framework, here are some best practices to keep in mind:

- Consider your development and deployment frameworks independently (though they may be the same).
- Ensure each model framework is compatible with all aspects of your development and deployment environments, respectively.

- Consider the skillsets of the teams that build and deploy your models, because this will impact engineering velocity.

### **1.7.4 Compute hardware selection**

Compute hardware selection can be intimidating, because the risk of suboptimal choices can result in underutilized resources and/or increased cost. In some cases (e.g., edge deployments), the hardware may already have been selected before the deployment, but in most cases, a wide variety of solutions may exist. Though serverless solutions can help abstract away the hardware selection piece, the hardware is most often a critical consideration. Here are some best practices to consider:

- Clearly define all requirements (latency, memory, throughput, I/O, and others) prior to making a critical selection for production.
- If using deep learning in a latency-constrained application, you almost certainly will need a GPU or other non-CPU ASIC designed for highly parallelized computation.
- Perform empirical tests on a multitude of hardware stacks to validate your utilization hypotheses (e.g., with NVIDIA's Triton Model Analyzer).
- Frequently monitor your inference system to detect changes that require hardware reconfiguration, as could be the case with a drop in hourly request volume.
- Consider scale. In the most dynamic and scaled applications, a serverless container orchestration solution (like Kubernetes) may be important to consider.

### **1.7.5 Model evaluation**

Like writing unit tests to validate software, developing an evaluation pipeline is crucial to ensure adequate model performance prior to deployment. Common components include a “hold-out” set of test data (i.e., data not used to train a model), performance metrics, and subscale testing strategies like canary deployments. Some best practices include:

- Never use evaluation data to train models (this can result in an artificially inflated performance score that fails in production, otherwise known as “overfitting”).
- Consider all means possible to validate model performance and mitigate deployment risk, up to and including human review.
- Minimize requests to use your evaluation pipeline, as repeated use may result in reduced reliability due to a problem called “information leakage”.
- Ensure evaluation hardware is the same (or nearly the same) as your deployment hardware, as performance differences could add risk to meeting production requirements.

Careful consideration of these recommendations can help ensure a more stable deployment, which reduces the risk of problems encountered in production. If these points are not considered prior to deployment, issues may arise during production operations that could cause otherwise avoidable system downtime or poor user experience.

# *AI inference case studies*

---

In this section you will find a library of brief case studies that demonstrate what a successful inference deployment could look like across a variety of industry verticals: natural language processing (NLP), speech AI, computer vision (CV), recommender systems, and fraud detection. Though the business uses of machine learning algorithms vary substantially, the applications we describe here are some of the most well-established in production scenarios. There are certainly active areas of machine-learning research that have not yet made the jump to “production-ready.” However, the case studies outlined here have shown demonstrable business value and deployability. Many as-yet-unseen use cases will likely become commonplace in the near future, but currently the case study selections outlined in this chapter dominate the field.

These examples highlight a range of industries as well as a variety of machine-learning algorithms in order to show just how pervasive AI inference applications have become. Though not exhaustive, this section illustrates a host of possibilities in practical artificial intelligence, and by *practical* we mean artificial intelligence with both technical feasibility and business viability. Though brief, these stories include links to further reading that will provide additional detail for the curious reader. We hope these stories will inspire and educate, in addition to helping you avoid some common missteps and hurdles you may encounter as you design, build, and/or manage your own inference systems.

Prior to detailing these case studies, however, it’s worth highlighting a couple of key technologies that underpin many of them. These tools, TensorRT and the Triton Inference Server, both developed by NVIDIA, will get practical walkthroughs and

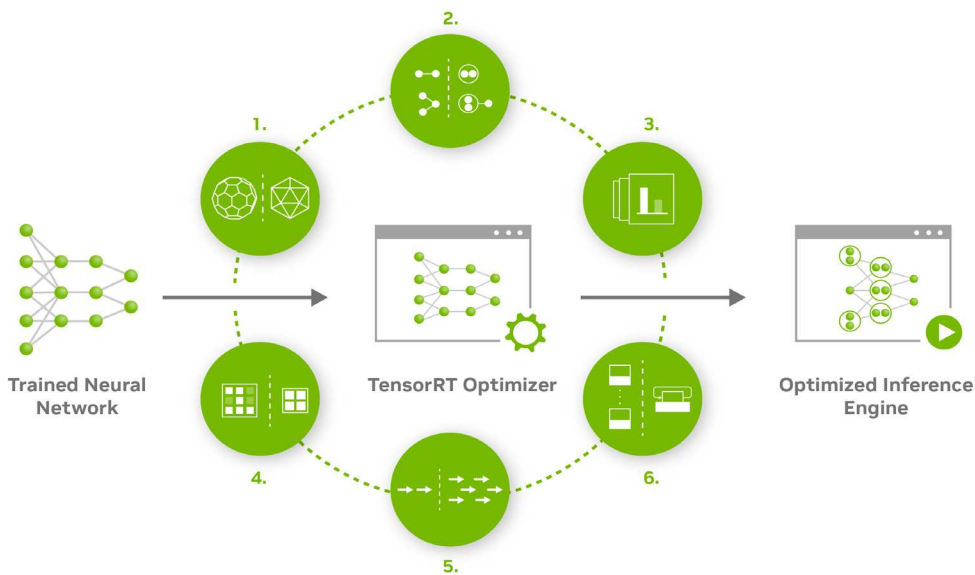
more detail in chapter 3. But understanding them at a high level will help you get the most out of these case studies.

## 2.1 **Optimizing and scaling solutions with NVIDIA TensorRT and Triton**

Setting up inference systems for deep learning that scale well is both easier and faster now that we have repeatable tools that are very reliable. Two relatively recent developments have become especially important: SDKs to streamline the process of optimizing models and serving frameworks that streamline deployment. Options among SDKs include Tensor RT, Deep Speed, and the TensorFlow Model Optimization toolkit. And serving frameworks include Triton, Seldon Core, TorchServe, Bento ML, TensorFlow Serving, and KServe. We won't go into the differences between all these options here; instead, we will focus on the two tools that our case studies rely on: Tensor RT and the Triton serving framework.

TensorRT is an SDK for high-performance deep learning that includes an inference optimizer and runtime that delivers low latency and high throughput. Triton, meanwhile, is an open-source inference serving framework that standardizes model deployment and execution and delivers fast and scalable AI in production. When used together, TensorRT and Triton allow for the optimization and scaled deployment of development models in production contexts. Let's get a quick idea of how each works.

TensorRT uses a host of tactics to optimize models prior to the deployment step. Figure 2.1 highlights some of these methods.



**Figure 2.1** TensorRT model optimization strategies. [Source](#).

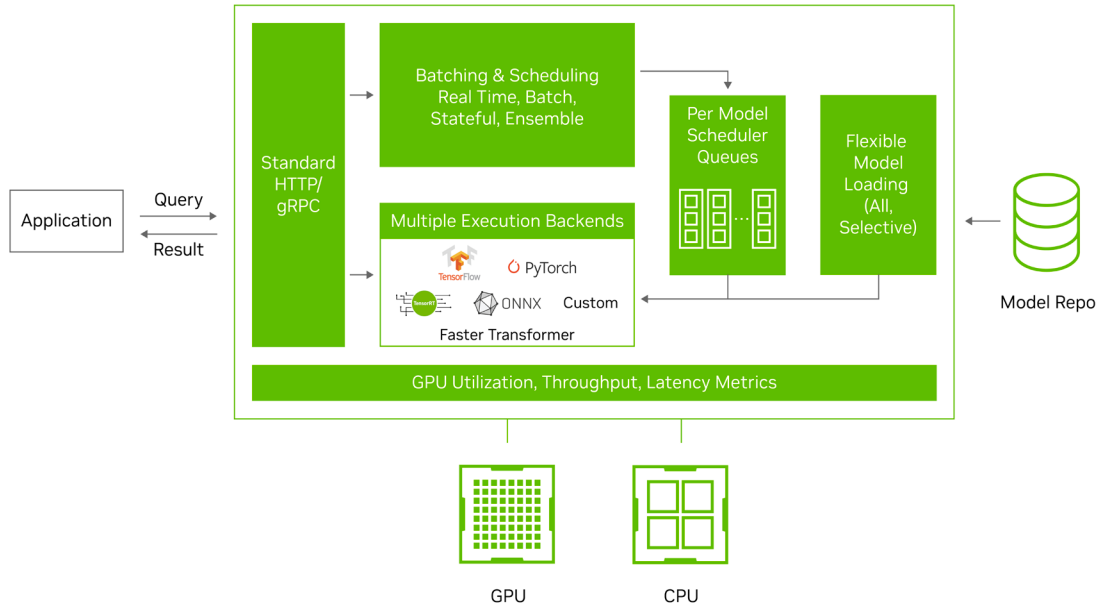


Each numbered balloon in the figure represents a crucial piece of the model optimization strategy. Let's look at those pieces in a bit more detail, listed here with the corresponding number from the figure. You will find much more detail about TensorRT and each of these features at [NVIDIA TensorRT](#).

- 1** *Weight and Activation Precision Calibration:* In order to maximize throughput while maintaining accuracy, TensorRT quantizes the parameters of the development model to FP16 or INT8 (lower precision number formats for faster computation) with calibration data in the loop. While quantizing, the optimization algorithm can thus maintain, or nearly maintain, baseline performance.
- 2** *Layer and Tensor Fusion:* When a single operation can provide approximately the same behavior as multiple operations, TensorRT combines them. For example, if sequential shuffle operations randomize the order of values, a single shuffle can achieve the same goal. Using this logic, the optimizer can reduce computation of a variety of layer combinations.
- 3** *Kernel Auto-Tuning:* By selecting the best possible CUDA kernels (i.e., functions executed on a GPU) for operations internal to the model, the model optimizer can further improve computational performance. This ensures that predictors follow more optimal computational pathways on GPU hardware.
- 4** *Dynamic Tensor Memory:* Tensors allocated during intermediate computation for inferences can occupy a nontrivial amount of memory. By deallocating tensors after they are no longer necessary, one can minimize the model's memory footprint. Minimizes memory footprint and reuses memory for tensors efficiently.
- 5** *Multi-Stream Execution:* By executing computational sequences in parallel, overall throughput is maximized along with device utilization. Scalable design allows the system to process multiple input streams in parallel.
- 6** *Time Fusion:* Recurrent neural networks use the same model parameters to iteratively process a sequence of inputs. The model uses intermediate outputs from prior time steps along with intermediate inputs from the sequence until a final output is produced. By using dynamically generated kernels, the optimizer can streamline computation over sequential time steps. Optimizes recurrent neural networks over time steps with dynamically generated kernels.

Though not comprehensive, these approaches allow TensorRT to take base models from a variety of source frameworks (e.g., PyTorch) and optimize runtime execution. Once optimized, the model can be used in conjunction with the Triton framework for deployment.

The Triton Inference Server is an open-source framework that creates a standard interface for high performance model serving. Regardless of the development framework, Triton supports a host of model backends while still enabling a multitude of runtime features that could be expected of a modern, capable inference service. Figure 2.2 shows an overview of the Triton server framework, components, and workflow.



**Figure 2.2** NVIDIA Triton Inference Server overview. [Source](#).

Many of the Triton components in figure 2.2 (such as the application and batching service) you should recognize from the generic configuration we showed you in chapter 1 (figure 1.4). Some additional features in the Triton system are worth pointing out. For instance, the flexible model loading from a connected model repository allows for easier model management and deployment, especially as new artifacts are generated over time. In order to minimize time to develop iteratively, Triton also includes bundled tools like the Model Analyzer, which allows the developer to profile a deployed model and further optimize the final configuration.

When used with a TensorRT-optimized model, the Triton server affords the developer a tool to deploy reliable infrastructure that is robust against demanding production workloads. We'll look at Triton's features in action as we go through the real-life case studies in this chapter.

## 2.2 Inference in natural language processing (NLP)

Natural language processing (NLP) refers to a class of algorithms that derive insight from human language (including speech or text). Tasks such as sentiment analysis, machine translation, text generation, and text search have already become an important part of living in the world of today. When translating a phrase from a foreign language, searching for an article, or checking your written grammar in a word processor,

NLP drives the backend that allows so much power at your fingertips. Let's look at some examples in practice. (At the end of each example, we provide references and resources for further reading.)

