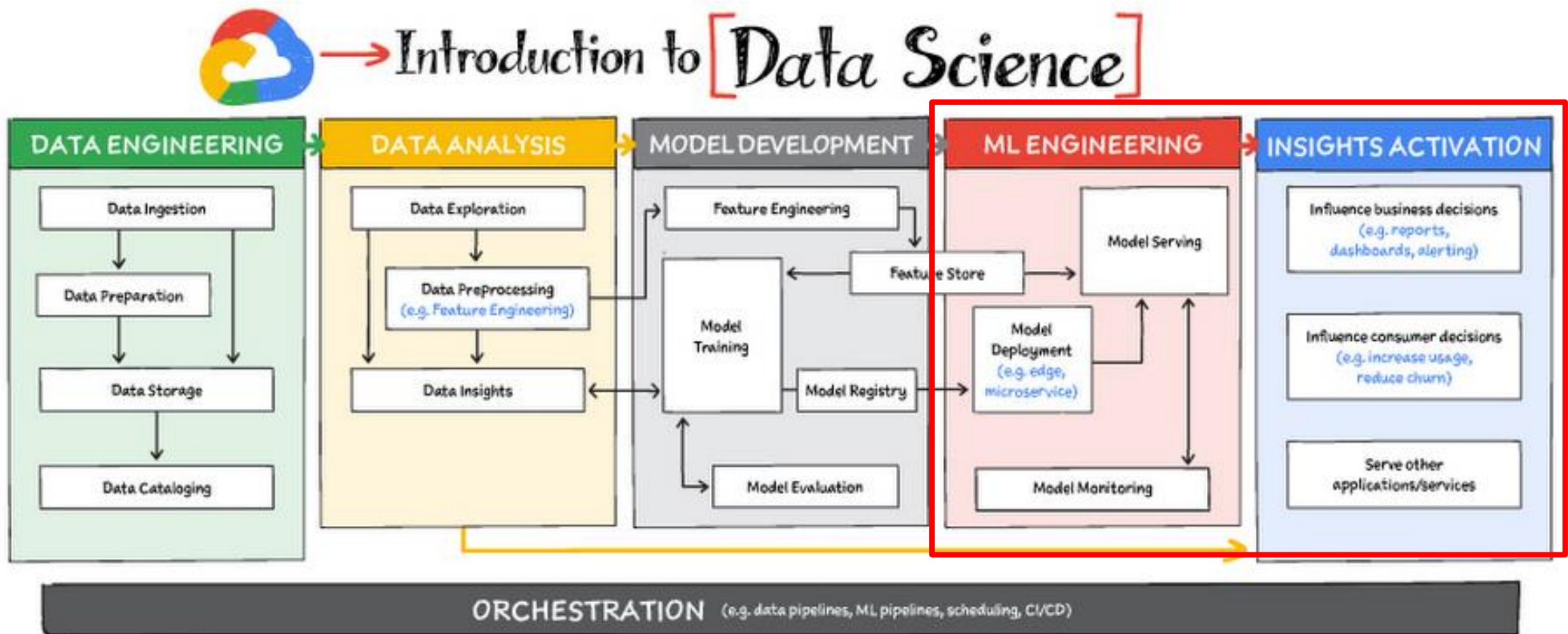


Model Severing

Szu-Chi Chung

Department of Applied Mathematics, National Sun Yat-sen University

The Pipeline



Ship an inference model

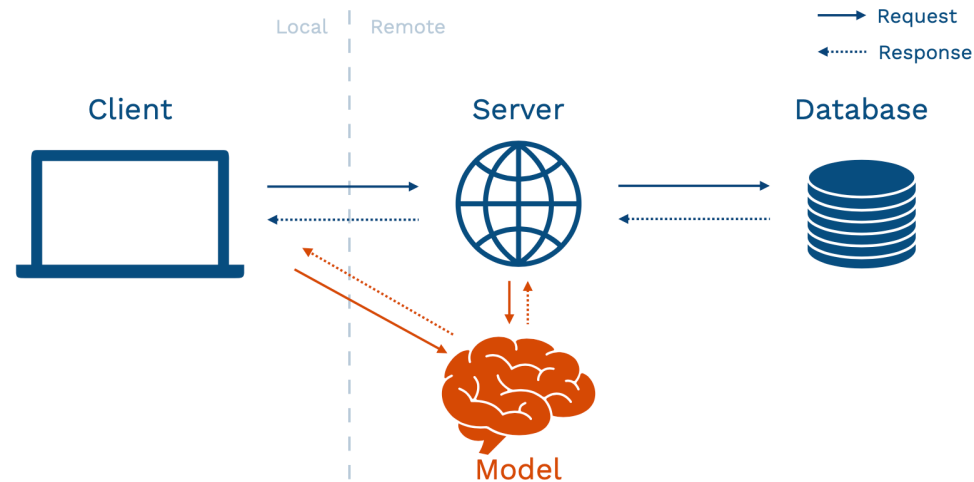
- ▶ A data science project doesn't end when you arrive at a Colab/Kaggle notebook that can save a trained model
 1. First, you may want to *export* your model to something other than Python:
 - ▶ Your production environment may not support Python at all - for instance, if it's a mobile app or an embedded system
 - ▶ If the rest of the app isn't in Python (it could be in JavaScript, C++, etc.), the use of Python to serve a model may induce significant overhead
 2. Second, since your production model will only be used to output predictions, rather than for training, you have room to perform various optimizations that can make the model faster and reduce its memory footprint!
 3. Finally, you should monitor the health and potential drifts of your system!

