Deploying Machine Learning Models in Production

In the fourth course of Machine Learning Engineering for Production Specialization, you will learn how to deploy ML models and make them available to end-users. You will build scalable and reliable hardware infrastructure to deliver inference requests both in real-time and batch depending on the use case. You will also implement workflow automation and progressive delivery that complies with current MLOps practices to keep your production system running. Additionally, you will continuously monitor your system to detect model decay, remediate performance drops, and avoid system failures so it can continuously operate at all times.

Understanding machine learning and deep learning concepts is essential, but if you're looking to build an effective AI career, you need production engineering capabilities as well. Machine learning engineering for production combines the foundational concepts of machine learning with the functional expertise of modern software development and engineering roles to help you develop production-ready skills.

Week 2: Model Serving Patterns and Infrastructure

Contents

Model Serving Architecture

ML Infrastructure

On Prem

- Train and deploy on your own hardware infrastructure
- Manually procure hardware GPUs, CPUs etc
- Profitable for large companies running ML projects for longer time

On Cloud

- Train and deploy on cloud choosing from several service providers
  - Amazon Web Services, Google Cloud Platform, Microsoft Azure, etc

- We're going to look at models serving including patterns and infrastructure, as well as a little bit of scaling.
- Let's start with a very high level look at the two main places that you can run your infrastructure, on which a machine learning models are going to be built and used.
- This in general boils down to those two main places. You can either have all of the hardware and software required on your own premises.
- Or you can outsource them to a cloud provider who will provide all the hardware management for you, as well as software services such as pipeline management as part of their offering.
- When you run on your own premises, you have complete control over your hardware infrastructure.
- Obviously this can be very flexible and you can be nimble to adapt to changes in your requirements quickly, but it does come at a cost.
- You have to procure, install, configure, and maintain your hardware infrastructure, which can be complicated and expensive.
- **Larger companies** tend to take this approach because they can leverage economies of scale, such as having extensive in-house experience.
- The other option and one often used by **smaller companies** is to outsource their infrastructure needs to experts in building, managing scaling and monitoring hardware, software and applications.
- Great examples of this are the services from the likes of Amazon, Google Cloud Platform, and Microsoft Azure.

Model Serving

On Prem

- Can use open source, pre-built servers
  - TF-Serving, KF-Serving, NVidia and more...

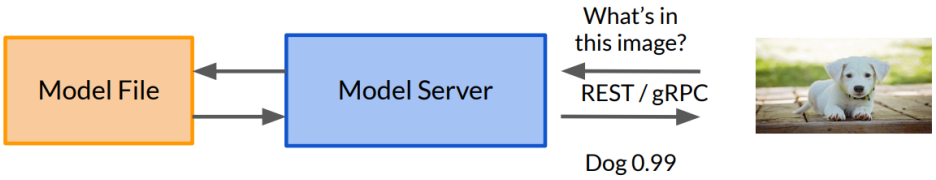
On Cloud

- Create VMs and use open source pre-built servers
- Use the provided ML workflow

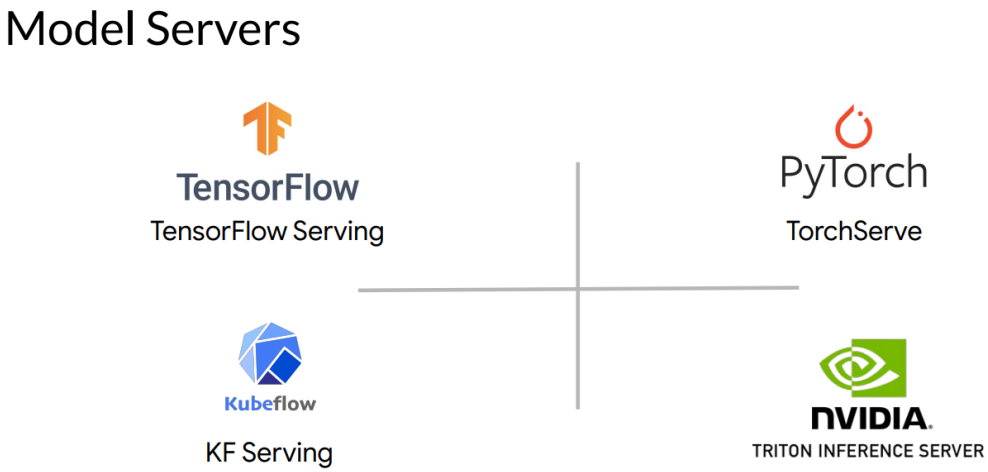
- Going beyond choosing the infrastructure, you also have to consider how model serving will work, depending on the choice you made.
- When using on-premises hardware, you have the flexibility to choose which model server you like, and you're responsible for installing, configuring and maintaining it.
- So options like TensorFlow serving, Kubeflow, NVidia server, and more are going to be made available to you. And we'll explore what some of these look like in just a moment.
- If you do the model serving on the cloud, then you can create **virtual machines** on their infrastructure that give you the same flexibility as on-premise, so you can install and configure the same types of pre-built server, such as TensorFlow serving and Kubeflow and all of those.
- Or you can use a provided suite of tools and services that are available from that cloud vendor.
- For example, if you had chosen the Google Cloud Platform, you could use things like AutoML or any of the Google Cloud AI services. If you've chosen Amazon Web Services, then SageMaker Autopilot would be available to you.
- Now that you've explored the higher level options around how you can do model serving, be it on premise or using a cloud provider, let's now look into model servers and explore some of the popular options that are available to you.
- These include TensorFlow, Kubeflow, PyTorch, and NVidia servers, but before looking at them, let's have a quick recap at **what a model server actually does**.

Model Servers

- Simplify the task of deploying machine learning models at scale.
- Can handle scaling, performance, some model lifecycle management etc.,

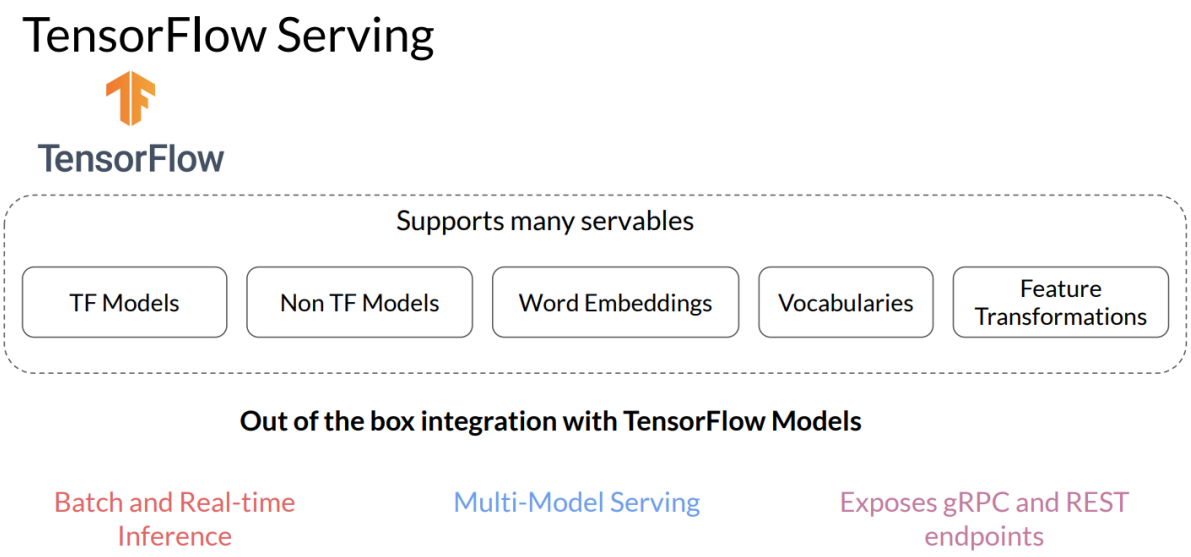


- The high level architecture of a model server can be summarized in a diagram like this one.
- Your model is typically saved to the file system. You could also have multiple versions of the same model, so you can try out different ones
- Ultimately, the model is available to be read by the model server whose job is to **instantiate the model** and **expose** the methods on the model that you want to make available to your clients.
- For example, if the model is an image classifier, the model itself will take in tensors of a particular size and shape.
- For mobile net, these would be 224 by 224 by 3.
- The model server receives this data, formats it into the required shape, passes it to the model file, and gets the inference back.
- It can also manage multiple model versions should you want to do things like AB testing or have different users with different versions of the model.
- And the model server then exposes that API to the clients as we previously mentioned.
- So in this case for example, it has a REST or RPC interface that allows an image to be passed to the model.
- The model server will handle that and get the inference back from the model, which in this case is an image classification.



- Some of the most popular model servers are listed in the slide: TensorFlow serving, Torch Serve, Kubeflow Serving, and the NVIDIA Triton Inference Server.
- For example, the Triton Inference Server allows deployment of models from any framework, from local storage, Google Cloud Platform, or AWS S3.
- You will look briefly at the architecture of each of these next.

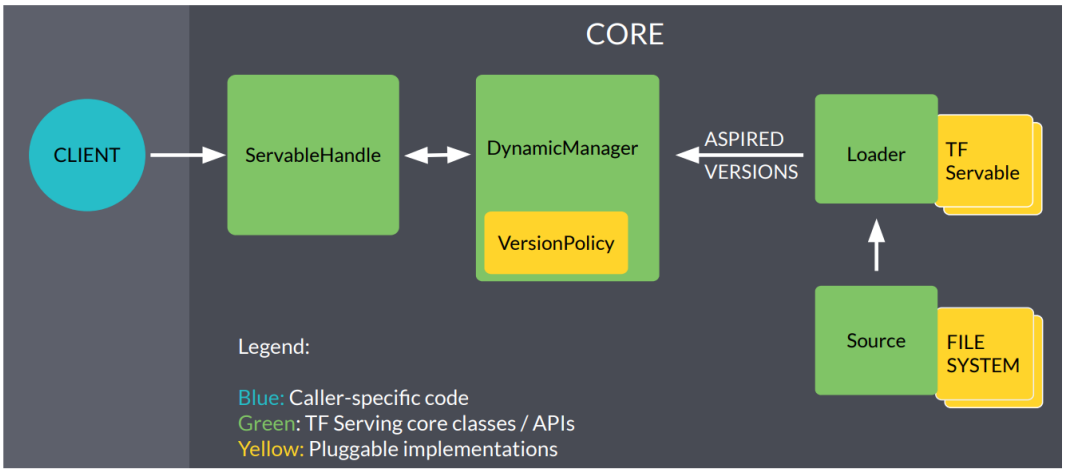
Model Servers: Tensorflow Serving



- So now that you've seen model serving from a high level, let's dig into some of the architecture is a little more deeply, starting with TensorFlow Serving.
- In week 1, you had a chance to experiment with the basics of TF serving, which is a flexible, high performance serving system for machine learning models.
- It provides out of the box integration with Tensorflow models, and it can also be extended to serve other types of model.
- So let's take a look at the architecture of this serving system and some important features are that it provides both batch and real time inference.

- So you can either get a bunch of inferences at the same time (which is useful if you're building something like a recommendation engine that requires a lot of predictions) or real time inference if you want to answer to a single task back quickly (useful for, for example, image classification)
- Multi model serving is also available and this allows you to have multiple models for the same task and the server chooses between them.
- This could be useful for example, for A/B testing, audience segmentation and more.
- It also provides remote procedure calls or traditional rest endpoints on which you can call your server.