### 2.2.1 Amazon product search

#### CHALLENGE

Amazon aims to provide an effortless user experience with their product search. In practice, the platform places a heavy emphasis on highly accurate spelling correction, because spelling errors can put a barrier between users and the content they are targeting. Though deep learning has provided measurable improvements over previous classical methods, the increased accuracy incurs a steep computational cost. While running experiments, Amazon discovered that the Text-to-Text Transfer Transformer (T5) was highly performant and worth replacing the previous backend predictor. However, deployment of the T5 model at any appreciable scale proves to be difficult due to its sheer size and computational demand. The key challenges the system developers had to address were latency, throughput, and cost efficiency. To meet production demand, Amazon needed to achieve a sub-50-millisecond P99 latency (i.e., a sub-50-millisecond latency in at least 99% of cases).

#### SOLUTION

In the production solution, Amazon employed the TensorRT framework to optimize model inference and Triton to do the work of deployment. When leveraged in tandem, TensorRT's model optimization tactics, along with the options offered by the Triton Inference Server (batching, queueing, model profiling with the Model Analyzer, etc.), allowed Amazon to reach their production requirements and avoid incurring unnecessary compute costs. The combination of features that both tools provided enabled widespread deployment of the T5 product search model, despite its computational complexity.

#### OUTCOME

When tested on both NVIDIA T4 and A10G GPUs, speedups ranging from 400% to 760% were seen consistently across various model configurations and precisions. In the most extreme case, the t5-base model latency was improved from 60.0 to just 8.5 milliseconds on the A10G GPUs (using a g5.xlarge instance on AWS). In effect, leveraging the TensorRT optimization framework enabled Amazon to make a prototype model production-ready.

Similarly, the Triton inference server enabled dynamic batching on the server side that effectively optimized GPU utilization in addition to maintaining a synchronous near-real-time experience on the client side. Overall, these techniques constituted a powerful deployment strategy that not only performed optimally but allowed for a much more seamless user experience due to tolerance for misspelled or otherwise misleading queries.

**FURTHER READING**

- [“How Amazon Search achieves low-latency, high-throughput T5 inference with NVIDIA Triton on AWS”](#)
- [“Optimizing T5 and GPT-2 for Real-Time Inference with NVIDIA TensorRT”](#): Interactive model optimization walkthrough

**2.2.2 Microsoft Translator****CHALLENGE**

Translator is a part of Microsoft Azure Cognitive Service that helps people communicate with one another. More specifically, it is a powerful API that allows developers and users to perform language translation. “Our vision is to eliminate barriers in all languages and modalities with this same API that’s already being used by thousands of developers,” the development manager for Translator has said. With some 7,000 languages spoken worldwide, it’s an ambitious goal. However, production-quality language models are generally very compute-heavy, which makes the transition to deployment a challenging process. To make matters worse, the team eventually selected a transformer-based mixture of experts (MoE) model to perform the translation. The final architecture has more than 5 billion parameters and is 80x larger than the second largest NLP model under Microsoft’s purview. Nevertheless, the team targeted sub-two-second latency per translated document.

**SOLUTION**

In order to frame the problem in a way that’s easily parallelizable, Translator breaks a given document translation request into a batch with roughly hundreds of sentences per sample. In practice, this provides the local context needed to make a translation possible, while removing the need to process a large document with a single forward pass of a model. However, the work of optimizing the translation latency of a single sample prediction still remains.

The team turned to NVIDIA’s Triton inference server on Microsoft Azure cloud infrastructure to handle the prediction optimization. In particular, features such as dynamic batching and model optimization specific to transformer architectures allowed for maximizing compute utilization and therefore minimizing overhead cost of inference.

**OUTCOME**

As expected, the team was able to improve prediction latency over non-optimized GPU runtime predictions. In fact, during preliminary prototyping, they were able to speed up prediction by a factor of 27x. This work is expected to extend directly to production; however, the rollout process is ongoing. The team plans to release the new Translator architecture for a few key languages, and then progressively add support for all known languages. Though ambitious, the team has taken major strides toward achieving this goal. Optimized deployment technologies allowed Microsoft to deploy a very complex and compute-heavy model in order to keep translation quality high, thereby improving user experience as well as the overall utility of the system.

**FURTHER READING**

- “Getting People Talking: Microsoft Improves AI Quality and Efficiency of Translator Using NVIDIA Triton”
- “Scalable and Efficient MoE Training for Multitask Multilingual Models”

**2.3 Inference in computer vision**

Computer vision (CV) refers to a host of algorithms that process vision data. Whether digital photos, video, or some other visual data source, the algorithms often operate on raster-based input formats (single- or multi-channel pixel grids) to generate inferences. Many modern model architectures draw inspiration from the human visual cortex, leading to a host of creative and computationally intensive solutions. Let’s look at how this plays out in industrial applications.

**2.3.1 Siemens Energy autonomous plant inspections****CHALLENGE**

Siemens Energy is a leading supplier of power plant equipment and technologies and has a massive portfolio of machines and sites to service. Siemens Energy is responsible for tens of thousands of gas turbines, steam turbines, generators, and gas and diesel engines. In addition, market pressures from the renewable energy industry have created more scrutiny on the efficiency of traditional power generation. Because of this, Siemens Energy has embarked on an automation journey that will help reduce the overhead costs of keeping their existing infrastructure online.

Inspections, in particular, place a lot of sometimes unpredictable demand on the power generation workforce, which is both aging and declining in Europe. In addition, hundreds of inspection types are currently performed via human walk-throughs and are required to detect and address issues that, left unchecked, could create damage costing millions of dollars. These issues amount to oil leaks, undesired steam, spills, and other issues. Siemens Energy needed a solution to deploy a multitude of computer vision models trained to address these issues in a scalable manner. In particular, they wanted to avoid changing their hosting solutions because they hosted models for different kinds of analytics. Additional key requirements included a need for occasional edge deployment, as well as pre-processing capabilities, as tasks like person anonymization were required prior to making inferences.

**SOLUTION**

Siemens Energy ultimately decided to leverage Triton Inference Server to tackle their inference needs. The multi-model, preprocessing, and edge support capabilities are key features that enable production-quality outcomes. In addition, AWS was leveraged to host the backend infrastructure, which enables scale across the geographic areas serviced by Siemens Energy.

**OUTCOME**

Siemens Energy now conducts visual inspections using computer vision models as part of routine operations. The product management team points, in particular, to the flexibility of the Triton solution, which enables autonomous monitoring of complex power plants whose sensors and cameras may use legacy software. Though the current solution has the capacity to run inference in edge scenarios where data export is prohibited, the enterprise plans to progressively integrate these edge devices into their production infrastructure. By adopting automated visual inspection technology, Siemens Energy could address labor market gaps, increase inspection efficiency, and minimize risk of costly downtime.

**FURTHER READING**

- [“Electrifying AI: Siemens Energy Taps NVIDIA Triton Inference Server for Power Plant Inspections, Autonomy”](#)

## 2.4 *Inference in recommender systems*

Recommender systems do exactly what their namesake implies: recommend things. Frequently these algorithms provide suggestions for users based on browsing history, the behavior of users similar to the target user, as well as additional sources that ultimately increase the likelihood of further engagement. However, a host of ranking algorithms and statistical techniques underpin these architectures so that the “best” (however the business defines this word) content is made available to the user, client, or customer. Here we will outline how this can be achieved in practice.

### 2.4.1 *Snap recommendations*

**CHALLENGE**

Every year, eCommerce is responsible for trillions of dollars in sales worldwide and serves billions of consumers. Recommender systems live at the heart of these platforms. Utilizing these powerful systems results in a more engaging experience for the user as well as an increased revenue for the digital retailer.

However, to provide better recommendations, there exists incredible motivation to make predictive models bigger, better, and faster. The computational demand associated with these changes demands an optimized and scalable solution.

Snap, the parent company to social media app Snapchat, services more than 300 million daily active users. Ads served on their platform provide a primary revenue stream; therefore, it is paramount that Snap prioritizes content ranking (and therefore recommendation) performance that maximizes ad engagement. Though Snap has been able to produce models that produce high quality ad ranking, the computational load of the baseline is prohibitive to deploy on the production infrastructure.

**SOLUTION**

Snap used NVIDIA GPUs and Merlin to boost its content-ranking capabilities. Snap leveraged Merlin for trained model optimization in preparation for inference. By

leveraging this framework, Snap enabled the creation of highly performant models that are optimized for inference on NVIDIA GPU hardware.

### OUTCOME

Snap was ultimately able to build and deploy models trained within the Merlin framework, and because of the inherent focus on optimized inference, was able to reach target cost and performance goals simultaneously, including a 50% increase in cost efficiency for the inference procedure and a two-fold decrease in latency. That reduced latency, in turn, freed up compute power to improve the accuracy of their models to better serve their advertising partners. It's clear that in practice, model performance achieved in a development environment can be transferred to the production setting when appropriate care and design forethought are employed. Furthermore, by avoiding a sacrifice in accuracy in the deployed context, Snap was able to maintain a high click-through rate (CTR) and thus drive more ad revenue relative to a model with lesser performance.

### FURTHER READING

- “Billions Served: NVIDIA Merlin Helps Fuel Clicks for Online Giants”
- “NVIDIA Merlin HugeCTR”

## 2.5 Inference in fraud detection

By abusing certain systems and institutions, attackers sometimes use manipulative tactics to seek illegitimate personal gain. This fraudulent activity results in losses and inefficiencies that incur costs to mitigate, in addition to the baseline cost of the fraud. When deploying fraud detection algorithms, institutions can fight back and limit damages, thereby protecting legitimate business operations. When you lose a credit card and your bank calls you before you've even realized you lost it, it is certain that fraud detection tools detected an attempt at misuse. Here we will see how this technology can be scaled to protect financial customers.

### 2.5.1 American Express (AMEX) fraud detection system

#### CHALLENGE

Cybercrime costs the global economy \$600B annually, or nearly 1% of worldwide GDP, according to an estimate in 2018 from [McAfee](#). AMEX alone is responsible for more than 8 billion transactions per year. With more than 115 million active credit cards, AMEX has maintained the lowest fraud rate in the industry for 13 years in a row, according to the [Nilson Report](#) (see also [BusinessWire](#)). However, because of the inherent adversarial dynamic between fraudsters and financial institutions, the institutions must continually adapt to combat novel fraud strategies. In addition, online transactions are on the rise, which puts pressure on the financial industry to serve real-time models at a global scale.

**SOLUTION**

Because transaction history is such a crucial part of making fraud determinations, AMEX selected a model architecture capable of consuming sequences as inputs. After some experimentation, the AMEX team selected the long short-term memory (LSTM) deep neural network architecture to do the work of transaction classification. Though highly performant, as in many other cases within this text, the model incurred a large computational cost. The team was able to leverage NVIDIA DGX systems (multi-GPU on-prem compute servers) to perform the model training. Once created, the model artifacts were optimized using the TensorRT framework and deployed using a Triton Inference Server with NVIDIA T4 GPUs.

**OUTCOME**

When combined with the company's long-standing gradient boosting machine (GBM) model, the LSTM model can improve fraud detection accuracy by up to 6% in specific segments. As deployed, the architecture can generate inferences with a sub-2 millisecond latency, which is a generally unattainable target metric. Clearly, if intentional design decisions are made, extremely low-latency deep neural network models can be deployed at a global scale. The Machine Learning and Data Science Research team at AMEX feels they have created "best-in-class fraud protection and servicing." The use of low-latency fraud detection algorithms at scale enables AMEX to both deliver optimal customer experience as well as mitigate lost revenue due to fraud mitigation.

**FURTHER READING**

- [“American Express Adopts NVIDIA AI to Help Prevent Fraud and Foil Cybercrime”](#)



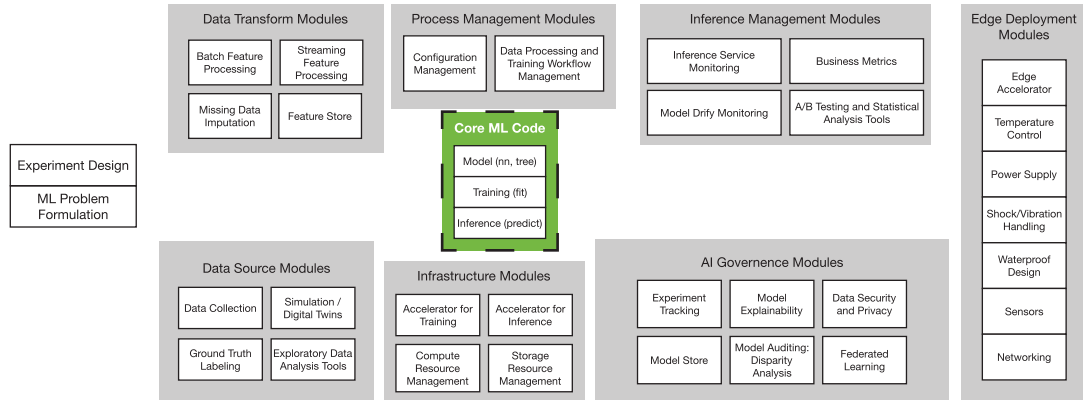
# *AI inference in practice*

---

Now that you understand the concepts behind inference deployment and have seen real-world case studies showing the numerous ways that it is used in practice, let's look at the process of actually implementing it. In this chapter, we will cover challenges, tools, and processes to deploy inference in practice. At the end of the chapter, we present a code lab for you to apply the processes you learn here. In this lab, you will build a reverse image search model server which will give you hands-on experience with end-to-end model training and deployment.