How to save a model?


- ▶ It's a good idea to include all the *preprocessing layers* in the final model you export so that it can ingest data in its natural form when deployed to production
 - ▶ This avoids deal with preprocessing separately within the application. It also makes it simpler to update them and limits the risk of mismatch between a model and the preprocessing steps
 - ▶ However, we can skip this step if we retrieve our features from the same source location for both training and serving, ie. from a feature store
- ▶ Using save utility that serializes and de-serializes a model
 - ▶ For reproducibility and quality control needs, when different environments should be taken into account, exporting the model in common format (e.g. Open Neural Network Exchange format (ONNX)) or even using a container might be good options

A Typical Example

- ▶ Once you have trained a model, you can easily use it in any Python code
 - ▶ But as your infrastructure grows, there comes a point where it is preferable to wrap your model in a small service whose sole role is to make predictions and have the rest of the infrastructure query it (e.g., via a REST or gRPC API)
 - ▶ This decouples your model from the rest of the infrastructure, making it possible to easily switch model versions or scale the service up as needed (independently from the rest of your infrastructure), and ensure that all your software components rely on the same model versions



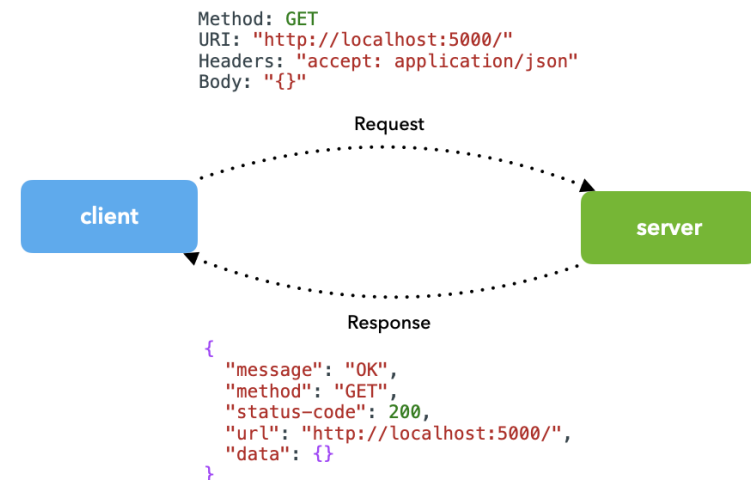
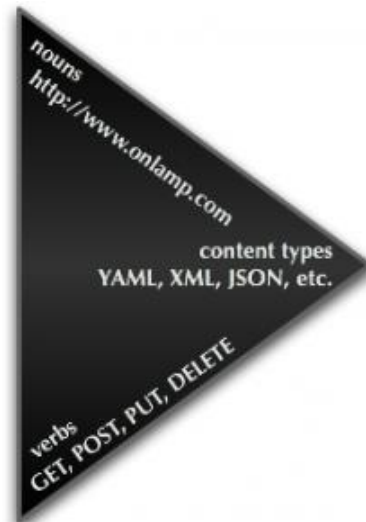
Deploy the model!

- Back to that you're starting your own data science consulting shop. You put up a fancy website, you notify your network. The projects start rolling in:
1. A personalized photo search engine for a picture-sharing social network—type in “wedding” and retrieve all the pictures you took at weddings
 2. Flagging spam and offensive text content among the posts of a chat app
 3. Building a music recommendation system for users of an online radio
 4. Detecting credit card fraud for an e-commerce website
 5. Predicting display ad click-through rate to decide which ad to serve to a given user at a given time
 6. Flagging anomalous cookies on the conveyor belt of a cookie-manufacturing line
- 



1. Deploying a model in remote server

- ▶ This is perhaps the common way to turn a model into a product: install model environment (Like `scikit-learn`, `XGBoost` or `Keras`) on a server or cloud instance, and query the model's predictions via a [REST API](#)
 - ▶ You could build your own serving app using something like `Flask` or `FastAPI` (or any other Python web development library)
 - ▶ `Keras/Tensorflow` also has its own library for shipping models as APIs, called [TensorFlow Serving](#). For other models, you can use [BentoML](#)