TensorFlow Serving Architecture



- The high level architecture for Tensorflow serving looks like this.
- It's built around the core idea of a **Servable**, which is the central abstraction in TF serving.
- These are the underlying **objects** that clients used to perform computation, for example, inference or lookups.
- They can be any kind of type or interface, and this makes them very flexible.
- A typical Servable is a Tensorflow saved model but it could also be something like a lookup table for embedding.
- The Loader manages a Servable's lifecycle.
- The Loader API enables common infrastructure independent from specific learning algorithms, data or whatever product use cases were involved.
- Specifically, Loaders standardizes the API for loading and unloading a Servable.
- Together these produce **aspired versions,** and these represent the set of Servable versions that should be loaded and ready.
- Sources communicate this set of servable versions for a single servable stream at a time
- When a source gives a new list of aspired versions to the Manager, it supersedes the previous list for that Servable stream.
- The Manager unloads any previously loaded versions that no longer appear in the list.
- The Manager then handles the full life cycle of the Servables, including loading the Servables, serving the Servables, and of course unloading the Servables.
- The Managers will listen to the sources and will track all of the versions according to a version policy
- The ServableHandle provides the exterior interface to the client.
- There's a lot of pieces here, so let's look at an example of how this would work.
- Let's take this as an example, say a source represents a Tensorflow graph with frequently updated model weights.
- The weights are stored in a file and disk.

- The source detects a new version of the model weights, and it creates a Loader that contains a pointer to the model data on disk.
- The source notifies the DynamicManager of the aspired version. The DynamicManager applies the version policy and decides to load the new version.
- The DynamicManager tells the Loader - that is enough memory.
- The loader instantiates the Tensorflow graph as a Servable with these new weights.
- A client requests a handle to the latest version of the model, and the DynamicManager returns a handle to the new version of the Servable.
- You can then run inference using that Servable and this is what you would have seen last week when you built the fashion MNIST model example.

Model Servers: Other Providers

NVIDIA Triton Inference Server

- Simplifies deployment of AI models at scale in production.
- Open source inference serving software
- Deploy trained models from any framework:
  - TensorFlow, TensorRT, PyTorch, ONNX Runtime, or a custom framework
- Models can be stored on:
  - Local storage, AWS S3, GCP, Any CPU-GPU Architecture (cloud, data centre or edge)

HTTP REST or gRPC endpoints are supported.

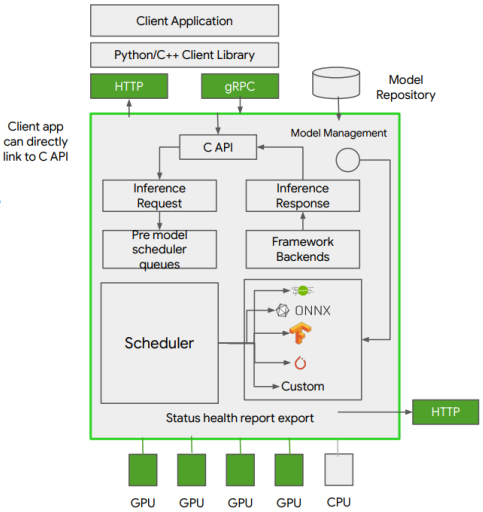
- The Triton Inference Server, an offering from NVIDIA, simplifies deployment of AI models at scale in production.
- It's an **open-source inference serving software** that lets teams deploy trained AI models from **any framework**: TensorFlow, TensorRT, PyTorch, ONNX Runtime, GCP, AWS S3, or even a custom framework.
- You can deploy from local storage or from a Cloud platform, like the Google Cloud Platform or AWS, on any GPU or CPU-based infrastructure, and can be the Cloud, the data center, or even the edge.

Architecture

Triton Inference Server Architecture supports:

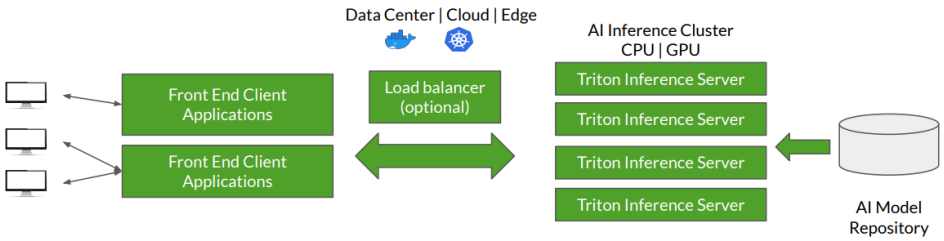
- Single GPU for multiple models from same or different frameworks
- Multi-GPU for same model
  - Can run instances of model on multiple GPUs for increased inference performance.

Supports model ensembles.



- The Triton Inference Server runs multiple models from the same or different frameworks concurrently on a single GPU using CUDA streams.
- In a multi-GPU server, it **automatically creates an instance of each model on each GPU**.
- All of these increase your GPU utilization without any extra coding from the user.
- The Inference server supports low latency real-time inferencing with batch inferencing to maximize GPU and CPU utilization.
- It also has built-in support for streaming inputs if you want to do streaming inference.
- Users can use shared memory support for higher performance.
- Inputs and outputs need to be passed to and from Triton's Inference Server can be stored in the systems or the CUDA shared memory.
- This can reduce the HTTP C or gRPC overhead and increase overall performance.
- It also supports model ensemble.

Designed for Scalability



Can integrate with KubeFlow pipelines for end to end AI workflow

- Triton Inference Server integrates with Kubernetes for orchestration, metrics, and auto scaling.
- Triton also integrates with Kubeflow and Kubeflow Pipelines for an end-to-end AI workflow.
- The Triton Inference Server exports Prometheus metrics for monitoring GPU utilization, latency, memory usage, and inference throughput.
- It supports the standard HTTP gRPC interface to connect with other applications like load balancers.
- It can easily scale to any number of servers to handle increasing inference loads for any model.
- The Triton Inference Server can serve tens or hundreds of models through the Model Control API.
- Models can be explicitly loaded and unloaded into and out of the Inference server based on changes made in the model control configuration to fit in a GPU or CPU memory.
- It can be used to serve models and CPU too.
- It supports heterogeneous cluster with both GPUs and CPUs and does help standardized inference across these platforms.
- During peak loads, it can dynamically scale out to any CPU or GPU.

Torch Serve

- Model serving framework for PyTorch models.
- Initiative from AWS and Facebook

PyTorch
TorchServe

Batch and Real-time Inference

Supports REST Endpoints

Default handlers for Image Classification, Object Detection, Image Segmentation, Text Classification

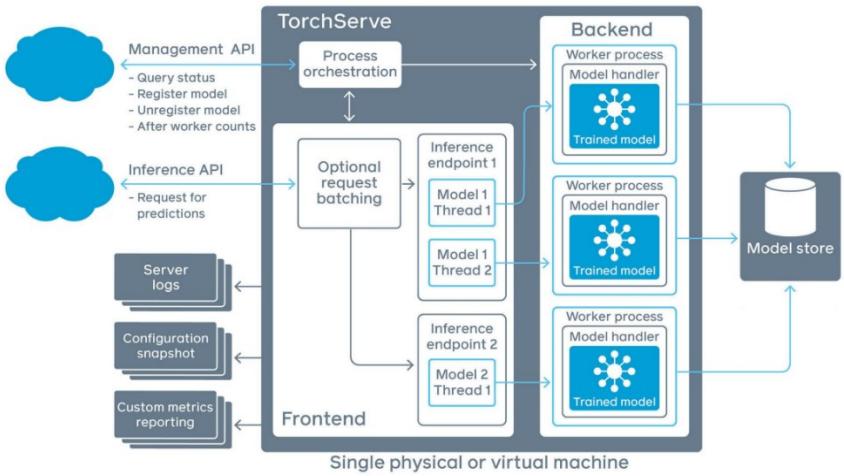
Multi-Model Serving

Monitor Detail Logs and Customized Metrics

A/B Testing

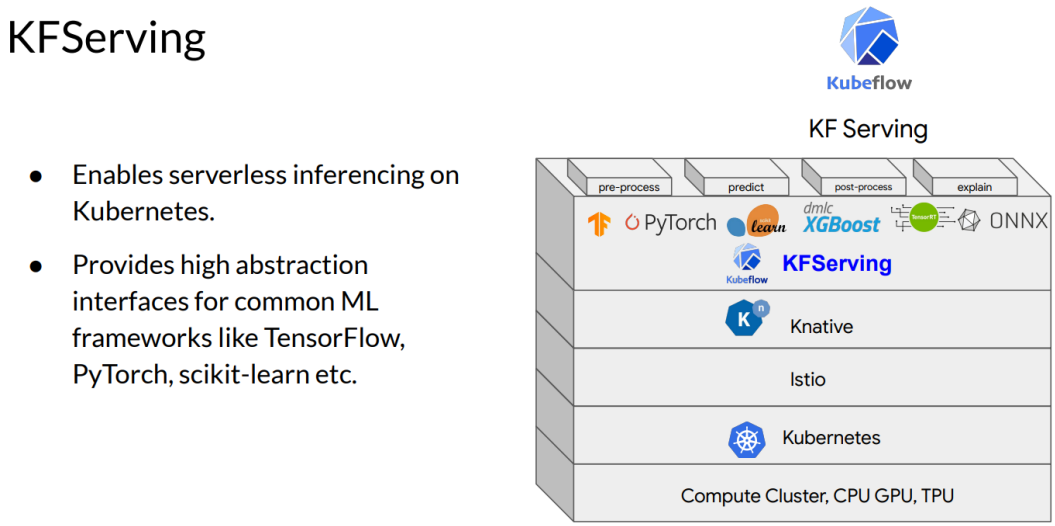
- Another popular serving environment is TorchServe, designed around PyTorch.
- TorchServe is an initiative by AWS and Facebook to build a model serving framework for PyTorch models.
- Before the release of TorchServe, if you wanted to serve PyTorch models, you had to develop your own model serving solutions like custom handlers for your model.
- You had to develop a model server, maybe build your own Docker container. You had to figure out a way to make the model accessible via the network and integrated it with your cluster orchestration system and all of that good stuff.
- With TorchServe, you can deploy PyTorch models in either eager or graph mode.
- You can serve multiple models simultaneously. You can have version production models for A/B testing.
- You can load and unload models dynamically and you can monitor detail logs and customizable metrics.
- Best of all, TorchServe is open source. Hence it's extensible to fit your deployment needs.

TorchServe Architecture



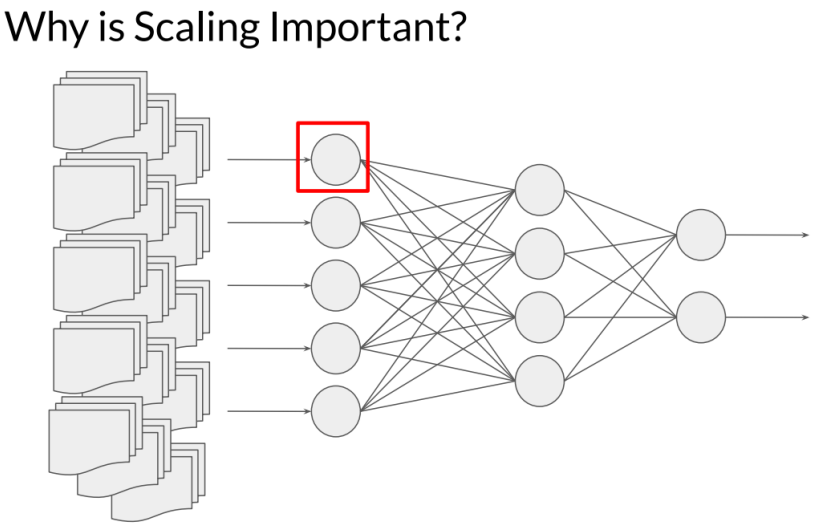
- The server architecture looks like this.
- The front end is responsible for handling your requests and your responses. It handles both requests and responses coming in from clients and the model life cycle.
- The backend uses model workers that are running instances of the model loaded from a model store.
- They're responsible for performing the actual inference.
- You can see that multiple workers can be run simultaneously on TorchServe.
- They can be different instances of the same model or they could be instances of different models.
- Instantiating more instances of a model enables handling more requests at the same time and can increase the throughput.
- A model is loaded from cloud storage or from local hosts.

- TorchServe support serving of eager mode models and jet saved models from PyTorch.
- The server supports APIs for management and inference, as well as plugins for common things like server logs, snapshots, and reporting.
- As well as TF Serving, NVIDIA's Inference Server, and TorchServe, you may also want to look at Kubeflow Serving. There's a bit too much to go into detail here, but let's look at it briefly.