## **3.1 Challenges of inference deployment**

The seminal paper “[Hidden Technical Debt in Machine Learning Systems](#)” states “only a small fraction of real-world ML systems are composed of the ML code.” As the paper points out, “the required surrounding infrastructure is vast and complex” compared with the machine-learning algorithms at the center of it all. A visual, figure 3.1, will drive this point home. Look at the tiny box in the center marked *Core ML Code* and you will get an idea of its size relative to the infrastructure. As you can see, to make machine learning operational in production, a number of other modules—including data transform, AI governance, process management, and infrastructure—all need to work coherently.



**Figure 3.1** Machine learning code, represented by the small box in the middle, is only a small piece of the complex infrastructure in machine-learning systems. This figure is inspired by a similar figure from the paper “Hidden Technical Debt in Machine Learning Systems,” but updated with more recent tools and modules for contemporary machine learning in production.

So, the overarching challenge of inference deployment is getting it to work well with all the other components in the system. Here we briefly walk through this and other common challenges based on our first-hand experience deploying inference in production.

### 3.1.1 Latency and throughput

Real-world applications often require the inference to have low latency and high throughput. For example, safety-critical applications, such as autonomous driving, place strict requirements on throughput and latency expected from deep learning models. The same holds true for most consumer applications, including recommendation systems. In real-time ad bidding systems, the inference is expected to be done within several milliseconds.

### 3.1.2 Integration with production data

It is not uncommon to have pre-processing and post-processing steps in the inference process. For example, the pre-processing step can be used to normalize the data, and the post-processing step can be used to convert the output to a desired format.

A common pitfall is to use different feature transformations in model training than the feature transformations in the live inference. The feature transformations are closely coupled with the model. They work hand in hand. Besides manually defined feature-extraction logic, vocabularies, lookup tables, normalization and scaling parameters are generated during the training process. These parameters control how feature transformations are done. As data scientists evolve the training recipes, the feature transformation logic and parameters can change from one version of a model to the next. If a new model is deployed without updating the version of its feature transformations, the

inference may deliver underwhelming accuracy in production. For example, consider a model using the product's height in inches as a feature. An update is made so that the height is now measured in centimeters. If the production feature pipeline is not updated to reflect this change, the inference can behave poorly.

### **3.1.3 Handling different deployment platforms**

Cloud (AWS, GCP, Azure), on-premise servers, mobile devices, and IoT devices are all possible locations to deploy inference. With so many choices of platforms, each with different toolchains or SaaS products, it can be overwhelming to make the right choice.

### **3.1.4 Handling different types of models, model architectures, and ML frameworks**

Linear models, tree-based ensemble models, convolutional nets, recursive neural networks, transformers, and many other types of models are used to solve different problems. A wide variety of ML frameworks are used to train models: TensorFlow, PyTorch, JAX, MXNet, Scikit-learn, XGBoost, etc. As the model and framework differ, the best practices for deploying and integrating also differ. Navigating the complex landscape is a challenge for data science and engineering teams.

### **3.1.5 Scaling to multiple models and complex prediction flows**

Multiple models can be involved in the inference. For example, a model can be used to detect non-trivial movement in a camera stream and trigger the follow-up analysis, and a second model can be used to detect persons, packages, and other fine-grained information.

### **3.1.6 Minimizing downtime for rollout**

The ability to deploy new inference with minimal downtime can be critical to many consumer-facing applications. Server interruption may mean lost revenue, poor user experience, and added operational burden. Tools that support smooth deployment and rollout, as well as warm start, can be very helpful.

### **3.1.7 Resource optimization**

As deep-learning models become more and more accurate, their size also increases. The amount of compute, memory, and storage required to run a deep-learning model can be very large. Thus, it is important to have the ability to profile a model for its resource consumption and to optimize the model to reduce its resource requirement.

### **3.1.8 Monitoring**

As inference servers scale to many models serving millions of requests, it is important to have visibility of the health of the service. Is the server overloaded? What are the actual latency and throughput? How much is resource consumption? Is there a memory leak?

Are there anomalies in the inference server? Reliable tools are needed to answer these questions so that Dev-Ops and ML-Ops teams can make informed decisions.

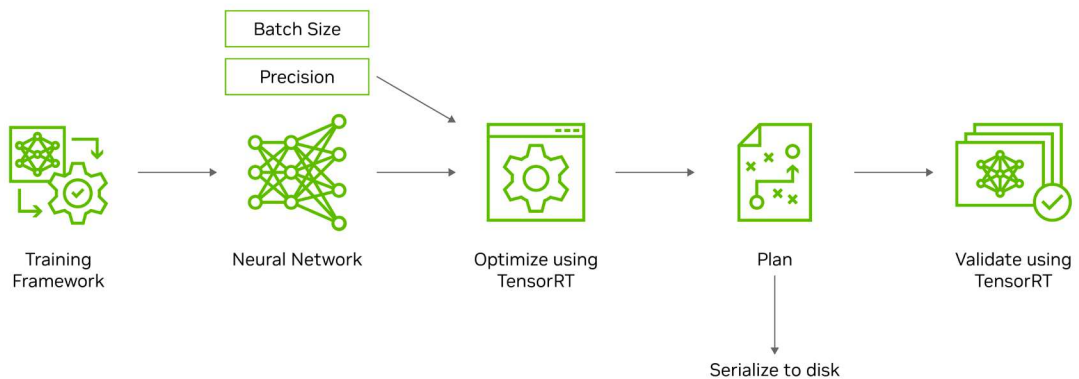
This list of challenges we've just covered is in no way exhaustive, but it is a good starting point to help you understand the need for selecting the right tools and processes for each project.

Next, we look into two tools, TensorRT and Triton Inference Server, that alleviate many of the pain points we've just described. You've already been introduced to these tools; here we will look at how to work with them.

### 3.2 Optimize models with TensorRT

TensorRT is an SDK to optimize a trained machine learning model to have low latency and high throughput on a specific GPU.

Figure 3.2 shows a typical development workflow for using TensorRT.



**Figure 3.2** A typical development workflow for using Tensor RT.

As you can see, after data scientists and machine learning engineers train models using a framework of their choice, they need to optimize the model file. TensorRT can be used for this post-training optimization, producing an optimized model file for a target GPU device. The optimized model is then deployed to serve production traffic.

#### 3.2.1 Precision and speed

TensorRT supports computations using data types of different precisions including FP32, FP16, INT8, Bool, and INT32.

Therefore, you can easily instruct TensorRT to use FP16 calculations for your entire model. For regularized models whose input dynamic range is approximately one, this typically produces significant speedups with negligible change in accuracy.

### 3.2.2 Quantization

TensorRT supports quantized floating point, where floating-point values are linearly compressed and rounded to 8-bit integers. This significantly increases arithmetic throughput while reducing storage requirements and memory bandwidth. When quantizing a floating-point tensor, TensorRT must know its dynamic range—that is, what range of values is important to represent—values outside this range are clamped when quantizing.

### 3.2.3 API languages

TensorRT’s API has language bindings for both C++ and Python, with nearly identical capabilities. The Python API facilitates interoperability with Python data processing toolkits and libraries like NumPy and SciPy. The C++ API can be more efficient, and may better meet some compliance requirements, for example, in automotive applications.

## 3.3 When to use TensorRT

First, TensorRT, like similar SDKs, is used *after* you have trained a model with a deep-learning framework. It is not intended to train or fine tune a model.

Second, TensorRT is used to optimize a model for a specific GPU. If you plan to deploy inference on GPU devices in the cloud (such as V100) or the edge (such as Jeston Nano), then TensorRT is great for you. Otherwise, tools like OpenVINO can be handy for optimizing deep learning models for CPUs.

### 3.3.1 Optimize Tensorflow models

TensorRT is closely integrated with TensorFlow. Figure 3.3 illustrates the workflow of optimizing a TensorFlow model.

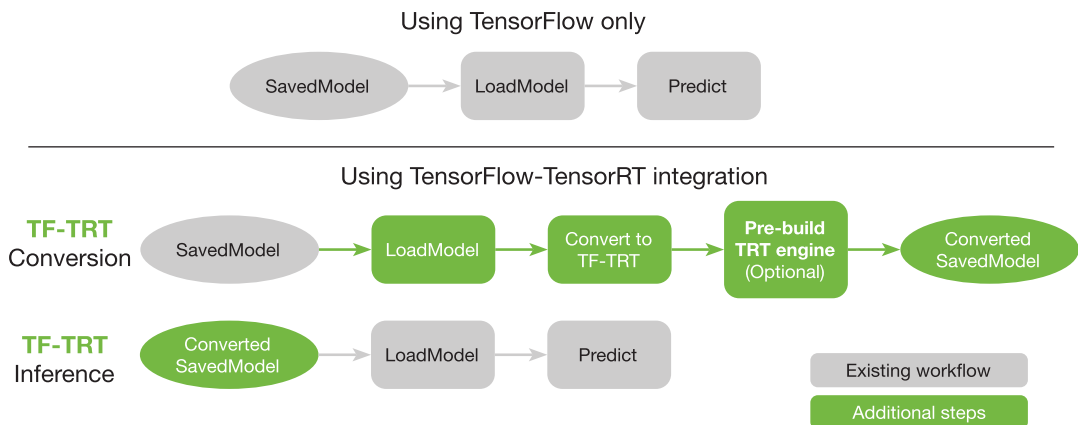


Figure 3.3 Optimizing a TensorFlow model.

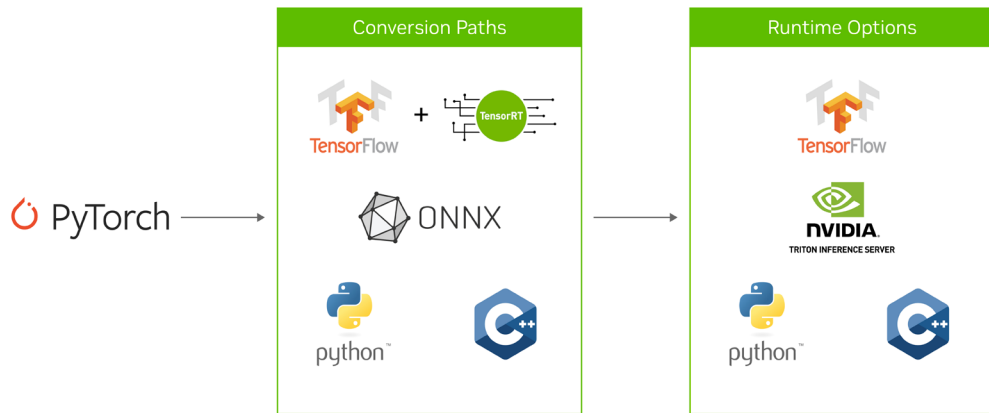
A TensorFlow model is usually stored in the “SavedModel” format, which is then loaded and used in inference. With TensorRT, a series of steps is carried out to first convert this SavedModel into an optimized version. This optimized SavedModel can be loaded in the same way for inference. The additional steps can be integrated seamlessly with the existing workflow.

### 3.3.2 Optimize PyTorch models

Torch-TensorRT is an integration for PyTorch that leverages inference optimizations of TensorRT on NVIDIA GPUs. With just one line of code, it provides a simple API that gives up to 6x performance speedup on NVIDIA GPUs.

Torch-TensorRT acts as an extension to TorchScript. It optimizes and executes compatible subgraphs, letting PyTorch execute the remaining graph. PyTorch’s comprehensive and flexible feature sets are used with Torch-TensorRT that parse the model and apply optimizations to the TensorRT-compatible portions of the graph.

As figure 3.4 indicates, there are multiple pathways of converting a PyTorch model using TensorRT. Multiple runtimes are supported, including Tensorflow, custom python and C++.