1. Deploying a model in remote server

- ▶ An important question when deploying a model as a REST API or webapp is whether you want to host the code *on your own*, or whether you want to use a fully managed third-party cloud service
 1. Compute engine: Google Cloud AI Platform lets you simply upload your model to Google Cloud Storage (GCS), and it gives you an API endpoint to query it. It takes care of many practical details, such as *batching predictions, load balancing, and scaling*
 2. Using container orchestration via [Kubernetes](#) for managed deployment is a good choice. There are also solutions for containers, such as SageMaker, VertexAI, etc
 - ▶ We want to be able to encapsulate all the requirements we need so that there are no external dependencies by using a container
 3. Use serverless options such as [AWS Lambda](#), [Google Cloud Functions](#), etc

1. Deploying a model in remote server

- ▶ You should use this deployment setup when
 1. The application that consumes the model's prediction will have reliable *access to the internet* (obviously)
 - ▶ For instance, if your application is a mobile app, serving predictions from a remote API means that the application won't be usable in airplane mode or in a low-connectivity environment
 2. The application does *not have strict latency requirements*: the request, inference, and answer round trip will typically take around 500 ms
 3. The input data sent for inference is *not highly sensitive*: the data will need to be available on the server in a decrypted form, since it will need to be seen by the model (but note that you should use SSL encryption for the HTTP request and answer)
- ▶ For instance, the image search engine project, the music recommender system, and the credit card fraud detection project are all good fits for serving via a REST API

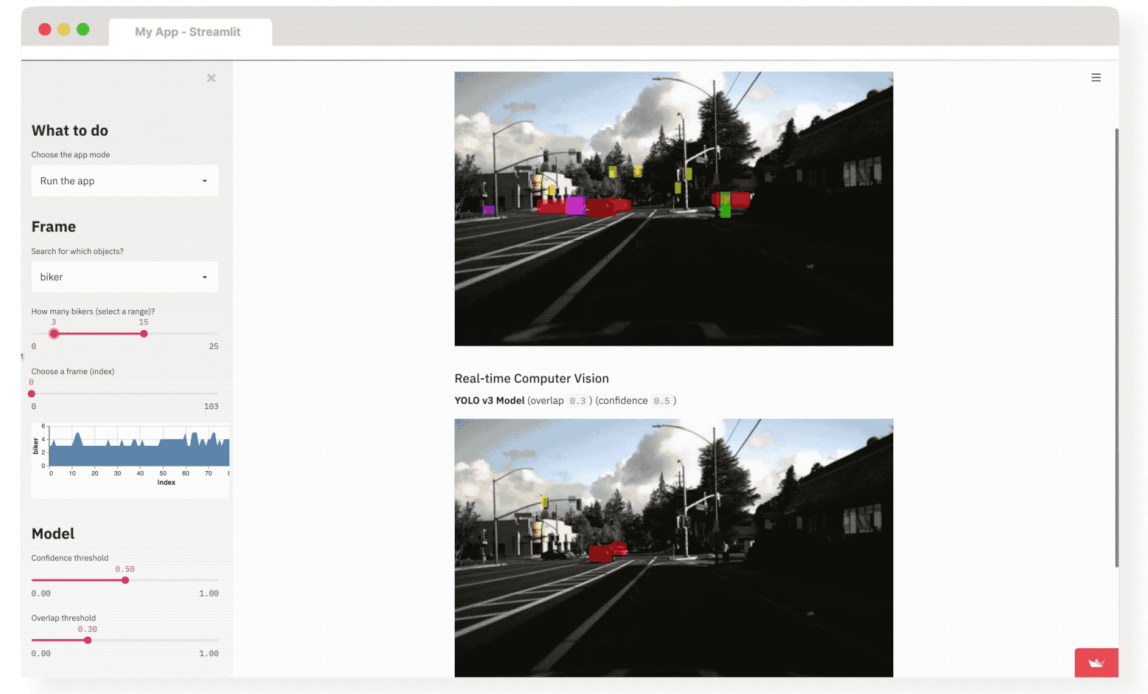
1. Deploying a model in remote server

- ▶ The REST API is nice and simple, and it works well when the input and output data are not too large. Moreover, just about any client application can make REST queries without additional dependencies
- ▶ However, it is based on JSON, which is text-based and fairly verbose
- ▶ It is inefficient in terms of serialization/deserialization time and payload size: many floats end up being represented using over 15 characters, which translates to over 120 bits for 32-bit floats! This will result in high latency and bandwidth. You may try other protocol like gRPC instead

The image shows a Swagger UI for a service named **iris_classifier** (version 0.0.0, OAS3). The UI lists several endpoints under the 'infra' section: `GET /healthz`, `GET /livez`, `GET /readyz`, and `GET /metrics`. Under the 'app' section, there is a `POST /classify` endpoint with the description 'InferenceAPI(NumpyNdarray() → NumpyNdarray())'. The 'Responses' section shows a 200 status code with a 'SUCCESS' description and a 'Media type' of 'application/json'. To the right, a terminal window shows the command `bentoml serve ./service.py:svc --reload --port 8050 &` being executed, with logs indicating the service is imported and running on `http://127.0.0.1:8050`.