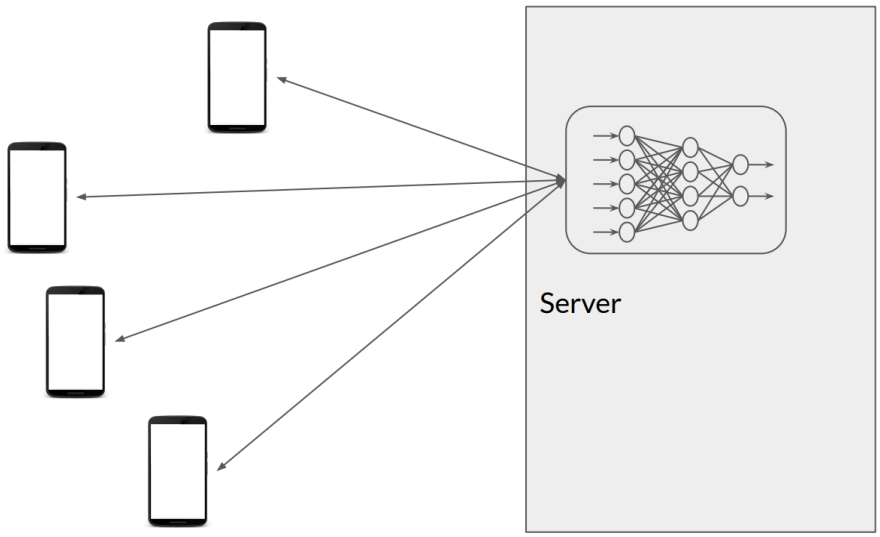
- Kubeflow also offers serving capabilities with Kubeflow Serving.
- This allows you to use a compute cluster with Kubernetes to have serverless inference through abstraction.
- It works with TensorFlow, PyTorch and others.
- I won't go into the detail on it here, but you can learn more about it at the URL provided in the readings.
- Now that you've seen how serving of machine learning models can be achieved, let's take a step back and explore scaling applications like these.
- You will understand the fundamentals of horizontal and vertical scaling, and see how you can use virtualization, containers, and container orchestration to manage the serving of your apps at an appropriate scale using open-source services.

Scaling Infrastructure

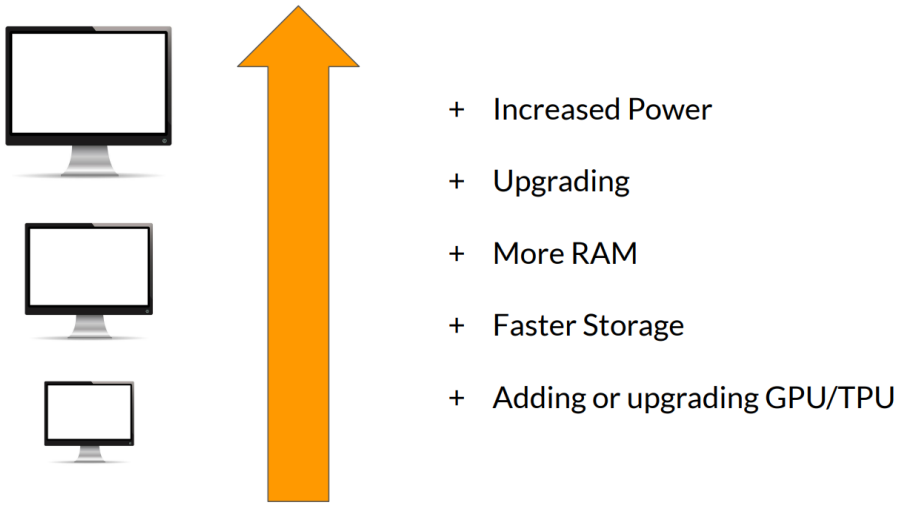


- Previously you looked at some model serving architectures and how they can work to provide an interface to a model.
- Of course, when it comes to serving a model like this, managing scale and lots of services becomes paramount.
- Next you look at what scaling is, why it's important, and some services that are available to give you scaling in your apps and your machine learning workloads.
- Consider a model, and here's a very simple example, as most models will be far larger and much more complex than this one.
- Consider the costs of **training deep neural networks with billions of operations** on huge datasets.
- It could take days to complete training on a standard CPU or a single GPU.
- If you can scale out the hardware on which the training runs and then distribute the training across different items of hardware and maybe even distribute the data by sharding it across this hardware, you can make that training far more efficient.
- Similarly, you could also consider the cost of training a network beyond just the data.
- The larger and more sophisticated the network, the **more parameters that need to be tuned and fine**-tuned.
- A simple neuron like this one has two floating point parameters. Consider large networks that have many millions of parameters that need to be learned.

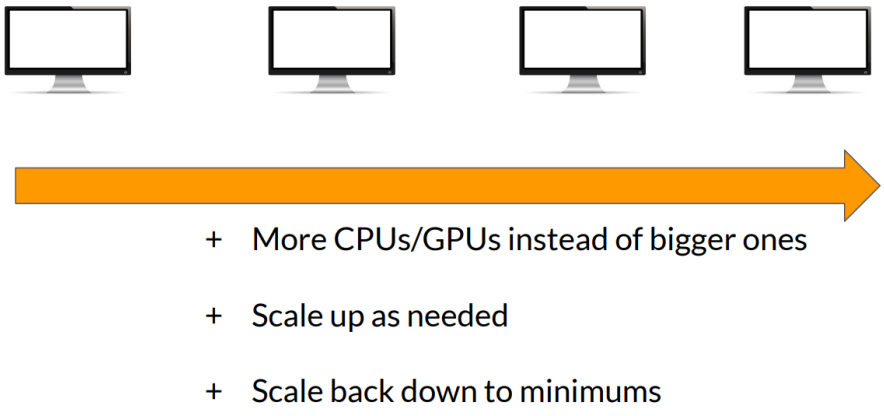
Why is Scaling Important?



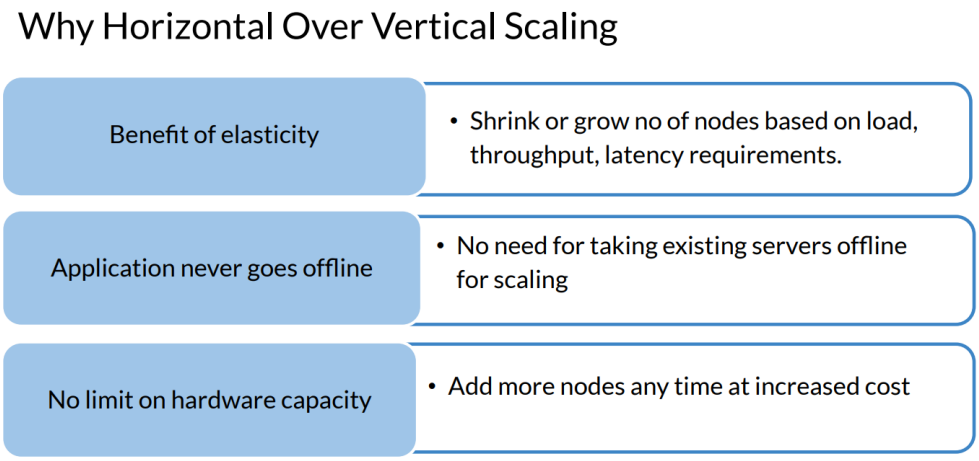
- Not only that, consider what happens when you have deployed your model to a server.
- **Huge volumes of requests to the server for inference can overwhelm it.**
- So the ability to scale the **runtime inference**, as well as the **training** is vital.



- There are two main ways to scale, horizontal and vertical. So let's start with **vertical**.
- It's pretty straightforward and as the image suggest, it's using bigger and more powerful hardware.
- It might be upgrading your CPU, adding more RAM, using newer GPU's and other types of power.
- If your car could only hold five people and you need to move 100 people, you can get a bigger car that holds 10 people, so then you can move twice as fast.

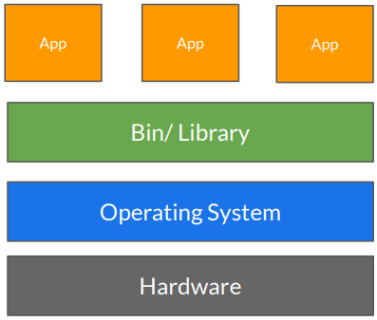


- Or horizontal scaling, which means adding more devices to the network. It adds more GPUs or CPUs when the load increases.
- Instead of buying a bigger car, you could get 20 cars the same size and transport all 100 people at once.
- To follow that metaphor, you could just borrow the other 19 cars along with your own for the time that you need them.
- That's basically the same concept with cloud computing, where you can scale up to your need and scale right back down again when you don't need it anymore, and only pay for what you use.



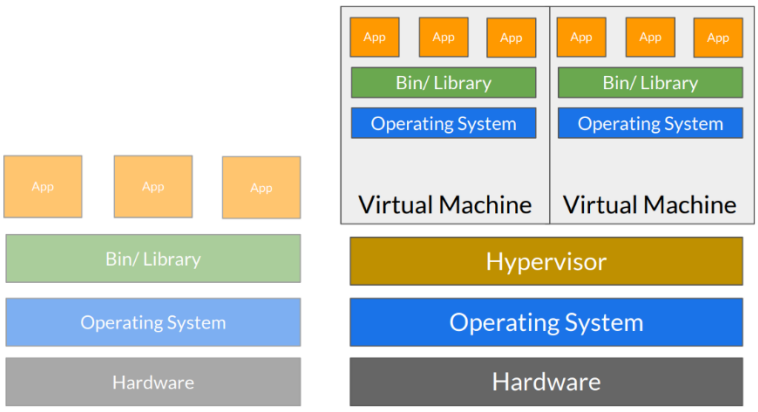
- **I would generally recommend horizontal scaling** for a variety of reasons.
- First of all is elasticity.
- Like my '100 people in cars that fit five people' scenario, instead of getting rid of your reliable car to get a bigger one and only chip away at the problem, you could lease 19 cars just like your own and give them back when you're done.
- That way you don't have to continually maintain and ensure all of the other cars, etc.
- Not only that, if you're scaling **vertically**, you generally have to take your app offline in order to upgrade the hardware resources.
- When it's elastic, you don't need to do this, you just **spin up new ones**.
- So some frameworks such as Google's App Engine are also really smart in using machine learning to predict usage patterns so that they can pre-warm up the machines before they're actually needed, reducing overall latency.
- Of course, there are limitations, but they usually **budgetary** and not hardware.
- If you need more nodes and you can afford them, you can just go get them.
- There are lots of vendors offering **cloud platforms that allow you to scale horizontally**. Keep an eye on what they offer when it comes to the overall scaling of your system, and I'll generally keep an eye out for three things.
    - Can I manually scale? What happens if I say I only want n instances of a VM?
    - Can I autoscale? What happens if I want my app to automatically spin up and down based on demand? What does latency and costs look like?
    - Finally, how aggressive is the system at spinning up and down based on my need?
- Then of course, the next question arises, and that is, how can I manage my additional VMs to ensure that they have the **content** on them that I want?
- For machine learning, there might be a lot of **dependencies, access to data, permissions, and many other configurable items**.
- If I'm going to scale horizontally, I want the new machines to be able to be up and running quickly.
- For that, there's **containerism**. Let's explore what that's all about next.