**Figure 3.4** Supported conversion paths and runtimes for PyTorch in TensorRT.

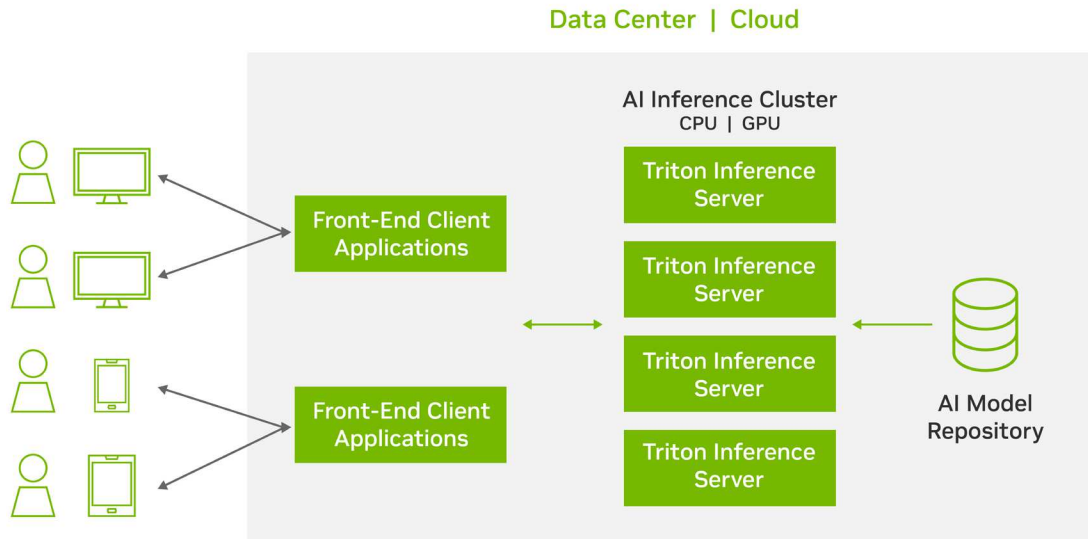
After compilation, using the optimized graph is like running a TorchScript module and the user gets the better performance of TensorRT.

## 3.4 Deploy inference with Triton Inference Server

After the model is optimized, it needs to be deployed via an inference server framework. The NVIDIA Triton Inference Server is an open-source solution for deploying deep-learning models on both CPUs and GPUs, with support for a wide variety of frameworks and model execution backends, including PyTorch, TensorFlow, ONNX,

TensorRT, XGBoost, LightGBM, Scikit-Learn Random Forest, and RAPIDS cuML Random Forest. Triton's features include dynamic batching, model ensembling and CPU/GPU execution. Its Docker container integrates with hosted Kubernetes services such as AWS EKS, Google GKE, and Azure AKS. And it is also available in Managed CloudAI workflow platforms such as Amazon SageMaker, Azure ML, and Google Vertex AI.

For more information about using Triton, please refer to the following link: <https://developer.nvidia.com/triton-inference-server/get-started>.



**Figure 3.5** Triton Inference Server framework on a highly scaled production deployment.

Figure 3.5 shows Triton at work. As shown in the figure, Triton can operate on a compute cluster, serve various front-end applications and end users, and provide dynamic model substitution using a connected model repository, all on self-hosted or cloud infrastructure. These features are reliable in demanding production environments, making it a likely choice for deployments at a scale similar to those mentioned in the case studies in chapter 2.

### 3.4.1 When to use Triton

Triton can be used in a range of scenarios. It can run on a CPU, GPU, and other accelerators. It can run on beefy machines in the cloud and also small edge computers such as Jetson Nano. Models trained with most major machine learning frameworks can be deployed using Triton.

Triton does require the model files and associated metadata to be organized in a certain way. But you may find it to be a small overhead compared to the potential benefits in latency, throughput, and a well-structured deployment process.

## 3.5 Recipes for different data types

In this section, we focus on practical recipes for different data types when deploying inference.

### 3.5.1 Tabular data

Ubiquitous in business applications, tabular data is data organized in a table consisting of rows that share the same set of columns. Many use cases rely on tabular data, such as predicting the probability of a user clicking on a search result, forecasting customer churn, and predicting if a medicine is effective on a group of patients. This is a domain where both deep learning and more traditional machine-learning models may shine. Gradient-boosted trees and sometimes logistic regression models demonstrate great accuracy when the problem can be formulated as a standard classification or regression problem. When multi-task learning, or a highly customized prediction path is needed, deep-learning models can provide additional flexibility.

#### TRAINING-INFERENCE DATA GAP

With tabular data we dive deeper into the common pitfall in deploying models: the gap and inconsistency between training and inference data. Often, a complex data pipeline is used to prepare training data. Different tables are joined. Feature aggregation, bucketing, vocabulary building, encoding, and feature crossing are applied. When data is missing or incomplete, imputation is done to backfill data. All of these steps have various parameters.

Now at inference time, what provides the data is often a separate data pipeline optimized for production-level speed and reliability. If the version of data schema and feature transformation in production differs from offline model training, the model's accuracy may be much worse than demonstrated during offline evaluation. It is very easy to make this mistake, even when great care is taken. There can simply be too many parameters and data schema versions to keep track of. Adding to the complexity, some data available during training may not be readily available during inference, due to lag of data logging, ingestion, and processing that can happen upstream for various reasons. This problem is amplified when there are multiple models that depend on one another's output. An error in the input data can propagate multiple times and cause larger and larger deviation from expectation.

Mitigation strategies include diligent testing and monitoring. Unit and integration tests can be put in place to test the parity between the feature transformation used in model training versus that deployed in production. If the feature transformation in production gives different results from the one used in model training, there is likely an error in the version of data schema, feature transformations, or their integrations. Tests can uncover such issues early, preventing problematic deployments.

After deployment, monitoring is a powerful tool to surface data problems. It is good to have parity between statistics of training data and that of live production data. Here is an incomplete list of metrics one should look for when doing monitoring:

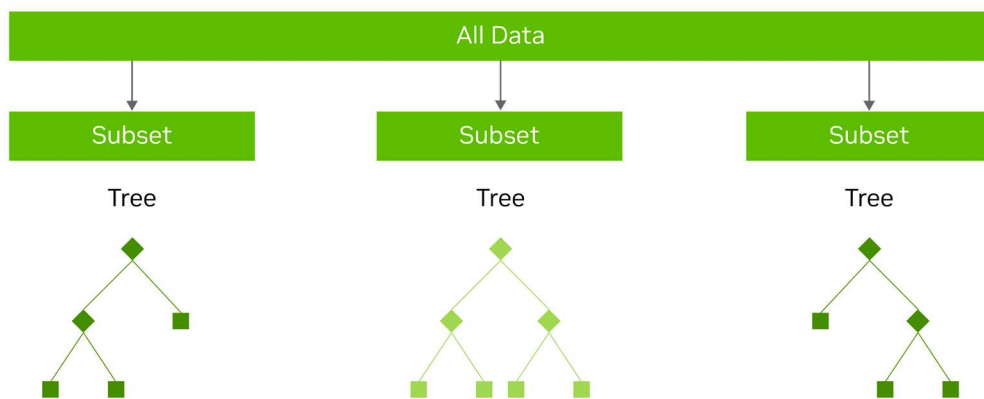


- Feature shapes
- Minimum, maximum, mean, and standard deviation of feature values
- Number of NaN and missing values
- Word clouds or pie charts for vocabularies and categorical variables that have new values over time
- Sudden increase or decrease of average feature values

These metrics can be tracked and visualized in a way that is easy to understand, share, and present to the stakeholders.

#### INFERENCE USING TREE-BASED MODELS

With Triton, you can use the FIL backend for tree-based models implemented in [XGBoost](#), [LightGBM](#), [Scikit-Learn](#) or [cuML](#). Figure 3.6 shows schematic illustration of a typical workflow to produce a tree-based ensemble model.



**Figure 3.6** How a tree-based ensemble model is produced.

As you can see, a tree-based model typically includes multiple decision trees, and its prediction is the weighted average of prediction from individual decision trees.

#### INFERENCE USING DEEP-LEARNING MODELS

A wide variety of deep learning models have been proposed for tabular data, such as [wide-and-deep](#), [TabNet](#), [non-parametric transformers](#), and [1D CNN](#). Both PyTorch and Tensorflow implementations of these models are publicly available and can readily be deployed with Triton.

### 3.5.2 Time series data

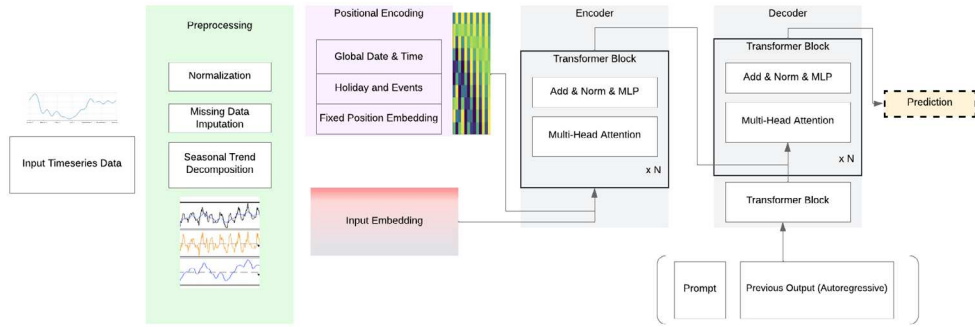
Time series data refers to data where one or more variables are recorded over time, most often at a consistent interval. Businesses use time series data to assist in sales forecasting, predictive maintenance, patient health trajectory forecasting, algorithmic trading, and many other use cases. Most best practices for tabular data also apply to time series data. However, time series data also has some unique challenges:

- The trend in the data tends to change in the long term. Models that work well in one historical time period may not be accurate for future data.
- Beside long-term trends, seasonality is typically present in time series data. There can be different granularities of seasonality at varying time scales, including monthly, daily, or hourly.
- Because the long-term trend and seasonality may change due to market shift and rare global events, the model needs to be refreshed with new data. It is also desirable to avoid over-confident predictions which fall apart when dramatic changes happen.
- Multi-step prediction is common, where prediction is done progressively for one time period after another. A prediction can depend on the prediction of the previous time period. As a result of this recursive inference, errors can propagate as we predict further into the future.
- The inference system may need to handle inconsistencies from different time zones and transitions like daylight saving time.

Modeling techniques for time series data have evolved over time, resulting in a range of model types implemented in different machine-learning frameworks.

Statistical models such as [ARIMA](#) and [SARIMA](#) are well studied and often provide a strong baseline for a wide range of time series problems. Careful hyperparameter tuning may be needed for ARIMA models to work well. [Prophet](#) (or [FBProphet](#)) further decomposes time series data into a long-term trend, seasonality, and change points.

Deep-learning models such as RNN, LSTM, and Transformers have gained popularity in recent research. The sequence nature of time series data makes it suitable to be modeled by these architectures. There have been hundreds of recent research papers on this topic in the last three years. As an example, [Autoformer](#) was shown to be effective in energy, traffic, economics, weather and disease use cases. [Merlion](#) is a recent open-source library that supports classic statistical methods, tree ensembles, and deep-learning approaches. The graph in figure 3.7 illustrates the architecture of typical transformer-based models for time series.



**Figure 3.7** A typical architecture of transformer-based models for time series data.

Let’s do a brief walkthrough of figure 3.7 to give you a broad idea of how the architecture is arranged. The data is preprocessed with normalization and missing data imputation. Sometimes it is decomposed into seasonal components and long-term trends. Then, positional encoding is performed on the global date and time (year, month, day) and holidays. The positional encoding is combined with the embeddings of other input variables to form the input to the Encoder module. The Encoder module consists of multiple Transformer blocks with Multi-Head Attention layers. At inference time, the Encoder output is combined with an optional “prompt” and passed to the Decoder module. The Decoder module either outputs single-valued predictions or repeatedly produces a sequence of output tokens in an auto-regressive fashion.

In terms of deployment, statistical models are usually implemented in SciPy, NumPy, statsmodels and stan, and, with the Triton server, can be deployed using Triton’s custom [Python backend](#). Deep-learning models are often implemented in PyTorch (this is the case for both Autoformer and Merlion) or Tensorflow, with the corresponding backend in Triton.

### 3.5.3 Image and video

The trend in computer vision models has been higher accuracy as the size and depth of the model increases. Two types of models have gained popularity recently:

- Vision transformers that incorporated self-attention mechanisms to model long-range interactions of image features.
- ConvNext and other improved convolutional networks that improved upon ResNet, the previous state-of-the-art, by increasing network width, tuning residual block shapes, and using better optimization algorithms.

These are backbone or foundational models that can be used as building blocks for downstream tasks such as semantic segmentation, object detection, and image retrieval.