2. Deploying a model in local computer

- ▶ Many models are often used in browser-based or desktop-based JavaScript applications
 - ▶ While it is usually possible to have the application query a remote model via a REST API, there can be key advantages in having the model run directly in the browser
 - ▶ This can be hosted on the laptop or on the user's computer!



2. Deploying a model in local computer

- ▶ You should only go with this option if your model is small enough that it won't hog the CPU, GPU, or RAM of your user's laptop or smartphone
 - ▶ Since the entire model will be downloaded to the user's device, you should make sure that nothing about the model needs to stay confidential
 - ▶ To deploy a model in JavaScript, the TensorFlow ecosystem includes [TensorFlow.js](#), a library for deep learning that implements almost all of the Keras API as well as lower-level TensorFlow APIs. You can easily import a saved Keras model into TensorFlow.js to query it as part of your browser-based JavaScript app or your desktop app
 - ▶ If you simply need a web interface, you can use [Streamlit](#) or [Gradio](#)

2. Deploying a model in local computer

- ▶ Optimizing your model for inference is especially important when deploying in an environment with strict constraints on available power and memory or for applications with low latency requirements!
- ▶ You should always seek to optimize your model. There are two popular optimization techniques you can apply:
 - ▶ *Weight pruning* — Not every coefficient in a weight tensor contributes equally to the predictions. It's possible to considerably lower the number of parameters in the layers of your model by only keeping the most significant ones. By deciding how much pruning you want to apply, you are in control of the trade-off between size and accuracy
 - ▶ *Weight quantization* — Models are often trained with single-precision floating-point (float32). However, it's possible to *quantize* weights to 8-bit signed integers (int8) to get an inference-only model that's a quarter the size but remains near the accuracy of the original model

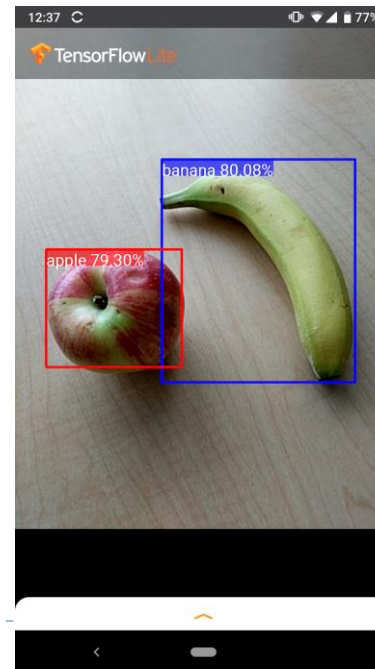
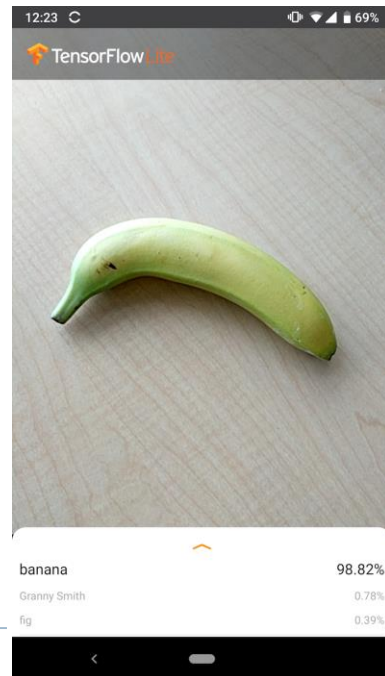
2. Deploying a model in local computer

► Use this setup when

1. You want to *offload compute* to the end user, which can dramatically reduce server costs
2. You need your app to keep working without connectivity, after the model has been downloaded and cached
 - When your web application is often used in situations where the user's connectivity is slow (e.g., a software for hikers), so running the model directly on the client side is the only way to make your website reliable
3. Your application has strict latency constraints. While a model running on the end user's laptop or smartphone is likely to be slower than one running on a large GPU on your own server, you don't have the extra 500 *ms* of network round trip!
4. The input data needs to stay on the end user's computer or phone. For instance, in spam detection project, the web version and the desktop version of the chat app should use a locally run model