Typical System Architecture

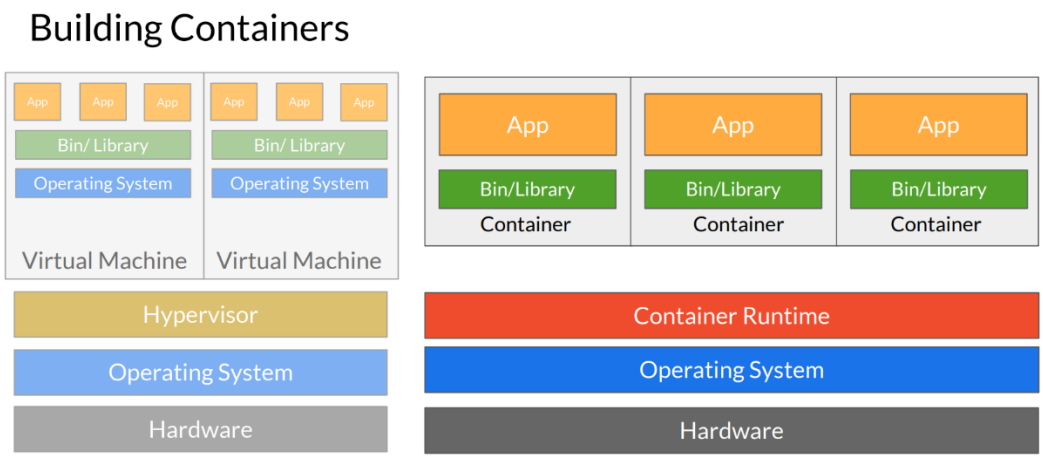


- Let's think about a typical system where I may have a number of apps that are running.
- The pattern will generally look like this, where I have my apps running using some binaries or libraries within an operating system that's executed by hardware.

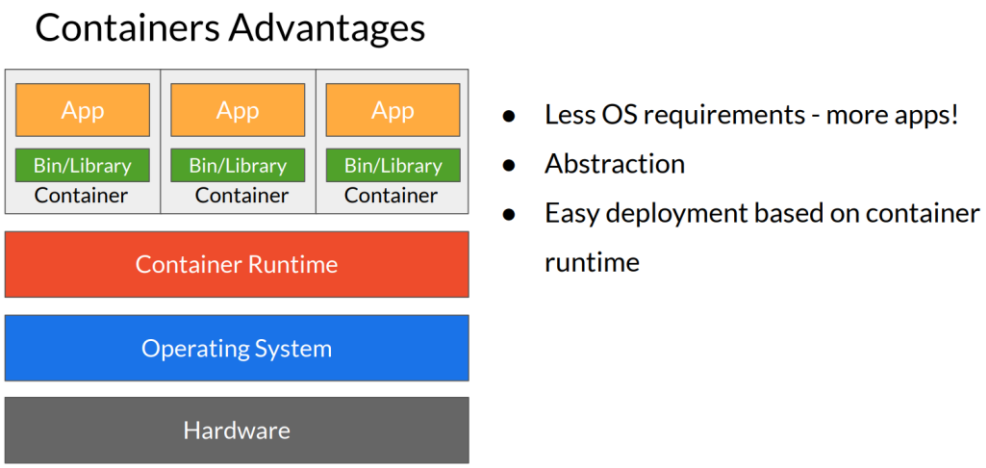
Virtual Machine (VM) Architecture



- An extension to this is the virtual machine architecture, where the apps can still run on bin library within an operating system, but that operating system does not run on hardware, and instead it runs on a virtual machine that has no hardware.
- This VM is managed by a hypervisor which acts as the manager of the virtual machines, each with its own operating system bin and apps as shown.
- The hypervisor will typically run on the metal of the hardware. Already, you can see how this could be used for horizontal scaling.
- Our hardware might support multiple instances of operating systems and apps as shown here.
- This is pretty cool, but it can face a **limitation**. Namely that each VM has to have a lot of stuff on it, not least a full operating system.
- As such, it might **not be the most efficient use of resources** in the system, and that's where containers can help.

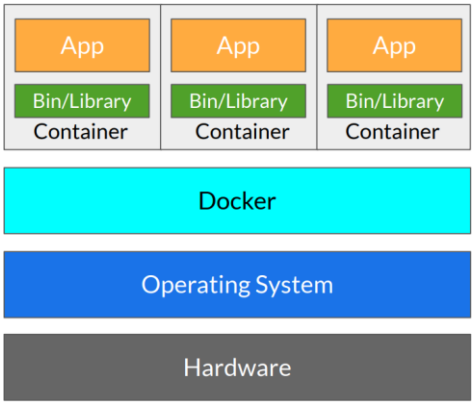


- As you can see, architecturally, they're quite similar to the VM concept, except they **don't require separate operating systems per partition.**
- Hence, they are much lighter in weight and the same hardware can typically manage more containers than virtual machines.



- **Consider the advantages of using containers, because each app instance doesn't require an operating system, you can generally run more of them on the same hardware, giving you much better horizontal scalability.**
- The abstraction of having them run within a container runtime also gives you this greater flexibility on hardware that supports a container runtime.
- You don't need to worry about the operating system, your app just works.
- This leads to easier deployment and more options in deployment.

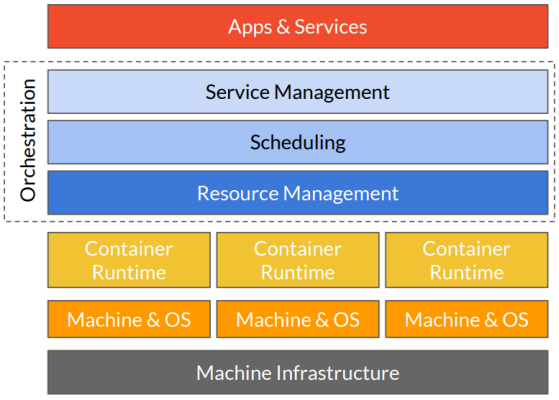
Docker: Container Runtime



- Open source container technology
- Most popular container runtime
- Started as container technology for Linux
- Available for Windows applications as well.
- Can be used in data centres, personal machines or public cloud.
- Docker partners with major cloud services for containerization.

- You've no doubt heard of Docker, which is the most popular container runtime.
- It started out as a Linux-based technology but spread rapidly to other operating systems like Windows, and it's widely used in data centers, personal machines, and public cloud infrastructures.
- You'll often see developer tutorials implemented as Docker instances, as this makes it much easier to handle complex dependencies or operating system quirks.
- Containers will offer you a really convenient way to do horizontal scaling, but then not without challenges of their own.
- For example, you might want to run **multiple containers on multiple machines and keep them all in sync.**
- Like any app, even one in a container, it's at risk of going down or the container host itself might fail, so you also want to keep a fleet of containers on a hot standby so you can switch traffic over should one go down.
- Thus, we'll need to keep the idea of **container orchestration** front of mind. Let's dig into that next.

Enter Container Orchestration

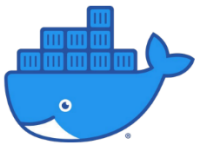


- Manages life cycle of containers in production environments
- Manages scaling of containers
- Ensures reliability of containers
- Distributes resources between containers.
- Monitors health of containers

- The idea behind container orchestration is simple. It's having a set of tools to do things like managing the lifecycle of containers, including their scaling.
- On top of your container manager, container orchestration generally gives multiple services such as **resource management** (to ensure that the containers aren't over-allocating hardware resources), **scheduling** (so that containers meet up and downtime requirements) and of course, general **service management** (so you can manage how the orchestration environment does its job).
- For example, you could use service management to ensure that a certain number of containers are **on hot standby** should some containers fail.
- Over time, as you understand how your system operates, you could tweak that number.

Popular Container Orchestration Tools

Kubernetes



Docker Swarm

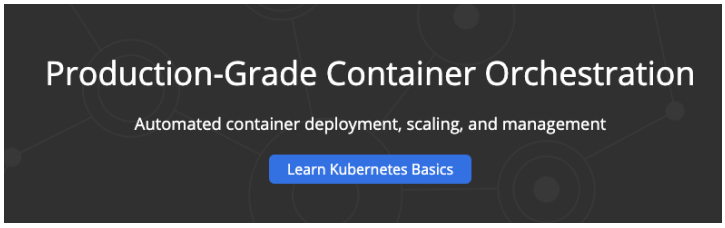
- **Two of the more popular container orchestration tools are Kubernetes and Docker Swarm**. Let's look into Kubernetes next as it underpins Kubeflow, a technology that lets you scale your ML apps and learning with horizontal scaling using containers.
- I won't go into a lot of detail about Kubernetes here, you can learn more about it at **kubernetes.io**.
- But in summary, it's an open source system for automating, deployment, scaling, and management of containerized applications.
- It **groups containers** that make up an application into logical units for easy management and discovery.
- Kubernetes builds upon many years of experience in building scalable applications at Google, combined with community experience, with a goal to bring you the best of both worlds.
- Check out the site and the link provided in the additional readings at the end of the lesson, and you'll see a video of how the Financial Times newspaper in London migrated hundreds of microservices to a container environment, scaling them horizontally using Kubernetes.
- Later this week you'll do a quick lab on TensorFlow Serving using Kubernetes with autoscaling.

ML Workflows on Kubernetes - KubeFlow

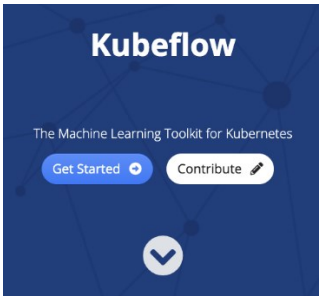
- Dedicated to making deployments of machine learning (ML) workflows on Kubernetes simple, portable and scalable.
- Anywhere you are running Kubernetes, you should be able to run Kubeflow.
- Can be run on premise or on Kubernetes engine on cloud offerings AWS, GCP, Azure etc.,

- **When it comes to deploying ML workflows on Kubernetes, Kubeflow evolved.**
- It's designed to make deployments of ML workflows so that data ingestion, feature extraction, training, model management, and all of those kind of things are portable and scalable using Kubernetes.
- An important attribute of Kubeflow is that it's designed so that anywhere Kubernetes can run, you can use it.
- Whether you're choosing a Cloud vendor, doing containers on your own premises, or some kind of combination of the two, your machine learning workflows will be able to run with it.
- Please visit kubeflow.org to learn more and in particular checkout their Getting Started to see how to install and use it.
- Now that you've been looking at scaling horizontally and exploring how to do that via containers and container orchestration with Kubernetes (**as well as Kubeflow specifically for machine learning workflows**), I recommend that you spend a little time to use these products before you go any further.

- First is Kubernetes. The site is https://kubernetes.io/, and at the top of the page, there's a big friendly button that says 'Learn Kubernetes Basics':



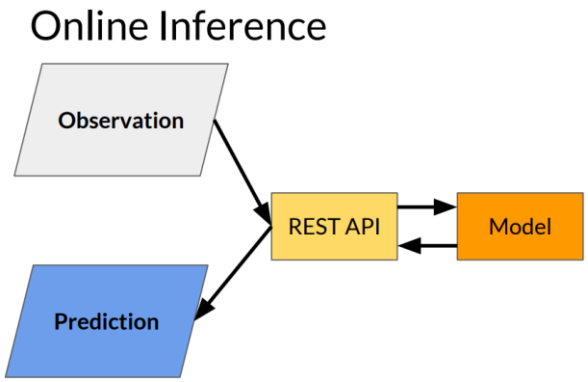
- Click on this, and you'll be taken to: https://kubernetes.io/docs/tutorials/kubernetes-basics/
- From here you can go through a lesson to create a cluster, deploy and app, scale it, update it and more. It's interactive, fun, and worth a couple of hours of your time to really get into how Kubernetes works.
- You may also want to check this video tutorial out
- For KubeFlow, visit: https://www.kubeflow.org/ , and at the top of the page, there's a Get Started button.



- Click on it to go to the tutorials. It doesn't have the nice interactive tutorials that Kubernetes has, but, if you can, try to at least install KubeFlow on one of the deployment options listed on this page – even if it's just your development machine.
- If you find it tricky to follow, don't worry because you will have an ungraded lab next week that will walk you through installing Kubeflow Pipelines (one of the Kubeflow components) in Kubernetes.
- In the meantime, you can watch this playlist particularly video #5 on Kubeflow Pipelines to get a short intro to this toolkit.