### CONSISTENCY OF PRE-PROCESSING BETWEEN TRAINING AND INFERENCE

The pre-processing of images can significantly influence the prediction output. So, it is important to make sure that the pre-processing is consistent between training and inference. The pre-processing steps also benefit from GPU acceleration and Triton provides readily made modules to run pre-processing in addition to inference (examples can be found in this [link](#)). Here are some common pre-processing steps.

#### RESIZING

Many pre-trained models require input images to be resized to a constant shape such as 224 x 224 pixels. While the inference may still work when you supply it with an image of different size (if the model is fully convolutional), the prediction accuracy tends to be lower.

Another common resizing option is to resize the longer side of an image to a target dimension while keeping the aspect ratio. This method avoids distortion of the image and is often used for object detection models.

#### NORMALIZATION

Three types of normalization for pixel intensity values are commonly used. One is standardization with pre-computed stats, where the pixel intensity is subtracted from the mean of a dataset and divided by the standard deviation of the dataset. The second type is per-image standardization, where each image is normalized to have zero mean and unit variance, based on mean and standard deviation of the specific image. Additionally, there is min-max normalization, where the pixel intensity is normalized to the range between 0 and 1.

The choice of the normalization method tends to have little impact on the accuracy. But it is important to keep it consistent between training and inference.

#### TEST-TIME AUGMENTATION

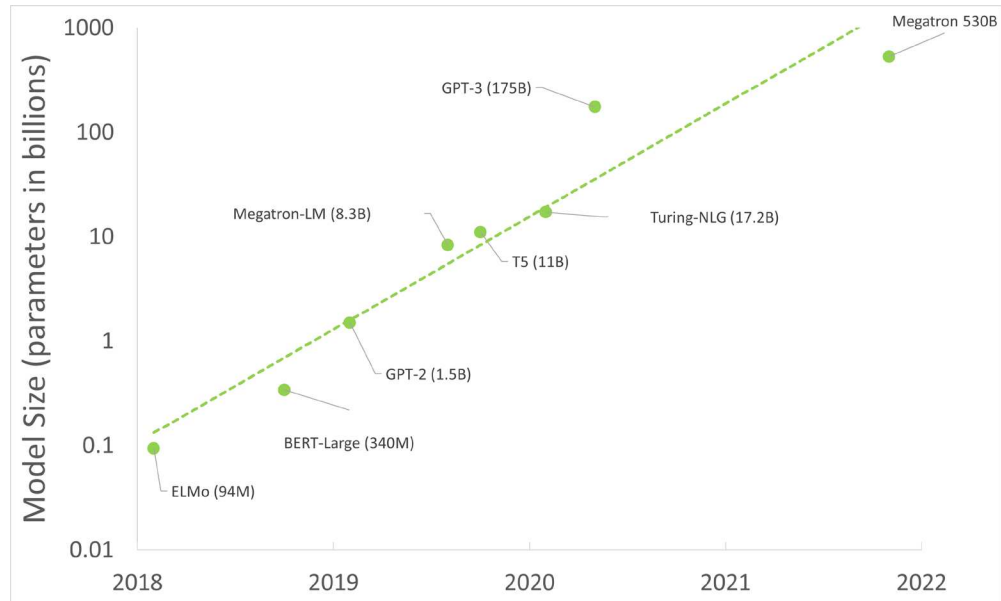
Image augmentation, usually applied during model training, generates more variety in the training data and helps the model generalize better to new data. Its counterpart, test-time augmentation (TTA), is the aggregation of predictions across transformed versions of a test input, which is shown to [improve prediction accuracy](#). The transformations include flipping, cropping, and scaling of the input image. They are easy to apply and do not change the model itself. The downside is that more compute is needed as the number of transformations increases.

#### EFFICIENT VIDEO PREPROCESSING WITH KEY FRAMES

Inference on video can pose a scale challenge, because a video is a long sequence of images, with thousands of frames. Making use of keyframes and delta frames can greatly reduce the number of images needed to be processed. To track objects over time and compute pixel correspondence from one frame to the next, block motion vectors in MPEGs can be used in lieu of the more computationally expensive optical flow.

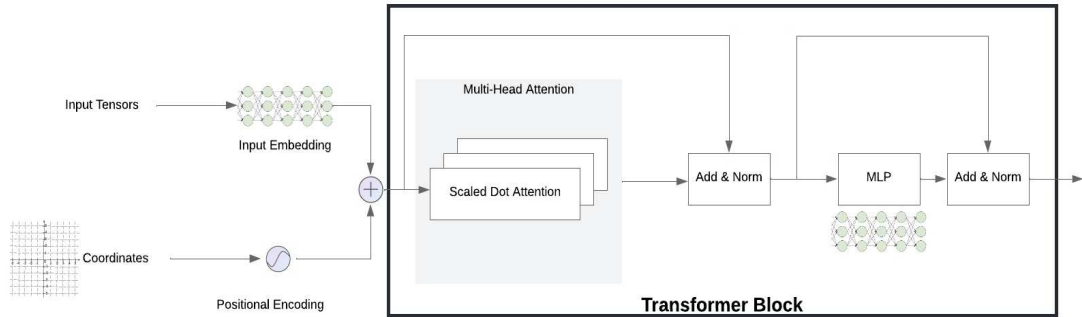
### 3.5.4 Natural language

Similar to computer vision models, large language models got more accurate and larger over time. The graph in figure 3.8 shows the size of the model over years. From the 94 million parameters of ELMo in 2018 to 175 billion parameters of GPT-3 in 2020, the increase was more than 100 times.



**Figure 3.8** The size of state-of-the-art NLP models has increased steadily over the years. (Source.)

**Transformer-based architectures** drove much of the recent progress in the accuracy of language models. Figure 3.9 shows a typical architecture of Transformer models. Different from other types of neural networks, a Transformer model typically includes a positional encoding module that preserves information from the ordering of input sequence, and a set of highly parallel Multi-Head Attention layers that encourage the flow of information among different parts of the data. These Multi-Head Attention layers are powerful and computationally expensive. They form the building block (a.k.a. Transformer block) of Transformer models. There are typically multiple repeated Transformer blocks in a model. Such mechanisms allow the representation of contextual relationships between words across longer distances than previous models and can do so in parallel rather than in a sequential manner.

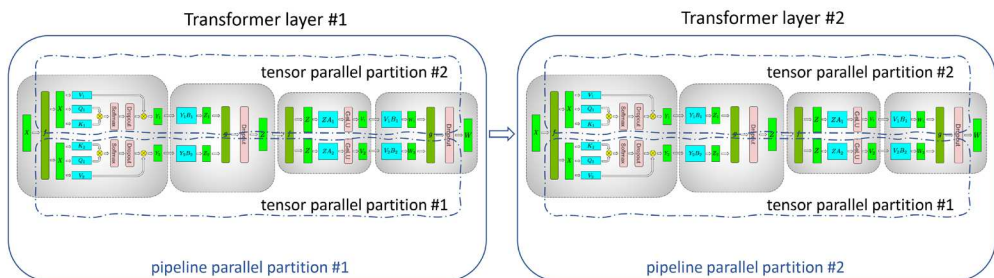


**Figure 3.9** A typical architecture of Transformer models.

Large language models pre-trained on huge corpora already encode much of language structure, so it does not take a large amount of data to fine-tune the model to capture new concepts. A few hundred new examples can be enough to retrain the next version of the model. This enables few-shot learning or even one-shot learning use cases.

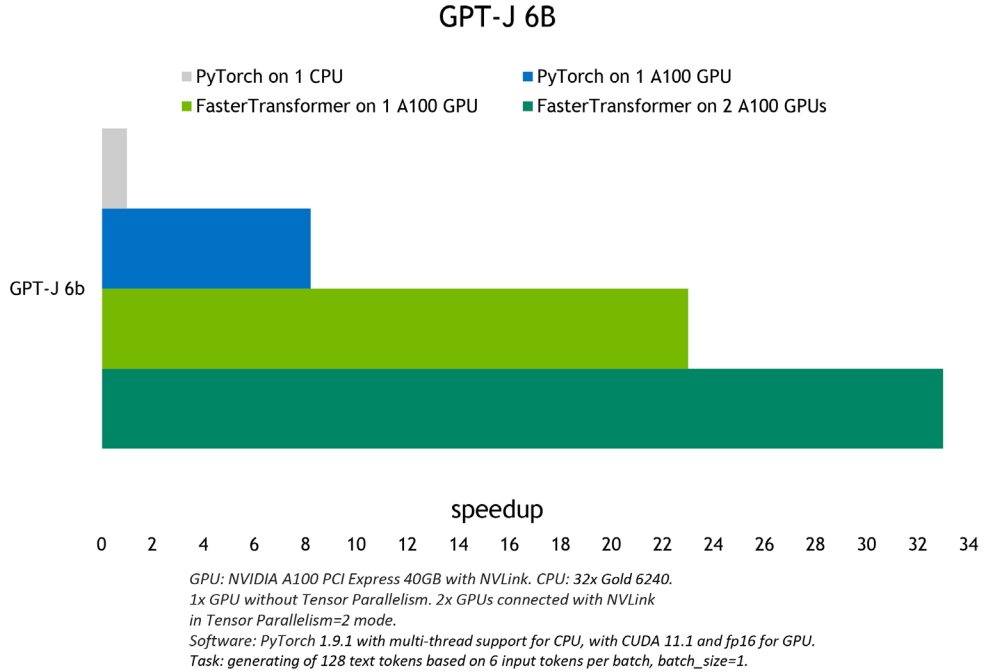
While powerful, large language models tend to be large and compute intensive due to the quadratic complexity when relating each token with every other token. Transformer models have millions or even billions of parameters, which can pose challenges when deploying them for inference in production. Some models may not even fit in the memory of a large GPU, and in such cases multi-GPU and multi-node executions can be needed for inference.

Triton backends like [FasterTransformer](#) can accelerate the inference speed of transformer models and lower the cost. In particular, FasterTransformer contains the implementation of the highly-optimized version of the Transformer block that contains the encoder and decoder parts. It supports the inference of large Transformer models in a distributed manner using multiple GPUs. Figure 3.10 shows a couple of transformer blocks distributed between four GPUs using tensor parallelism (tensor MP partitions) and pipeline parallelism (pipeline MP partitions).



**Figure 3.10** Inference acceleration of Transformer models. (Source)

The speedup with Triton's FasterTransformer backend is significant, achieving 8x to 30x compared with using unoptimized Pytorch models on CPU, as shown in the graph in figure 3.11 ([image source](#)).



**Figure 3.11** Triton provides significant speedup of large language model inference. ([Source.](#))

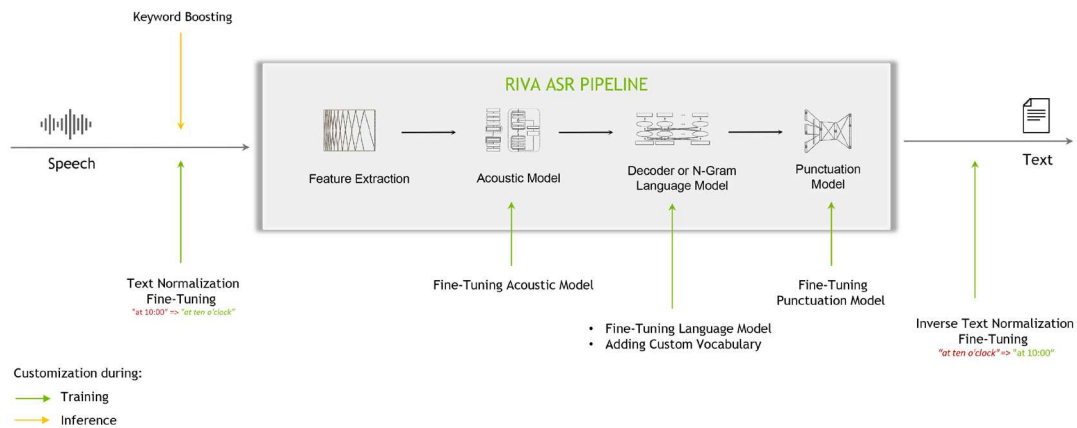
Having a successful deployment of NLP models is not the end, but the beginning, of a journey to realize business value. It is important to continuously evolve the inference through retraining. As new words and concepts appear in the corpus (for example, the phrase COVID-19 did not exist before 2019), the datasets and labels need to be updated, vocabulary and tokenizer refreshed, and the model retrained.