3. Deploying a model on a device

- ▶ Sometimes, you may need your model to live on the same device that runs the application that uses it - maybe a smartphone, an embedded ARM CPU on a robot, or a microcontroller on a tiny device
- ▶ You may have seen a camera capable of automatically detecting people and faces in the scenes you pointed it at: that was probably a small model running directly on the camera



3. Deploying a model on a device

- ▶ To deploy a Keras/Tensorflow model on a smartphone or embedded device, your go-to solution is [LiteRT](#)
 - ▶ It's a framework for efficient on-device deep learning inference that runs on Android and iOS smartphones, as well as ARM64-based computers, Raspberry Pi, or certain microcontrollers. It includes a converter that can straightforwardly turn your model into the TFLite format
 - ▶ For other types of model, you can use [Kivy](#) or [BeeWare](#)
- ▶ For instance
 - ▶ Spam detection model for chat app will need to run on the end user's smartphone as part of the chat app, because messages are end-to-end encrypted and thus cannot be read by a remotely hosted model
 - ▶ Likewise, the bad-cookie detection model has strict latency constraints and will need to run at the factory

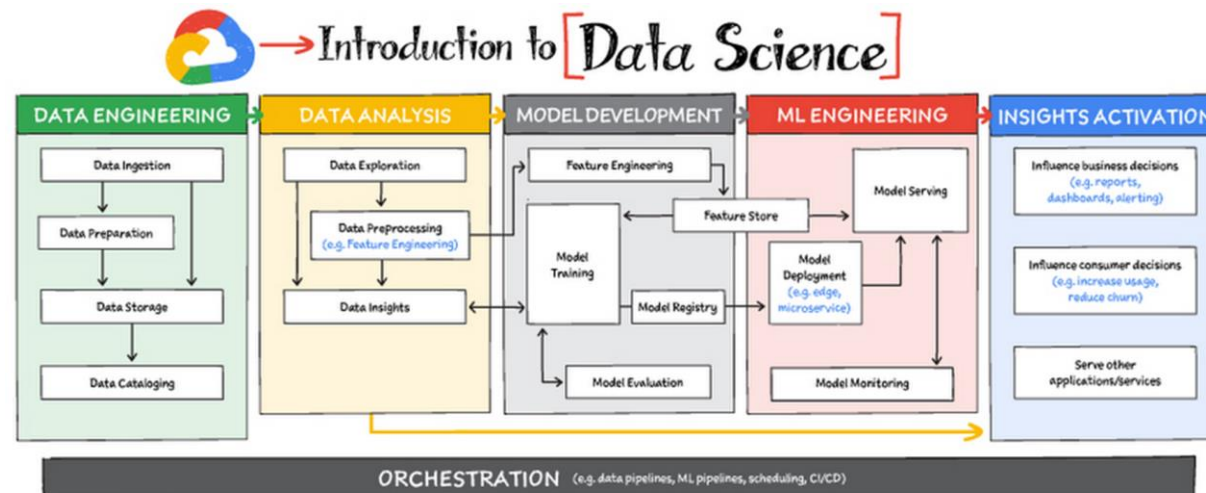
3. Deploying a model on a device

► You should use this setup when

1. Your model has *strict latency constraints* or needs to run in a low-connectivity environment. If you're building an immersive augmented reality application, querying a remote server is not a viable option
2. Your model can be made *sufficiently small* that it can run under the memory and power constraints of the target device. You can use the [TensorFlow Model Optimization Toolkit](#) to help with this
3. Getting the highest possible accuracy isn't mission critical for your task. There is always a trade-off between *runtime efficiency and accuracy*, so memory and power constraints often require you to ship a model that isn't quite as good as the best model you could run
4. The input data is *strictly sensitive* and thus shouldn't be decryptable on a remote server!

4. Monitor your model in the wild

- ▶ You've exported an inference model, you've integrated it into your application, and you've done a dry run on production data - the model behaved as expected
 - ▶ You've also written unit tests as well as logging and status-monitoring code. Now it's time to press the big red button and deploy to production
 - ▶ Even this is not the end! Once you've deployed a model, you need to keep monitoring its behavior, its performance on new data, its interaction with the rest of the application, and its eventual impact on business metrics