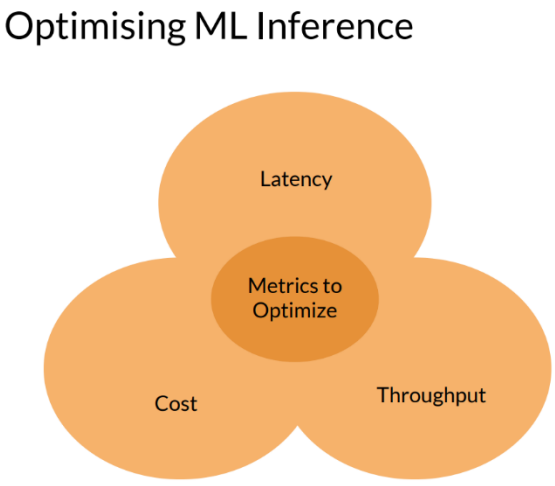
- Here's a fun case study on managing scale. In this extreme case, a famous boy band called 'One Direction' hosted a 10-hour live stream on YouTube, where they instructed fans to go visit a web site with a quiz on it every 10 minutes.
- This led to a really interesting pattern in scalability where the application would have zero usage for the vast majority of the time, but then, every 10 minutes may have hundreds of thousands of people hitting it.
- It's a complex problem to solve when it comes to scaling. It could be very expensive to operate.
- Using smart scaling strategies, Sony Music and Google solved this problem very inexpensively. Laurence isn't allowed to share how much it cost for the cloud services, but, when he and several of the other engineers went out for celebration drinks after the success of the project, the bar bill was more expensive than the cloud bill. (And they didn't drink a lot!)
- Check out the talk about how scaling worked for this system here: https://www.youtube.com/watch?v=aIxNm5Eed_8
- Learn about the event and the app here: https://www.computerweekly.com/news/2240228060/Sony-Music-Google-cloud-One-Directions-1D-Day-event-platform-services

Online Inference

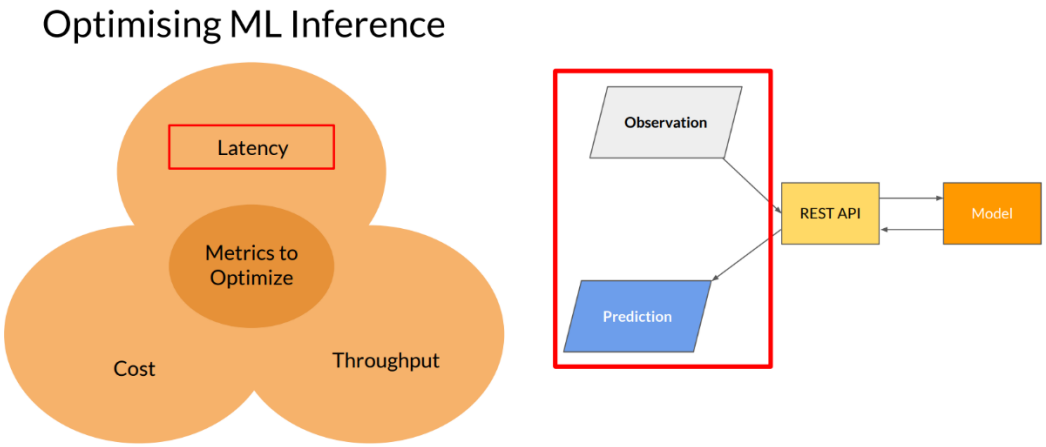


- Process of generating machine learning predictions in real time upon request.
- Predictions are generated on a single observation of data at runtime.
- Can be generated at any time of the day on demand

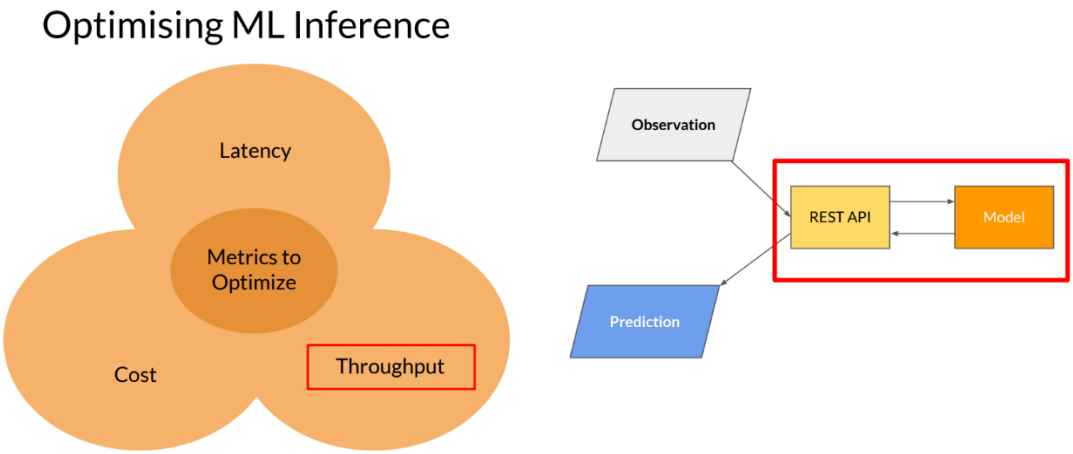
- Earlier you explore different infrastructures for serving models, as well as techniques and technologies for horizontal and vertical scaling.
- You also got hands on with a Colab to show you how to serve a simple model
- Next, you'll look at model serving and some of the things you need to take into account as **you serve models with online inference**.
- A typical interaction between a model and a caller online looks a bit like this.
- The interface between the outside world and the caller is via **REST** API.
- You'll typically have some form of data about the user often called an **observation** upon which you want to get a **prediction**.
- This might be, for example, context about a customer that can be used to predict what type of purchases may be appropriate for them so that you could have a **recommended** list.
- Or perhaps it might be something like a smart reply generator where in a conversation, the text can be used to auto generate replies that the user can select.
- The observation is posted to the model via rest API and the returned prediction is rendered



- In week one, I discussed the **metrics** to optimize for making inferences in a general scenario.
- Let's revisit them in the context of online inference
- If you want to optimize the system like this, you should think about this in three different axes.

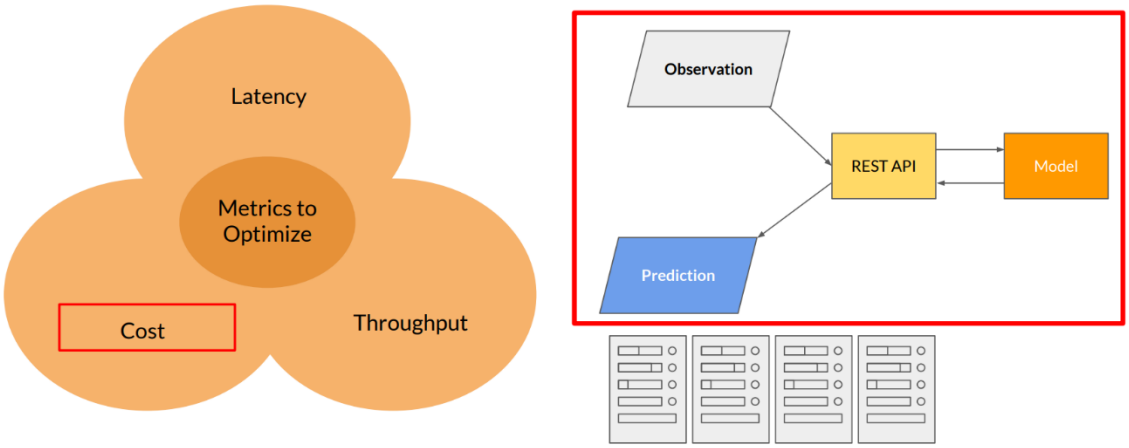


- You want to optimize for latency i.e. how long does it take for the data to be passed to the server, for the inference to execute, and for the response to be handled.
- But it's not just that, as latency comes in many forms. But from the user's perspective, it's the delay between their action and the response of their app.
- It's not just the inference, as you could have a **poorly designed app** that has a fabulous model and if latency is introduced by the transport of the data or even the rendering of the results, the user will see a delay.
- So it's **key to manage latency in all parts of your application design**,
- The focus of this course of course is on the model and the infrastructure, but it is important for you to keep an eye on **the total latency of your system**.



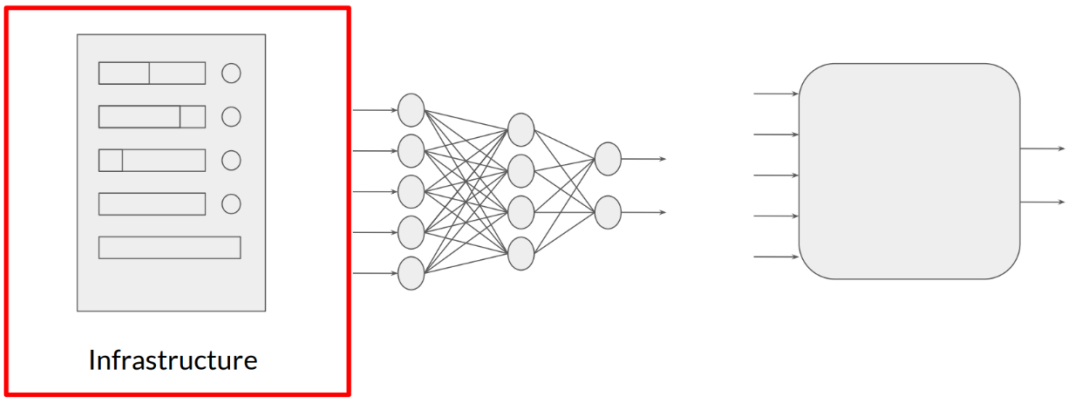
- This leads on to the need to understand throughput in the system, and how we optimize this
- It could be often with the type of horizontal scaling we spoke about earlier this week.
- While throughput is important for all systems, the **throughput measured in requests managed per unit time** is often more important for non-customer facing systems like intensive data processing apps.
- Still, it's important for you to keep it in mind for all of your systems

Optimising ML Inference



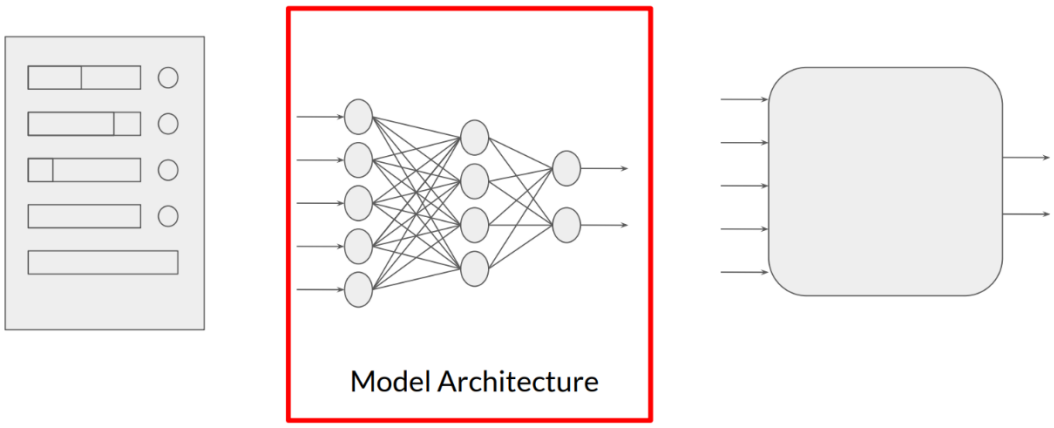
- Of course, there is cost. Most budgets aren't unlimited, so work that you do to make latency and throughput as efficient as possible will have a cost. You need to take this into account always
- There are multiple costs in your system, not just hardware, but things like engineering and testing time and effort, software licenses, opportunity costs for slow updates, and lost costs for new applications.