### 3.5.5 Speech recognition

Automatic speech recognition (ASR) is compute-intensive and requires a powerful and flexible platform to power modern conversational AI applications. There are unique challenges for inference on speech data. For example, when running inference on multiple sets of utterances, the inference server must restore the previous state of the components in order to maintain context. To do so, an utterance is represented as a sequence of audio chunks, and each audio chunk from a given sequence is associated with a sequence ID. In addition, inference often needs to be completed in real time,

putting a high standard for latency and throughput. Moreover, multiple models can be involved with a speech recognition application: from speaker counting and identification, to the core speech recognition model.

Figure 3.12 shows a typical pipeline for ASR deployed with Triton, using [Kaldi](#), a popular framework for automatic speech recognition.



**Figure 3.12** An accelerated end-to-end pipeline for automatic speech recognition (ASR).

The raw input contains multiple utterances. They are processed by a Feature Extraction and Sampling module. The inference server receives chunks of audio, each containing an amount of data samples and associated with an ID to indicate it belongs to a certain sequence. Extracted features are sent to an Acoustic Model for classification. Using the likelihoods produced by that classification, and with the help of an HMM Language Model Decoder, you can determine the most likely transcription for that audio.

## 3.6 Recipes for complex inference tasks

In the previous section, we discussed recipes for deploying inference for various data types. They can be readily applied to specialized inference tasks such as computer vision, natural language processing, speech AI, and fraud detection. In this section, we combine the recipes on more complex inference tasks that can involve multiple data types and multiple inference steps.

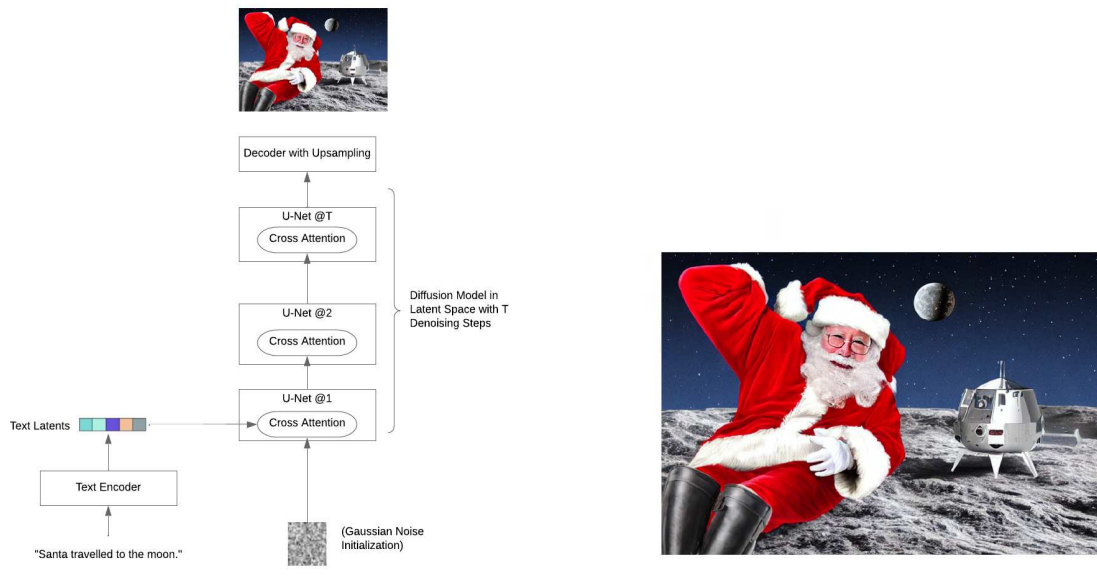
### 3.6.1 Text2Image

From social network feeds, you likely have seen intriguing photos that look realistic but cannot be real. These photos are generated by text2image models like [CLIP](#), [DALLE](#), [IMAGEN](#), and [Stable Diffusion](#) that gained popularity in the last few years. Using a text prompt such as “Santa traveled to the moon”, the model can generate (or “dream up”) a vivid picture with stunning details, even if there is no way to collect historical



data about such images. A large amount of text and image data is used to produce such models.

Figure 3.13 shows the architecture of the Stable Diffusion model that consists of a text encoder that translates text into a vector space, and a diffusion model that translates this vector into a high-resolution image. We made this Santa image using a desktop computer with a RTX 3090 GPU card and the open-source code downloaded from the [stable diffusion paper](#).



**Figure 3.13** Architecture and sample result from Stable Diffusion, an image generation model.

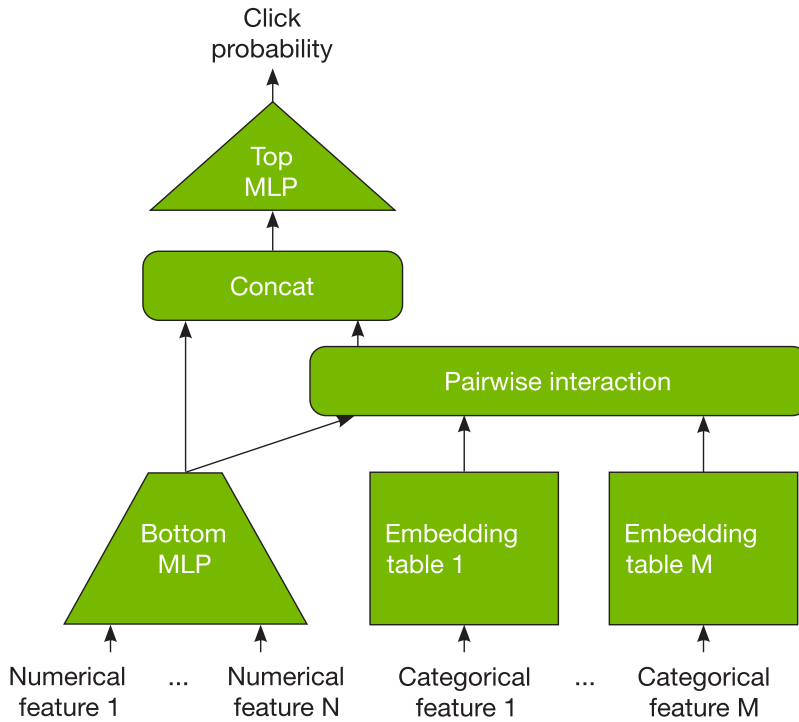
The word big does not do enough justice to the size of these models, because many of them are composed of large language models (as text encoders) as well as large generative models (such as diffusion models), each containing millions or billions of parameters.

Deploying such models in practice can prove to be difficult as the whole model may not fit in a single GPU. Triton utilizes all available GPUs automatically when the server has multiple GPUs, which allows you to focus on machine learning and business logic, rather than managing device memories and handling server crashes.

### 3.6.2 Recommender systems

Recommender systems process data about users and items, and rank items for each user such that the top-ranked items are more interesting to the user, resulting in more engagement, conversions and long-term retention. Depending on the use case, tabular, text, image, and audio data may be used as part of input data.

Though there are many ways to approach this problem, recent advances largely converged on neural network-based approaches where both users and items are represented by floating point vectors (a.k.a. embeddings). By doing so, both users and items are represented as vectors in a high-dimensional space. When a user vector is close to an item vector, the user is more likely to engage with the item. These embedding models for users and items can be trained using the user’s viewing, clicking, and purchasing history of items. When there are more interactions (e.g., clicks) between a user-item pair, the training algorithm nudges their embedding vectors to be closer. A simplified view of an embedding based recommender model is shown in figure 3.14.



**Figure 3.14** A schematic illustration of a recommendation engine model that predicts user clicks.

As you can see in the figure, numerical features and categorical features are extracted from raw data, and additional pairwise interaction features are created to capture their inter-dependencies. This set of features are then concatenated and passed to a feed-forward network (such as a multi-layer perceptron, or MLP) to produce the predicted probability of the user clicking on an item.

Compared to pure NLP models where compute is still the dominant factor in throughput, deep-learning recommenders tend to be heavier in terms of their memory footprint. The embedding tables in modern recommenders can reach multiple

terabytes, often exceeding the capacity of CPU or GPU memory, and involve pure memory lookup operations. The MLP portions remain relatively much smaller in comparison, making deep-learning recommenders memory-bound.

Accessing embeddings often generates a scattered memory access pattern. This can create challenges in memory systems, making them inefficient. NVIDIA GPUs have a highly parallelized memory system that has multiple memory controllers and address translation units, which is great for scattered memory accesses. Additionally, with the latest NVIDIA GPU technology, multi-GPU and multi-node GPU memory capacities are getting sufficiently large for large embeddings.

Traditional recommendation algorithms, such as collaborative filtering, usually ignore the temporal dynamics and the sequence of interactions when trying to model user behavior. However, users' preferences do change over time. Sequential recommendation algorithms can capture sequential patterns in the users' browsing that might help predict the users' interests for better recommendation. For example, users who are starting a new hobby such as cooking or cycling might explore products for beginners and may move to more advanced products over time. They may also completely move on to another hobby of interest. Therefore, recommending items related to their past preferences would become irrelevant.

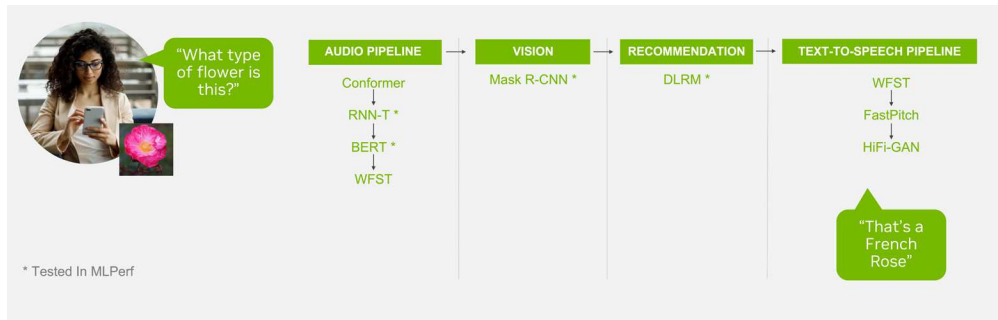
A special case of sequential recommendation is the session-based recommendation task where you only have access to the short sequence of interactions within the current session. This is very common for online services like e-commerce, news, and media portals where the user might be brand new or prefers to browse anonymously, and no cookies are collected as a result of GDPR compliance. This task is also relevant for scenarios where the user's interests change a lot over time depending on the user's context or intent, so leveraging the current session interactions is more promising than old interactions to provide relevant recommendations.

Transformer architectures can provide more accurate recommendation for sequential and session-based recommendation. The [Transformers4Rec](#) library, for example, makes developing transformer-based recommender systems much easier.

### 3.6.3 **Conversational AI**

Conversational AI is another example of real-world applications where multiple models are needed to make it work end to end. As shown in figure 3.15, the user may take a photo of a flower and ask a question to her smartphone using speech: "what type of flower is this?" After a second, the phone answers: "That's a French Rose." During this blink of time, three types of data: image, speech audio, and text were passed back and forth between at least four machine-learning models and inference services. First, the speech audio was processed by the audio pipeline and converted to text. Next, the photo of the flower was processed by a computer vision model to classify its category. Then a recommender system ranks the potential answers and picks one that best suits

the user's needs. Finally, the answer was converted to a natural voice using a text-to-speech pipeline.