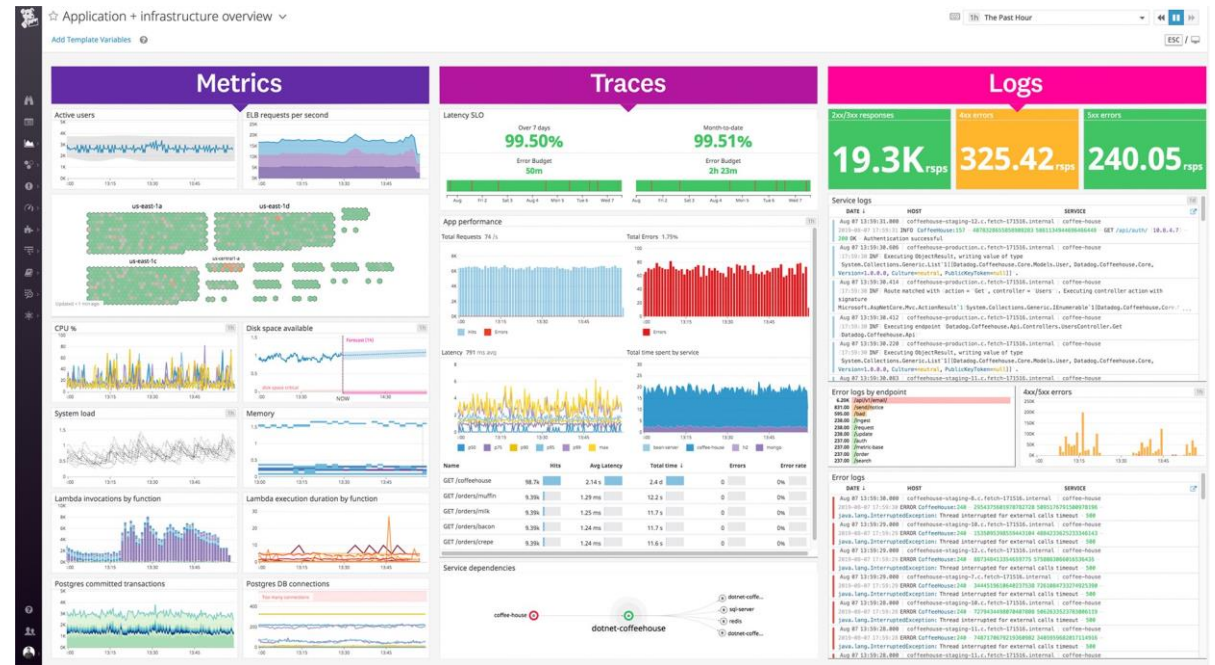
4. Monitor your model in the wild

▶ For instance

- ▶ Is user engagement in your online radio up or down after deploying the new music recommender system?
- ▶ Has the average ad click-through rate increased after switching to the new click-through-rate prediction model?
- ▶ If possible, do a regular manual audit of the model's predictions on production data. It's generally possible to reuse the same infrastructure as for data annotation
 - ▶ Send some fraction of the production data to be manually annotated, and compare the model's predictions to the new annotations. For instance, you should definitely do this for the image search engine and the bad-cookie flagging system
- ▶ When manual audits are impossible, consider alternative evaluation avenues such as user surveys (for example, in the case of the spam and offensive-content flagging system)

4. Monitor your model in the wild - Health

- ▶ The first step is to ensure that the actual system is up and running as it should
 - ▶ This can include metrics specific to service requests such as latency, throughput, error rates, etc. as well as infrastructure utilization such as CPU/GPU utilization, memory, etc
- ▶ Fortunately, most cloud providers and even orchestration layers will provide this insight into our system's health for free through a *dashboard*. In the case we don't, we can easily use [Grafana](#), [Datadog](#), etc. to ingest system performance metrics from logs to create a customized dashboard and set alerts

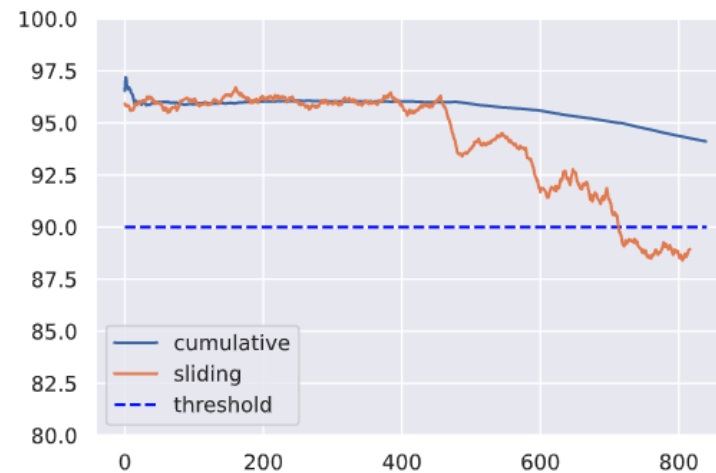


4. Monitor your model in the wild

- ▶ The model is subject to natural performance degradation over time, as well as unintended behavior, since the data exposed to the model will be different from what it has been trained on
 - ▶ This isn't something we should be trying to avoid but rather understand and mitigate as much as possible!
- ▶ Monitoring and data collection/Testing share a lot of similarities, such as ensuring that certain expectations around data completeness, distributions, schema are met
 - ▶ However, a key distinction is that monitoring involves *comparing live, streaming data distributions* from production to fixed/sliding reference distributions from training data