Inference Optimization: Infrastructure



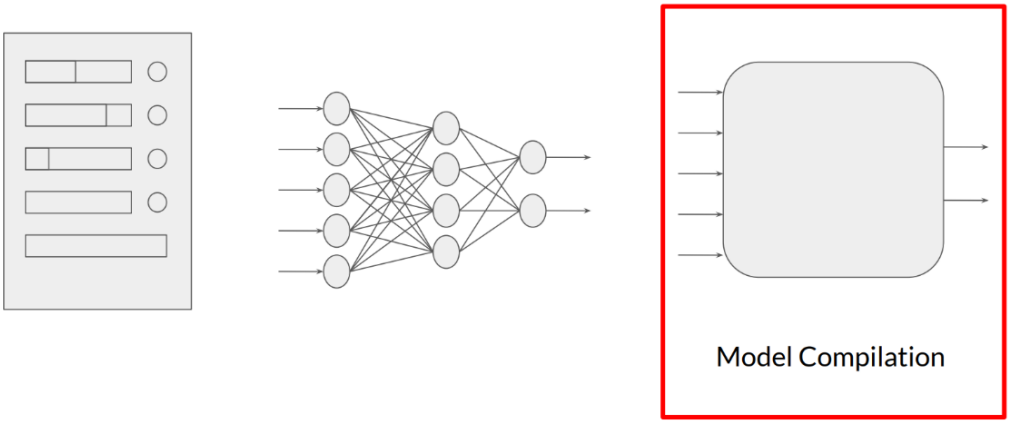
- Let's explore some **strategies for optimization** along with some technologies that can be used to **optimize for latency and throughput**.
- There are **three main areas** you can focus on if you want to optimize your inference.
- The first is the **infrastructure** used to serve the models and handle the user input and output.
- This can be scaled with additional or more powerful hardware as well as containerized virtualized environments like the ones we presented earlier this week.

Inference Optimization: Model Architecture



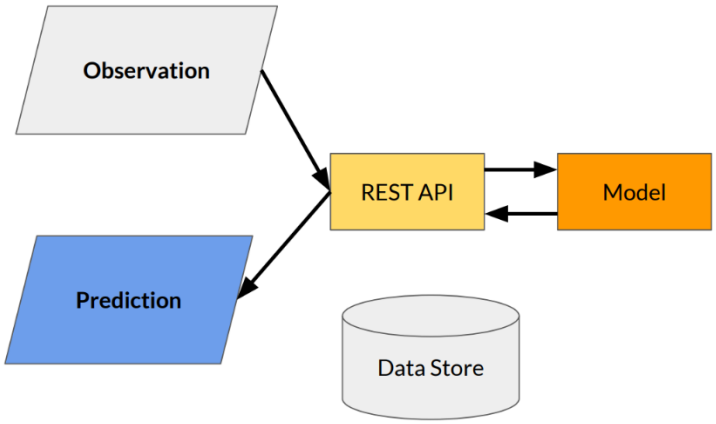
- The second of course, is to understand your model architecture and the metrics that it was trained and tested with.
- Often there's a **tradeoff between inference speed and accuracy**
- If a 99% accurate model is 10 times slower than a 98% accurate model, is it really worth the extra cost?

Inference Optimization: Model Compilation



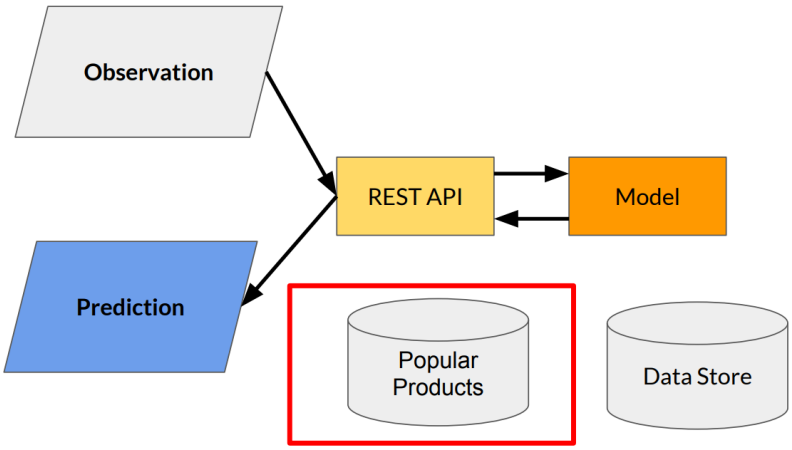
- And the third is model compilation.
- Model Compilation are running instances of the model responsible for performing the actual inference
- If you know the hardware on which you're going to deploy the model (for example, a particular type of GP), there's **often a post training step that consists of creating a model artifact and a model execution runtime** that's finally adapted to the underlying support hardware.
- You can **refine your model graph and inference runtime to reduce memory consumption and latency.**

Additional Optimizations

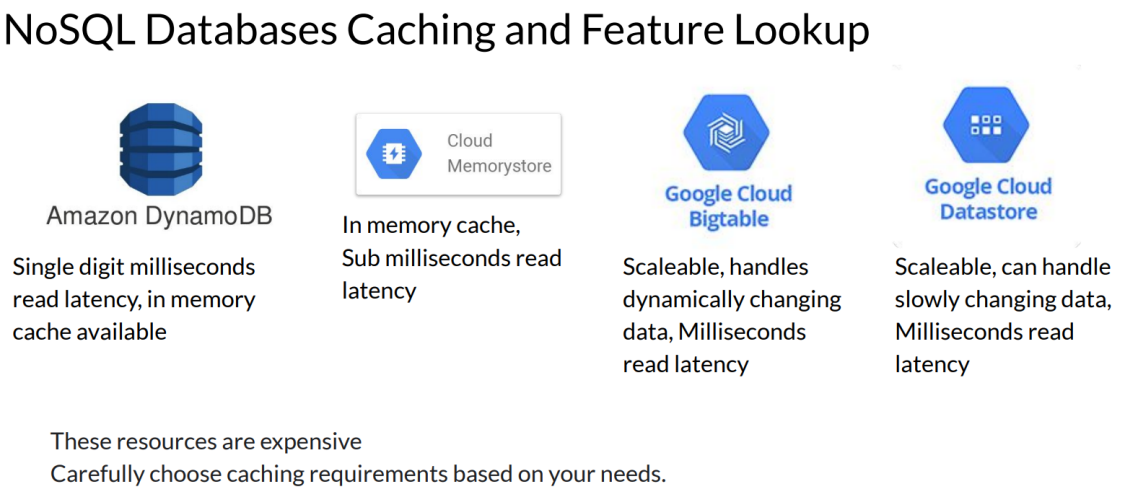


- Additionally, there are some optimizations that could be performed on the **application layer**.
- So for example, consider the scenario where you're doing a shopping prediction, giving your customer a list of products that they might want to buy next.
- The app will receive details about the customer, including some form of identifying this data is used by the model to generate an inference.
- And the model will return a bunch of IDs of products that might suit that customer.
- So the app has to look these up in a data store in order to get details about them, which it then returns to the client as a prediction.
- Now there's a lot of hitting database going on here, so an obvious optimization you can do is to consider common scenarios to be **cached** in something **faster** than a typical data store.

Additional Optimizations

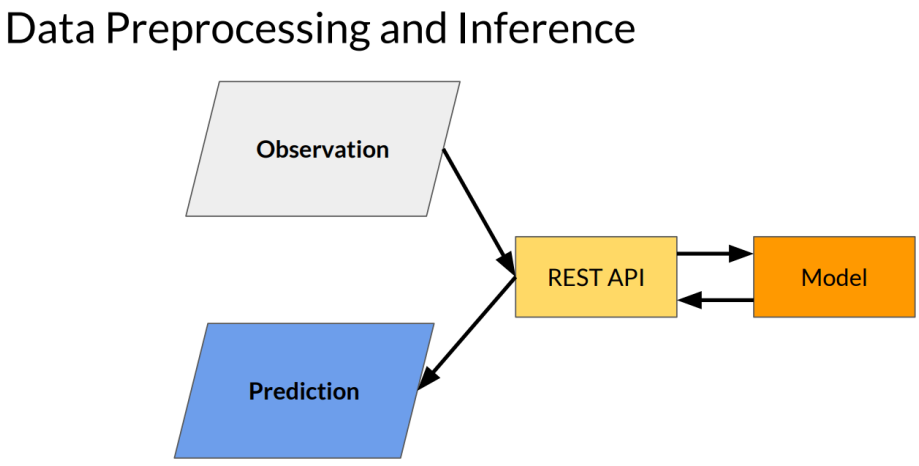


- So for example, if you have a number of hot popular products, these could be stored in faster data storage.
- **Of course the faster the storage, the more expensive it is**, so there's a tradeoff here and it may not be feasible for all of your data to be in such a store.
- If you can find that tradeoff for how many to put in there, you can **maximize for latency while minimizing your extra costs.**



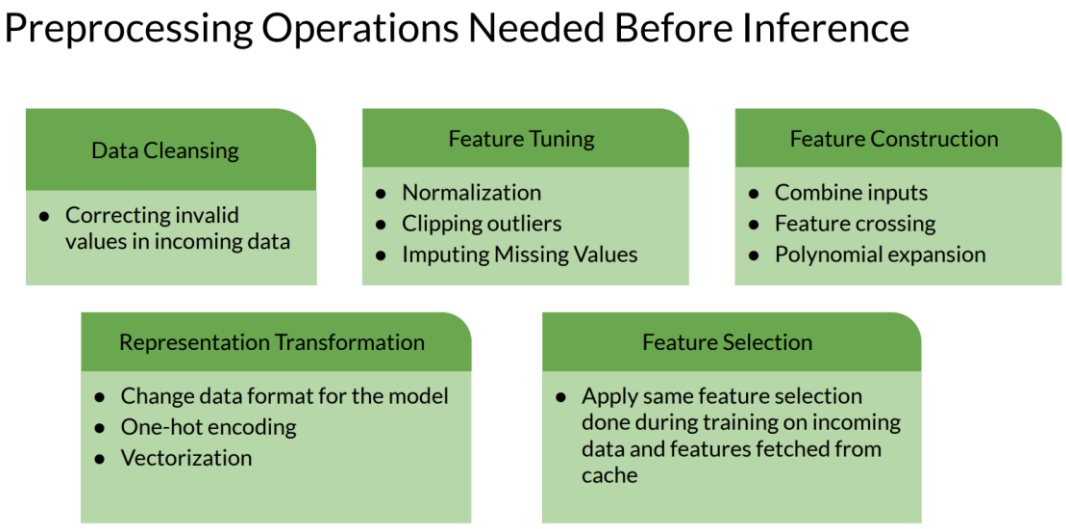
- Fast data caching is usually achieved using NoSQL databases on memory caching.
- There's a few products out there that can handle this such as Amazon's DynamoDB, Google Cloud's Memorystore, which used to be called MemCache.
- If you go back to the boy band case study that I used earlier, I use the term MemCache extensively
- We also have things like BigTable and Google Cloud Datastore that can handle large data sets quickly.
- Here you've looked at how online inference works from a very high level before going into a discussion about some scenarios and techniques for where you can optimize your ML inference, in particular looking at managing data within your app.
- There's another area where you can consider optimization and that's **in the data passed in and out of the model,** so you'll look next at preprocessing and post processing your data.

Data Preprocessing



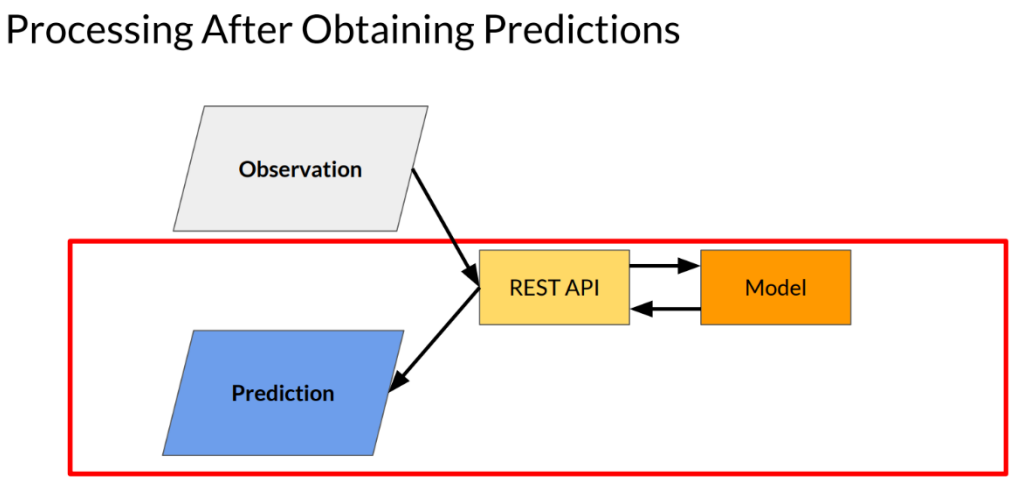
- Previously, you looked at some techniques on optimizing your app that uses models.
- You've explored hardware and container strategies, model architectures, and things like caching in your app to make sure that data access doesn't become a bottleneck.
- But there are other places where you can optimize, so let’s start by considering the input data to your app.
- If you go back to this very simple high level diagram of an app, remember that the observation data being passed into the system may be in one format, but that's not necessarily the same format that the model was designed to take in. The data has to be converted somehow.