**Figure 3.15** Real-world conversational AI applications use many models.

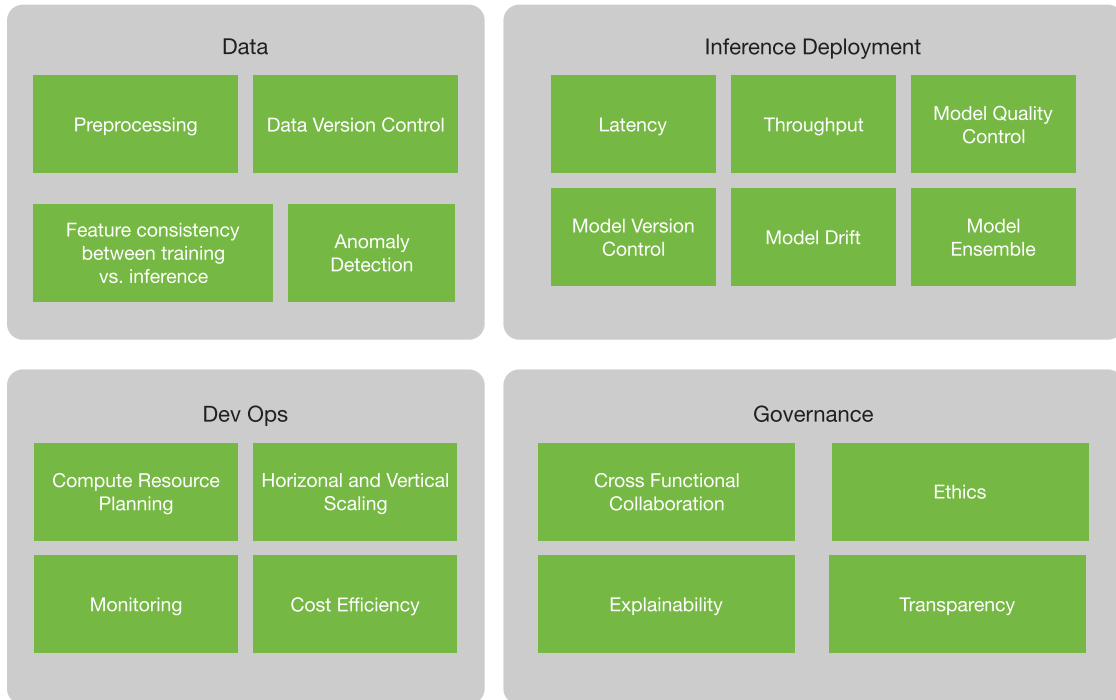
Each type of model has a different requirement for compute (CPUs and GPUs). To ensure a smooth user experience and a manageable total cost of ownership for the inference services, the throughput of the whole system needs to be optimized. As the throughput of the whole system depends on its bottleneck, a.k.a. the slowest part, more compute resources often need to be allocated to the inference model that has the largest number of parameters. This means allocating larger GPUs, CPUs, and more virtual machines to the bottleneck inference service. Autoscaling is often used to dynamically adjust the allocation of compute resources based on usage, to scale up when more users appear and scale down when the service is mostly idle.

### 3.7 *Deployment process and best practices*

The complexity of deploying inference, mentioned in the beginning of this chapter, is in the environment where inference operates. It is one thing to deploy inference in a controlled and isolated environment, and it is another thing to deploy inference in a production environment where many factors come into play. These factors include:

- Upstream data that feed into inference
- Downstream applications that consume
- Compute resources that carry out inference
- Rollout of new inference models
- Stakeholders that are responsible for the deployment, monitoring and maintenance of the inference system

It takes a village to handle all these factors. Cross-functional collaboration is needed among teams responsible for data, machine learning, DevOps, and business stakeholders. To give you an idea of how this collaboration might work, we've mapped out the inference deployment tasks and the teams that usually handle them in figure 3.16.



**Figure 3.16** The success of inference deployment in a production environment depends on multiple factors outside the inference system itself. The number and complexity of these factors require cross-collaboration between many teams.

In this section, we will discuss different aspects of the deployment process and best practices for deploying inference.

### 3.7.1 Managing changes in upstream data

Data is dynamic, just as the business events happening day to day. If input data to inference changes, it is important to proactively anticipate such changes and make sure that the inference system can handle them.

Here are some ways the upstream data may change:

- New trends in the data can emerge. For example, the COVID-19 pandemic has caused sea changes in how people live and work, and as a result the data collected reflects this.
- The user group of the data can change. Expanding a product from California to Florida, for example, may mean a shift in the distribution of certain features.
- New feature types can appear as the product or service evolves. For example, a new type of user conversion may become available.

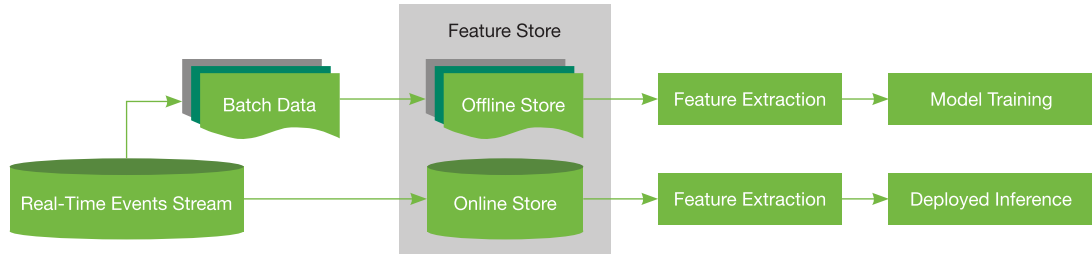
- An existing feature may become unavailable or deprecated.
- Adversarial actors such as fraudsters may actively seek out opportunities to use bots to influence ranking positions or fake clicks, causing pollution of data.
- Underlying technical systems may change. For example, during daylight saving time, certain software packages may not cope with the time change and can yield unexpected effects.
- Delays and interruptions in the data may occur, due to server disruptions or other unexpected technical disruptions. This can cause missing data or problematic aggregated data.

To cope with these challenges, being aware of the possible complications is a good first step. For critical inference deployments, dedicated engineering resources should be allocated, and tools put in place to monitor upstream data, perform quality control, and alert the stakeholders if any changes are detected.

As mentioned in [Breck et. al \(2017\)](#), “it can be difficult to effectively monitor the internal behavior of a learned model for correctness, but the input data should be more transparent. Consequently, analyzing, and comparing data sets is the first line of defense for detecting problems where the world is changing in ways that can confuse an ML system.” A number of tests were proposed in this paper regarding readiness of production machine-learning systems. Most relevant to this section are two examples of them:

- Training and inference features compute the same values.
- Changes in dependences result in a notification.

Feature stores are MLOps tools that can alleviate the pain points in managing changes in upstream data. It combines data from various data sources and turns it into a single source of truth for features. As illustrated in figure 3.17, a Feature Store often has Offline and Online stores which maintain consistency between how features are generated. The Offline store is consumed by Model Training and the Online store is consumed by Deployed Inference. As an example, [Feast](#) is an open-source feature store that connects upstream data sources for streaming data (e.g., Kafka and Kinesis) as well as batch data (e.g., BigQuery, S3, Snowflake). The workflow illustrated in figure 3.17 underscores the importance of maintaining separate Online and Offline feature stores in a deployed inference context. The consistency of data preparation procedures between offline model training and online production inference reduces the risk of faulty deployment of ML inference.



**Figure 3.17** A typical workflow for using Feature Stores to manage inference where the gap between offline and online data is reconciled.

### 3.7.2 Managing integration with downstream applications

Predictions from inference models are rarely the end of the story. They are often used to drive the behavior of downstream applications that more directly interface with users. In a real-time ad-bidding system, for example, a CTR (click-through rate) prediction model is typically built to predict the chance that a particular ad will be clicked by a user. The predicted CTR is then used as an input to the pricing policy, which decides the final bid price based on the campaign budget, target CPM (cost per mile), win rate, competitive landscape, pacing requirements, as well as the predicted CTR.

The scenario of inference integration varies greatly depending on the use case, product, and industry. Close collaboration between business operations, product owners, data scientists, machine-learning engineers, software engineers, and DevOps is often necessary to release the full potential of the inference system.

### 3.7.3 Managing compute resources and trade-offs in speed, reliability, and cost

Typically, the more CPUs are available on the server, the faster the inference runs, with higher throughput and lower latency. ML models can take advantage of parallel threads to run inference faster. If latency is the main concern, one may use large virtual machines with many CPUs or with a GPU.

Multi-model serving can achieve low latency at low cost by dividing the costs across many models. Peak load for different models occurs at different times. As a result, multi-model inference incurs a significantly lower cost without sacrificing latency. Large language models and computer vision models can require a lot of memory to run. On CPUs, the inference can take seconds or even minutes to run. On GPUs, the inference is much faster, but GPU memory is scarcer than CPU memory. When a single GPU is not enough to hold a very large model, multi-GPU, and multi-node inference is necessary. NVIDIA Triton uses two model parallelism techniques:

- *Pipeline (Inter-Layer)*: Parallelism that splits contiguous sets of layers across multiple GPUs. This maximizes GPU utilization in a single node.
- *Tensor (Intra-Layer)*: Parallelism that splits individual layers across multiple GPUs. This minimizes latency in single-node scenarios.

Out-of-memory (OOM) errors can occur during inference even after the model can be loaded and executed successfully to predict a number of examples. This is typically due to memory leak issues, where graph operations and additional overhead are created and accumulated after each prediction. Monitoring memory usage over time can help identify the issue.

Autoscaling is a great way to horizontally scale to tens or hundreds of machines, capable of handling large volumes of data. The machines are stopped and started as needed, and the cost saving is significant. However, there can be a cold-start problem. As queries ramp up, while the machines are being started, a number of queries can wait in line for a while or even fail. By increasing the minimum number of inference nodes (sometimes called warm nodes), such cold-start problems can be relieved, and the system can handle the queries with lower latency. It is a trade-off between cost and latency.

Due to the size of some models, loading and initialization can have significant overhead as well. Components of the model may also be lazily initialized. These factors cause high latency in the first inference, which can be several orders of magnitude higher than that of a typical inference request. By sending a sample of inference requests to warm up the system, the latency can be reduced.

### 3.7.4 **Rollout of new inference models**

Imagine a new inference model is trained, and offline evaluation shows promise of improvement. The model is deployed and ready to serve prediction requests. Rolling out the new inference is a source of excitement and anxiety at the same time. What if the new inference model is less accurate? How do you find out? Can the new inference deployment handle production traffic? What if an unexpected crash happens? Here are a few tools to make the rollout of new inference models smoother.

#### **SHADOW MODE**

“Shadow mode” refers to the process of deploying a new inference model, where both the new model and the current in-production model are used to calculate predictions, but the predictions of the new model are not used by the production system to affect user experience. Predictions from the new model are typically saved for further analysis, such as side-by-side comparison with the current model. Is the new model more accurate than the current model on live production data? Does it run slower? Does it require more compute and memory? Is it robust to fluctuations in the input data? These questions can be answered during shadow mode deployment.

#### **A/B TESTS AND MULTI-ARMED BANDITS**

Offline evaluation can demonstrate good model performance on historical data, but it cannot establish causal relationships between a new model and better user outcomes.



A/B testing is a common way to compare two or more models on a fair ground, by performing a randomized controlled trial. Sufficient time and data is needed to establish an adequate statistical power (probability of an improvement successfully detected) and significance (also known as p-value, probability of getting a false positive). Peeking and early stopping are common pitfalls which can result in a non-improvement being falsely identified as an improvement. There is also a risk of lost business outcome during the course of the test, if the new model turns out to be significantly worse than the current one.

Bayesian A/B tests can be used to provide a more intuitive interpretation of the test and robustness against peeking. Instead of p-values, you get direct probabilities on whether the new model is better than the current one and by how much.

Multi-armed bandits (MAB) learn from data gathered during a test while dynamically increasing the allocation in favor of better-performing variations. This optimizes business outcomes while performing a test.

### 3.7.5 Metrics for monitoring the inference system

Here we describe typical metrics for monitoring the inference system.

#### STANDARD MACHINE-LEARNING METRICS

Machine-learning metrics are useful for data scientists and machine learning engineers to keep track of the quality of the inference model and diagnose problems early. The metrics can be specific to the type of model being used. For example, computer vision models use a different set of metrics than natural language models.

- *Accuracy*: The fraction of examples that are correctly classified. This is useful for binary, multi-class, and multi-label classification models.
- *Mean average precision (mAP)*: This is the area under the curve for the precision recall curve. It is useful for information retrieval models, object detection models, and classification models. For object detection models, an IoU (bounding box overlap) threshold is usually chosen beforehand. It is sensitive to label imbalance in the data.
- *AUROC*: This is the area under the ROC curve. It is useful for classification models. It is not sensitive to label imbalance in the data.
- *NDCG*: This is the normalized discounted cumulative gain. It is useful for ranking models.

#### SERVICE-LEVEL METRICS

Service-level metrics are useful for machine learning engineers and DevOps engineers to monitor the quality of the deployment.

- *Latency*: The time it takes for a prediction to be made. Percentile statistics are often instrumental to collect and display.
- *Throughput*: The number of predictions per second.

- *CPU utilization*: The percentage of CPU time used by the inference system.
- *GPU utilization*: The percentage of GPU time used by the inference system.
- *Memory utilization*: The percentage of memory used by the inference system.
- *Disk utilization*: The percentage of disk used by the inference system.
- *Number of nodes*: The number of VMs, containers or pods being used by the inference system.

### **BUSINESS METRICS**

Business metrics is used to communicate the health of an inference system to executives, product management, and other business stakeholders. There can be thousands of business metrics specific to use cases, verticals, markets, and product lines. Some of the most common ones are:

- *CTR*: Click-through rate. The probability that a user clicks on a document, a product, or an ad.
- *Conversion rate*: The probability that a user adds a product to cart, purchases a product, or subscribes to a service.
- *CPC*: Cost per click. Mainly used in advertising use cases.
- *Hours saved*: The number of manual hours saved by using the inference system to automate business processes.