4. Monitor your model in the wild - Performance

- ▶ The next layer of metrics to monitor involves the model's performance
 - ▶ It's usually never enough to just analyze the cumulative performance metrics across the entire span of time since the model has been deployed
 - ▶ We should also inspect performance *across a period of time that's significant for our application (ex. daily)*. These sliding metrics might be more indicative and we might be able to identify issues faster by not obscuring them with historical data



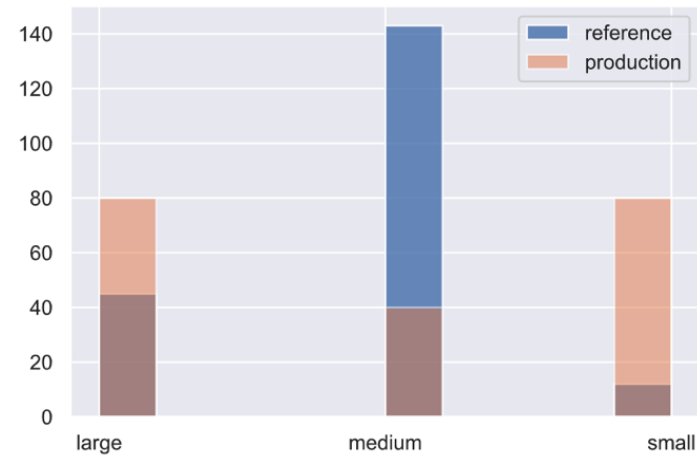
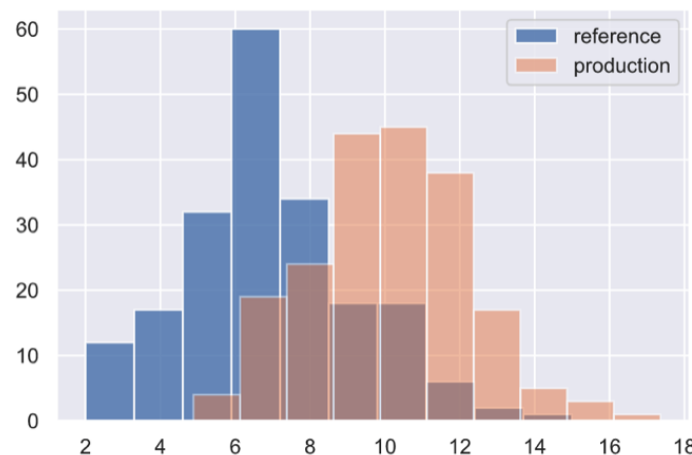
4. Monitor your model in the wild - Drifts

- ▶ We need to first understand the different types of issues that can cause our model's performance to decay (*model drift*). The best way to do this is to look at all the moving pieces of what we're trying to model and how each one can experience drift

		Drift
X	inputs (features)	Data drift $\rightarrow P(X) \neq P_{ref}(X)$
y	outputs (ground-truth)	Target drift $\rightarrow P(y) \neq P_{ref}(y)$
$p(y X)$	actual relationship between X and y	Concept drift $\rightarrow P(y X) \neq P_{ref}(y X)$

4. Monitor your model in the wild - Data drift

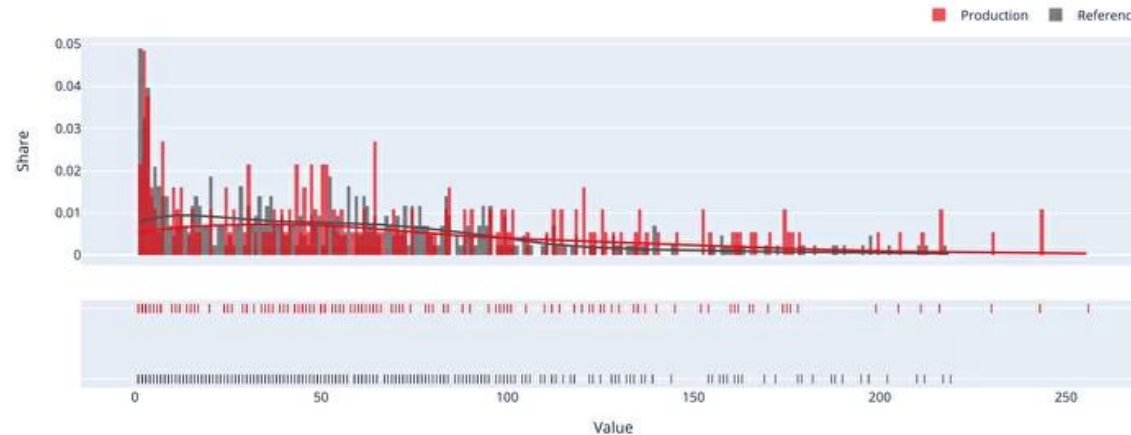
- ▶ *Data drift (feature drift/covariate shift)*, occurs when the distribution of the production data is different from the training data
 - ▶ The model can not deal with this drift in the feature space and its predictions may not be reliable. The actual cause of drift can be attributed to natural changes in the real-world but also to systemic issues such as missing data, pipeline errors, schema changes, etc
 - ▶ As data starts to drift, we may not yet notice significant decay in our model's performance, especially if the model is able to interpolate well. However, this is a great opportunity to potentially retrain before the drift starts to impact performance