- For example, consider a simple language model where maybe the observation is a sentence that the user typed and is stored as a string. The model is designed to classify that text to see if it's toxic.
- NLP models like this are trained on **input vectors** where words are transformed into a high dimensional vector and sentences are sequences of these vectors.
- Now that preprocessing of the data has to be done somewhere.
- And that's just a relatively simple example of where you need to do a translation of data from one format to another.



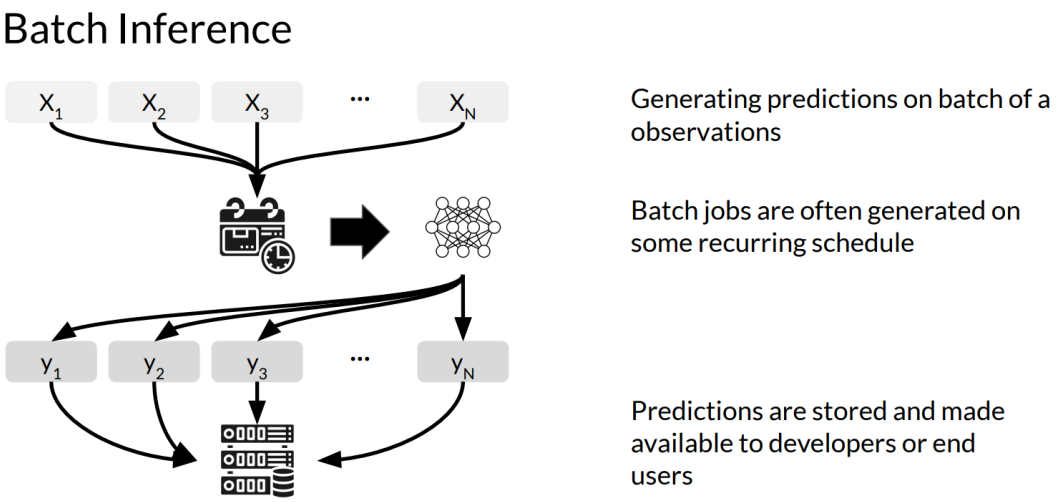
- Other areas to consider when preprocessing are things like **data cleansing**, where you correct invalid values in incoming data.
- For example, maybe you're building an image classifier and the user sends you a picture that's invalid because it's too big. You could reject it, of course, or you could take on the process of re-sizing it to get a valid picture size.
- Then there's **feature tuning** where you do some transformation on the data to make it suitable for the model.
- In the case of an image, this could be **normalization** where instead of having a 32 bit value to represent a pixel, you might convert it to three 8 bit values for red, green and blue, ignoring the alpha channel. And then instead of these having values between 0 and 255, you could convert them to values between 0 and 1 as **neural networks tend to deal better with normalized values** like that.
- Or with the example of a string coming in, it's encoding the string into the vocabulary representation that the model uses, which could **clip outliers** such as infrequently used words.
- Often models will require data preprocessing to involve **feature construction**. So for example, when it comes to a model that might predict the price of a house, the input data might have multiple columns such as the number of rooms and the size of each, but the model is trained on the total floor space of the house.
- So then **feature crossing** could be used to **multiply the values** that you have to get the feature type that the model uses.
- Other scenarios here could be a **polynomial expansion** where a new feature is calculated from a formula of the original feature. Maybe the data contains temperature in Celsius but the model expects Fahrenheit.
- Feature tuning and feature construction can also be a form of representation transformation, where the input data needs to be transformed for the model to understand it.
- One classic use of this is **one hot encoding** of data.
- And then there's the smart caching of features that you could have done during training.

- For example, in our sentence example, there may be common sentences such as ‘good morning’, that instead of us going through cleansing, tuning, construction and representation transformation, you could just have the input data format, pre transformed, cached and you use just that.

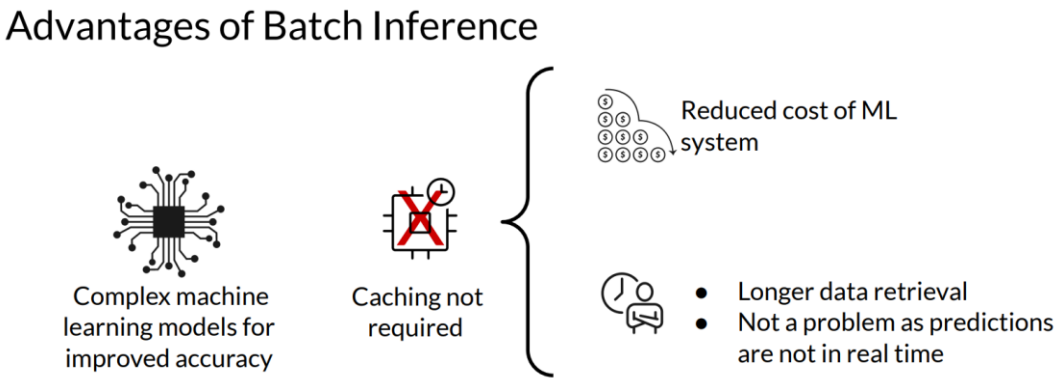


- But don't forget about **post processing**. Once your model has given its predictions back to the app, you still need to do something with those.
- For example, in a smart reply app, the model may give a few predictions about what the best next sentences could be, and these may be a **sequence of vectors representing words in a sentence, but not the sentence itself.**
- Your app will need to **convert these into a string** before returning them to the user. This is post processing of data. Everything that you saw in preprocessing can apply but usually in reverse.
- So now that you've seen what post processing looks like, let's explore a few products that are out there that can give you shortcuts to doing this task.
- Here is a reading note that introduces you to Apache Beam and TensorFlow Transform, which really help with the task of pre-processing:
    - Apache Beam is a product that gives you a unified programming model that lets you implement batch and streaming data processing jobs on any execution engine. It’s ideally suited for data preprocessing!
    - Go to https://beam.apache.org/get-started/try-apache-beam/ to try Apache Beam in a Colab so you can get a handle on how the APIs work. Make sure you try it in Python as well as Java by using the tabs at the top.
    - You can learn about TensorFlow Transform here: https://www.tensorflow.org/tfx/transform/get_started . It also uses Beam style pipelines but has modules optimized for preprocessing Tensorflow datasets.

Batch Inference Scenarios

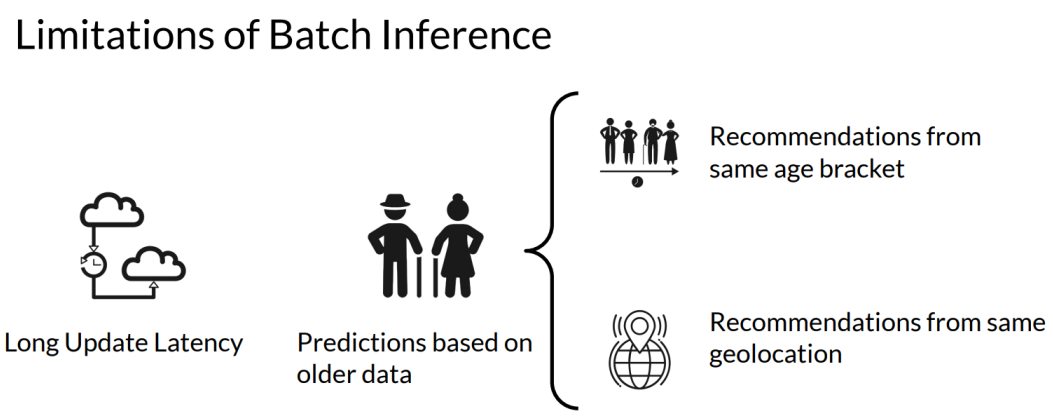


- You've looked at model scaling as well as architectures for inference.
- Now let's consider model performance and resource requirements for **batch inference**.
- After you train, evaluate and tune a Machine Learning model, the model is deployed to production to generate predictions.
- An ML model can provide predictions in batches, which will be applied to a use case sometime in the future.
- Predictions based on batch inference is when your ML model is used in a batch scoring job for a large number of data points where predictions are not required or not feasible to generate in real-time.
- In batch recommendations, for example, you might only use historical information about customer item interactions to make the prediction **without any need for real-time information**.
- Batch recommendations are usually performed in **retention campaigns for inactive customers** that have a **high propensity to churn** or in promotion campaigns and stuff like that.
- Batch jobs for prediction are usually generated on some recurring schedule like daily, at night or maybe weekly.
- Predictions are usually stored in a database that can then be made available to developers or end-users.

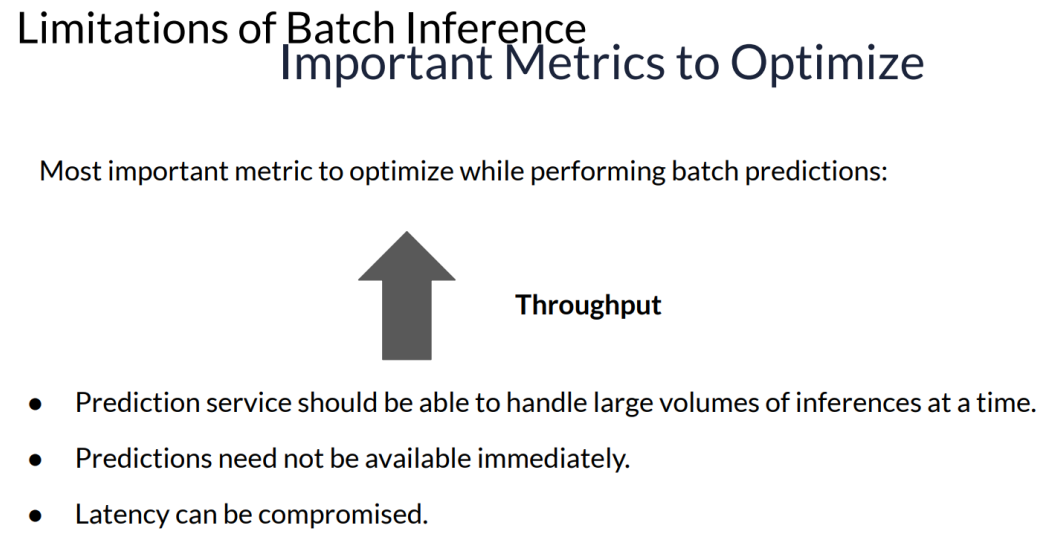


- Batch inference has some important advantages. You **can use complex** machine learning models in order to improve the accuracy of your predictions since there's **no constraint on inference time**.
- Also, caching of predictions like this is usually not required. Employing a caching strategy for features needed for prediction will increase the overall cost of your ML system. The data retrieval can take some time if no caching strategy is employed.

- Batch inference can also wait for data retrieval to make predictions since the predictions are not available in real-time.



- However, batch influence also has a few disadvantages. Predictions cannot be made available for real-time purposes. Update latency of predictions can be hours or sometimes even days.
- Predictions are often made using old data. This is problematic in certain scenarios. Suppose a service like a movie streaming one generates recommendations at night. If a new user signs up they may not be able to see personalized recommendations right away.
- To get around this, the system is designed to show recommendations from other users in a similar demographic like the same age bracket or maybe the same geolocation as a new user.