### **3.7.6 Teamwork and stakeholder involvement**

Ensuring and maintaining the quality of the inference system requires teamwork. The typical roles directly responsible for the inference system are:

- *DevOps and ML-ops*: They are responsible for provisioning sufficient compute resources to run inference. Production issues such as latency spike, throughput dip, connectivity disruptions, and server crashes are best addressed by this role. Out-of-memory errors are trickier, because this may be a resource provisioning problem, or it could be a memory leak due to the implementation of the inference model. The turn-around time is typically short.
- *Machine learning engineers*: They are responsible for optimizing and deploying inference models. It is a continuous improvement process to keep accuracy, latency, throughput, and cost in check, while new models are trained and deployed. Drifts and anomalies in model predictions are monitored and actions taken to mitigate them. The turnaround time can be short to medium.
- *Data engineers*: They are responsible for quality and preprocessing of the upstream data feeding into inference systems. The production data should match that of offline data used in model training. Drifts, anomalies, and missing data are best addressed by this role. The turn-around time can be short to medium.

- *Data scientists:* They are responsible for research and prototyping new models as candidates to deploy to production. New models that show promise in offline evaluation may not always translate to real gain when deployed to production. Results from production inference results should be analyzed, so the model training algorithms can be adapted and improved. The turn-around time is typically long.

The roles described here are not a rigid and static view of the team supporting inference systems. In smaller companies, a person may hold multiple roles, while in larger companies, there can be a multi-person team for each role.

### 3.7.7 Enterprise support for AI inference deployment

As AI initiatives move into the production stage, the need for a trusted, scalable support model for enterprises becomes vital for ensuring AI projects stay on track. [NVIDIA AI Enterprise](#), designed for enterprise-grade AI development and deployment, is an end-to-end, secure, cloud-native suite of AI software, enabling organizations to solve new challenges while increasing operational efficiency. It is available across bare metal, virtual, container, and cloud environments, reducing the time to move from pilot to production of AI solutions.

Available in the cloud, the data center, and at the edge, NVIDIA AI Enterprise offers key features to ensure business continuity: global NVIDIA Enterprise Support for NVIDIA Triton and TensorRT, guaranteed response times, priority security notifications, API stability, coordinated support across the full solution and partner products until resolution, control upgrade and maintenance schedules with long-term support (LTS) options, and access to NVIDIA AI experts.

A global financial services company selected NVIDIA AI Enterprise to support its AI initiatives by leveraging the curated AI stack, including NVIDIA Triton Inference Server, running on certified infrastructure to ensure performance advantage, resulting in gaining 20x performance on AI inference that outperformed its existing home-grown software.

## 3.8 Code lab: Deploy inference for reverse image search

Now for the fun part! Now that you've absorbed all this information, you can get some hands-on experience coding end-to-end model training and deployment in our customized code lab. Your project in this code lab is to build a reverse image search model server.

The lab is located at [https://github.com/kungfuai/triton-inference-examples/blob/main/reverse\\_image\\_search/](https://github.com/kungfuai/triton-inference-examples/blob/main/reverse_image_search/).

In this code lab, you will download image data from a public dataset, train a deep learning model or download a pre-trained model, and deploy it with Triton Inference Server. You will also learn how to use `perf_analyzer` to profile the latency and throughput of the inference server. Enjoy!

# *The AI inference horizon*

---



AI has become an invaluable tool for modern businesses, and we are currently on an upward trajectory that will continue to uncover novel and diverse applications. The factors that influence the current state of growth (e.g., education, hardware capabilities, and others) have their own dynamics that affect the broader state of the AI inference industry. Will the growing pool of engineers and scientists keep up with industrial demand? Will our algorithms demand more than modern compute hardware has to offer? Will new regulations fundamentally shift the dynamics of business in AI?

While the possibilities of the future remain to be seen, we have some hints at what may lie ahead. Four key areas strongly influence the growth trajectory of the AI industry: broad AI adoption, algorithms, hardware, and regulatory environments. We will look at each independently.

## **4.1 Broad AI adoption**

Each year, IBM (in partnership with Morning Consult) releases the Global AI Adoption Index. In their [2022 report](#), it was revealed that in the past year global AI adoption has increased by 4% to 35%. On top of that, 42% of companies are currently exploring AI and what benefits it may provide for their businesses. AI can afford automation and skills to directly address labor shortages, promote sustainability when human involvement is either too expensive or inaccessible, and provide a multitude of other benefits. “Two-thirds (66%) of companies are either currently executing or planning to apply AI to address their sustainability goals,” said the report. Statements such as these are on the rise.

“AI-curious” companies currently face some barriers that may erode over time as well. For example, up to now, many of these transitioning firms have struggled to shift prototype applications to production deployments. MLOps tooling that manages machine learning artifacts in production environments has become available and has matured to address this gap. Products like MLFlow, Weights & Biases, and Neptune.ai all help alleviate these production burdens, thus slightly lowering the barrier.

Additionally, adoption for some has been out of reach because of both price and skill-set. On the price front, some components of the machine-learning lifecycle are starting to become commoditized, which can make adoption more affordable. Though not all aspects of the lifecycle can follow this path, simplifying a few steps can make enough of a difference that business value can be realized. Increased competition also plays a role in controlling price. [The 2022 AI Index Report from Stanford](#) stated that “Since 2018, the cost to train an image classification system has decreased by 63.6%, while training times have improved by 94.4%. The trend of lower training cost but faster training time appears across other MLPerf task categories such as recommendation, object detection, and language processing, and favors the more widespread commercial adoption of AI technologies.” At minimum, the price to adopt is on a downward trajectory, and if more businesses can afford to train their own models, so too can they enable inference deployments.

The Stanford report also mentions some statistics regarding the talent pipeline emerging from universities, which, if focused on AI/ML, will enable further adoption in industry. For instance, in the decade from 2010 to 2020, the number of new computer science (CS) undergraduate graduates at doctoral institutions in North America grew by a factor of 3.5x and shows a consistent year-over-year growth trajectory. In addition, 21% of new PhD candidates in 2020 specialized in AI and machine learning, which is around 3x larger than the runner-up specialty (software engineering). Finally, over the same 2010-2020 decade mentioned previously, the fraction of new PhD graduates going to industry grew from around 45% to over 60% (at the expense of post-doctoral academic careers). The industrial demand and technical appeal of the artificial intelligence sector is garnering more interest for AI in the academic world, which should help alleviate some of the talent pool issues going forward.

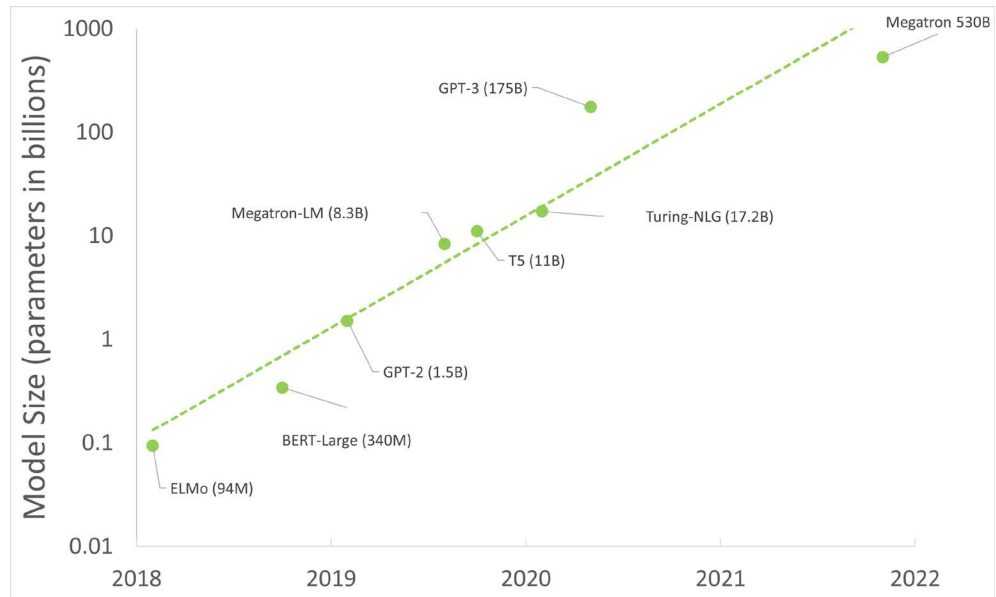
In summary, artificial intelligence adoption is on a sharp upward trajectory. Optimally deployed inference lies at the heart of most industrial applications, and therefore we should expect healthy market activity around the topic in the coming years.

## 4.2 Algorithms

A number of algorithmic developments are also lowering the barrier to deploying optimal prediction pipelines. There exists a constant tug between the algorithms that demand more computational power and those that seek to make the inference process more efficient. So far, the former seem to be outpacing the latter, which means that we continue to demand more and more performance from the hardware that our algorithms rely on.

Regarding the more demanding models, natural language applications, in particular, seem to be growing in complexity at an incredible rate. In a fairly extreme case, the

Megatron-Turing NLG (MT-NLG) model, released in late 2021, demands computation from 280 billion parameters. These models are continuing to grow in size with each release, and don't show signs of a plateau yet. In addition, models like OpenAI's CLIP, which learns to generate images using natural language data, are validating multi-domain approaches that will also add to model complexity over time. Figure 4.1 illustrates the accelerated growth of language models over the last several years (note the logarithmic scale on the parameter count axis).



**Figure 4.1** Graph illustrating large language model (“LLM”) growth over time. [Source](#)

By the same token, however, sparsely activated networks (i.e., networks that conditionally avoid computation on input samples) are becoming more popular too. Google’s Pathways architecture is intended to do just that: avoid unnecessary computation that only marginally contributes to prediction quality. Mixture of experts (MoE) models attempt to accomplish the same goal and have demonstrated success in production applications, as mentioned in the Microsoft Translator case study in chapter 2.

Ongoing algorithmic developments will continually demand more from the hardware we perform computations on, and therefore deployment efficiency remains paramount now more than ever.

### 4.3 Regulatory environments

Going back to the Stanford report, the number of bills passed into law that contain mentions of “artificial intelligence” grew from just 1 in 2016 to 18 in 2021. Clearly, governmental interest in artificial intelligence as a whole is increasing. However, though restrictive, regulatory compliance can also create business opportunities.

Tools around fairness, model explainability, and bias mitigation especially will become more and more important as discrimination inherent in deployed models begins to affect the general populous more deeply. Credit determinations, housing decisions, and other choices that underpin the lives of most individuals will be looked at under a magnifying glass. As a rule, the larger the impact, the larger the level of scrutiny we can expect going forward.

Though trustworthy AI, fairness, and bias mitigation is a nascent field, assessing the risks will become very important for companies that build products and services affecting individuals. This underpins the very nature of how we perform inference, and especially how we monitor inference once it begins to touch people in their daily lives.

#### **4.4 Additional trends**

In our personal experience as AI consultants, we are seeing a rise in demand for AI business strategy consulting services. Businesses want to know that the problem they are trying to solve will generate value once put into play. Many companies have completed perfectly well-executed engagements, only to find that the eventual delivery was devoid of any true value creation. The importance of strategy work that ensures the most “bang for buck” is becoming incredibly important, and organizations are finding that a lot can be accomplished with relatively little. It’s often the case, for example, that partial automation with human supervision results in higher quality outcomes, as opposed to full automation with untraceable mistakes. Time will tell who has succeeded to deploy the *right* inference solution in the first place, let alone properly executed ones.

There also seems to be a push toward more standardization. In the same way that containers revolutionized software deployments, model format and inference deployment standards (such as ONNX and NVIDIA Triton, respectively) are allowing engineers across a wide variety of industries to speak the same language. This will have a positive effect on the pace of innovation.

Finally, the rise of an open-source culture around AI/ML is having a massive impact on the state of machine learning model deployment in production. Small start-ups are finding access to the work of a huge network of talented engineers, which enables them to create high quality products in record time. Oft-utilized open-source libraries (such as PyTorch, Scikit-Learn, and Tensorflow, to name a few) are able to respond quickly to bug fixes due to their respectively huge communities of users and developers, as well as feature requests. The amount of activity surrounding the space is at a record high and shows no signs of slowing down. Because of this, it is all but imperative that businesses make sound decisions around inference tooling.

#### **4.5 Summary**

AI will continue to be integrated into aspects of our everyday lives that we hadn’t considered before, and the infrastructure underpinning these novel systems will mature in kind. Only time will tell what the future may hold, but only by remaining educated can we keep a pulse on the fascinating and powerful world surrounding artificial intelligence inference.