- The most important metric to **optimize** while performing batch predictions is **throughput**.
- You should always aim to increase the throughput in batch predictions rather than latency.
- When data is available in batches, the model should be able to process large volumes of data at a time.
- As throughput increases, the latency with which each prediction is generated increases also.
- But this is not a big concern in batch prediction systems since predictions need not be available immediately.
- Predictions are usually stored for later use and hence, latency can be compromised.

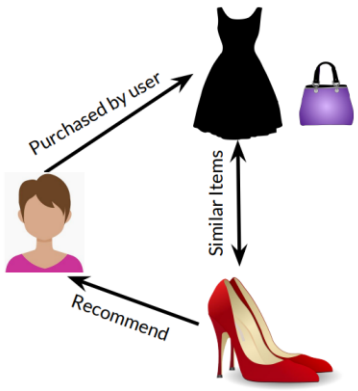
Limitations of Batch Inference

How to Increase Throughput?

- Use hardware accelerators like GPU's, TPU's.
- Increase number of servers/workers
  - Load several instances of model on multiple workers to increase throughput

- Throughput of an ML model or Production System processing data in batches can be increased by usage of hardware accelerators like GPUs, TPUs and all that.
- You can also increase the number of servers or workers in which the model is deployed.
- You can load several instances of the models and multiple workers to increase the throughput.

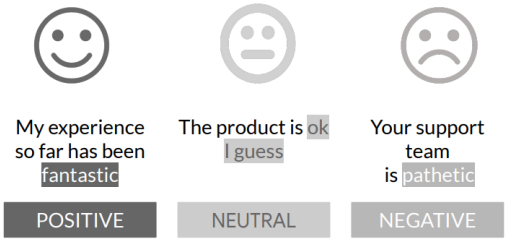
Use Case - Product Recommendations



- E-commerce sites: new recommendations on a recurring schedule
- Cache these for easy retrieval
- Enables use of more predictors to train more complex models.
  - Helps personalization to a greater degree, but with delayed data

- Let's look at some use cases of batch predictions.
- First, new product recommendations on an e-commerce site can be generated on a recurring schedule.
- Then caching these predictions for easy retrieval rather than generating them every time a user logs in.
- This can save inference costs since you don't need to guarantee the same latency as real-time inference needs to have.
- You can also use more predictions to **train more complex models** since you don't have the constraint of prediction latency.
- This **helps personalization to a greater degree,** but using delayed data that may not include new information about the user.

Use Case - Sentiment Analysis



- User sentiment based on customer reviews
- No need for realtime prediction for this problem
- CNN, RNN, or LSTM all work for this problem
- These models are more complex, but more accurate for the problem
- More cost effective to use them with batch prediction

- Let's look at a sentiment analysis problem. Based on the users' reviews, usually in text format, you might want to predict if a review was positive, neutral or negative.
- Systems that analyze user sentiment for your products or services based on customer reviews can make use of a batch prediction on a recurring schedule.
- Some systems might generate products sentiments on a weekly basis, for example.
- Real-time prediction is not needed in this case since the customers and stakeholders are not waiting to complete an action in real time based on the predictions.
- Sentiment predictions can be used for improvements of services or products over time.
- As you can see, it's not really a real-time business process.
- A CNN, RNN or LSTM based approach can be used for sentiment analysis.
- These models are more complex but they often provide higher accuracy. That makes it more cost-effective for you to use them with batch prediction.

Use Case - Demand Forecasting

- Estimate the demand for products for inventory and ordering optimization
- Predict future based on historical data (time series)
- Many models available as this is a batch predictions problem

- Let's look at a different example, forecasting demand for products or services.
- You can use batch predictions for models that estimate the demand for your products perhaps on a daily basis for inventory and ordering optimization.
- It can be modeled as a time series problem since you're predicting future demand based on historical data.
- Since batch predictions have minimal latency constraints, time series models like AROMA, SARIMA or an RNN can be used over approaches like linear regression for more accurate prediction.

Batch Processing with ETL

Data Processing - Batch and Streaming

- Data can be of different types based on the source.
- **Batch Data**
  - Batch processing can be done on data available in huge volumes in data lakes, from csv files, log files etc.,
- **Streaming Data**
  - Real-time streaming data, like data from sensors.

- Now that you've seen batch processing of static data, now let's explore what it looks like with **time-series** data or other data types that are **updated frequently** and which you need to read in as a stream.
- Data can be of different types based on the source.
- Large volumes of batch data are available in **data lakes** from CSV files, log files, etc.
- Streaming data, on the other hand, arrives in real-time. One example of such data could be data from sensors.

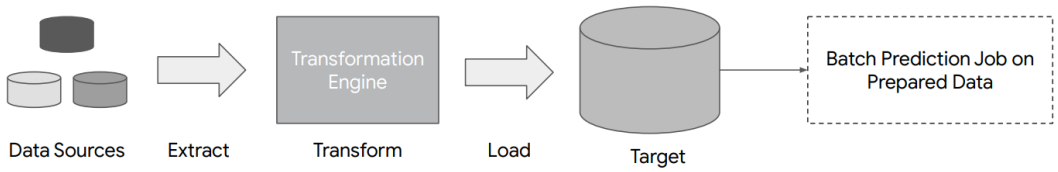
ETL on Data

- Before data is used for making batch predictions:
  - It has to be extracted from multiple sources like log files, streaming sources, APIs, apps etc.,
  - Transformed
  - Loaded into a database for prediction

This is done using ETL Pipelines

- Before data is used for making batch predictions, it has to be extracted from the multiple sources like the log files and the CSV files we spoke about, but it could also be APIs, other apps, streaming sources, and all that kind of thing.
- The extracted data should be transformed so as to make ML predictions, and then loaded into a database from where it can be sent in batches for prediction.
- The entire pipeline that prepares data is known as an ETL pipeline. ETL stands for extract, transform and load.

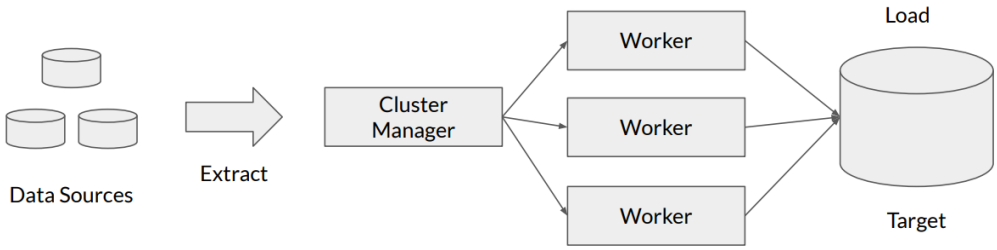
ETL Pipelines



- Set of processes for
  - extracting data from data sources
  - Transforming data
  - Loading into an output destination like data warehouse
- From there data can be consumed for training or making predictions using ML models,

- **ETL pipelines are a set of processes** for extracting data from data sources, then transforming the data, and then loading the data into an output destination like a data warehouse, from where it might be used for multiple purposes: encoding, running batch predictions, performing other analytics, doing data mining, and all that good stuff.

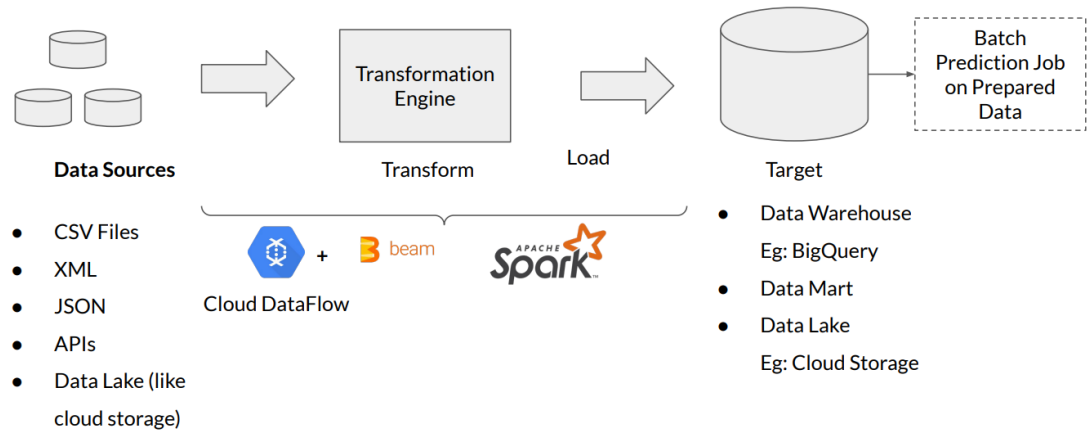
Distributed Processing



- ETL can be performed huge volumes of data in distributed manner.
- Data is split into chunks and parallely processed by multiple workers.
- The results of the ETL workflow are stored in a database.
- Results in lower latency and higher throughput of data processing.

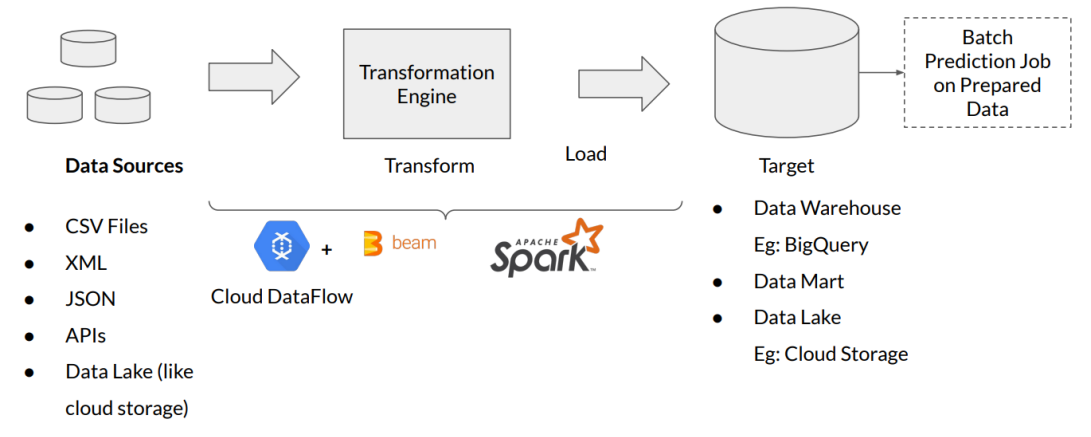
- Extraction from data sources and transformation on data can be performed in a **distributed** manner.
- Data is split into chunks and then can be parallel processed by multiple workers.
- The results of the ETL workflow are stored in a database, and the results are a lower latency and higher throughput of data processing.

ETL Pipeline components Batch Processing



- Let's take a look at the various frameworks that can be used for batch processing of data in ETL pipelines before they're sent for inference.
- Data can come from multiple sources, like CSV files, XML files, JSON, APIs, or data lakes like Google Cloud Storage.
- The **ETL on data is performed by engines like Apache Spark or Google Cloud Dataflow**, making use of the **Apache Beam** programming paradigm.
- The transformed data is stored into data warehouses like BigQuery and can be sent back into a **data lake like Google Cloud Storage**, before it's sent for batch prediction.

ETL Pipeline components Batch Processing



- Continuously updating data sources like sensors can be connected to Apache Kafka, Google Cloud Pub/Sub, and products like these.
- Cloud Dataflow, using Apache Beam, can perform ETL on streaming data also.
- Spark has a product specifically for processing streaming data, and Apache Kafka can also act as an ETL engine for streaming data.
- This data may in turn be collected into a data warehouse like BigQuery, or into a data mart or a data lake.
- It can also serve as a source for streaming data to another pipeline.

- Let's now put that into practice by exploring a scenario on the Google Cloud Platform where you use Beam and TensorFlow. While the data you use, molecular data about carbon, hydrogen, nitrogen, and oxygen is static and not streamed, it will give you an idea for how all of the components in beam-based systems will work together for learning.
- This optional lab will show you how to preprocess, train, and make batch predictions on a machine learning model using Apache Beam and Tensorflow Transform. To prevent costs of using Cloud resources, you will just run the entire pipeline in Colab. We linked the original article which gives the option to run in GCP in case you want to give it a shot afterward. **Click here to launch Colab!**
-
-

References

- TensorFlow Serving
- TorchServe
- KubeFlow Serving
- NVIDIA Triton