4. Monitor your model in the wild - Target drift

- ▶ We can also experience drift in our outcomes which is *target drift*
 - ▶ This can be a shift in the distributions but also the removal or addition of new classes with categorical tasks. Though retraining can mitigate the performance decay caused target drift, it can often be avoided with proper *inter-pipeline communication* about new classes, schema changes, etc

Target Drift: detected, p_value=0.00829



4. Monitor your model in the wild - Concept drift

- ▶ We can have the actual relationship between them drift as well which is *concept drift*
 - ▶ This concept drift renders our model ineffective because the patterns it learned to map between the original inputs and outputs are no longer relevant. Concept drift can be something that occurs in various patterns
 - ▶ Monitor the *model performance* to detect concept drift
 - ▶ You may try to use online learning to alleviate it



4. Monitor your model in the wild - Concept drift

- ▶ No model lasts forever! You've already learned about *concept drift*: over time, the characteristics of your production data will change, gradually degrading the performance and relevance of your model
- ▶ The lifespan of your music recommender system will be counted in weeks. For the credit card fraud detection systems, it will be days. A couple of years in the best case for the image search engine. As soon as your model has launched, you should be getting ready to train the next generation that will replace it
 - ▶ Watch out for changes in the production data. Are new features becoming available? Should you expand or otherwise edit the label set?
 - ▶ Keep collecting and annotating data, and keep improving your annotation pipeline over time. In particular, you should pay special attention to *collecting samples that seem to be difficult* for your current model to classify - such samples are the most likely to help improve performance

4. Monitor your model in the wild - Locating drift

- ▶ Now that we've identified the different types of drift, we need to locate and how often to measure it. Here are the constraints we need to consider:
 - ▶ **Reference window:** the set of points to compare production data distributions with to identify drift
 - ▶ **Target window:** the set of points to compare with the reference window to determine if drift has occurred
 - ▶ Typically, the reference window is a fixed which is the recent subset of the training data while the target window slides over time

4. Monitor your model in the wild - Measuring drift

- ▶ Once we have the window of points we wish to compare, we need to know how to compare them
 - ▶ **Expectations** - The first line of measurement can be rule-based such as validating expectations around missing values, data types, value ranges, etc
 - ▶ **Univariate** - Once we've validated our rule-based expectations, we need to quantitatively measure drift. Traditionally, in order to compare two different sets of points to see if they come from the same distribution, we use two-sample hypothesis testing on the distance measured by a test
 - ▶ **Multivariate** - Measuring drift is fairly straightforward for univariate data but difficult for multivariate data

4. Monitor your model in the wild - Measuring drift

► Univariate

- Kolmogorov-Smirnov (KS) test or Chi-squared test

► Multivariate

- First perform dimensionality reduction – PCA or Autoencoder
- Two-sample tests
 - Maximum Mean Discrepancy (MMD): a kernel-based approach that determines the distance between two distributions
 - Kolmogorov-Smirnov (KS) Test + Bonferroni Correction

► Other metrics



Conclusion

- ▶ Model serving should consider
 - ▶ You need to determine how to save your model
 - ▶ Deploy the model using the REST API server, through local computer or onto device
 - ▶ Monitor the health and potential drifts of your system!
- ▶ This concludes the universal workflow of machine learning—that's many things to keep in mind. It takes time and experience to become an expert, but don't worry. You're already a lot wiser than you were a few weeks ago. You are now familiar with the big picture—the entire spectrum of what data science projects entail. Always keep in mind the big picture!

References

- [1] [Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 3rd Edition Chapter 1 and 19](#)
- [2] [Deep learning with Python, 2nd Edition Chapter 9](#)
- [3] <https://madewithml.com/courses/mlops/monitoring/>
- [4] <https://fullstackdeeplearning.com/spring2021/lecture-11/#ii-model-monitoring>
- [5] <https://fullstackdeeplearning.com/course/2022/lecture-5-deployment/>
- [6] <https://evidentlyai.com/blog/tutorial-1-model-analytics-in-production>



Appendix

Resources and libraries

▶ Lectures

- ▶ <https://github.com/microsoft/ML-For-Beginners/blob/main/3-Web-App/1-Web-App/README.md>
- ▶ <https://github.com/microsoft/Data-Science-For-Beginners/blob/main/5-Data-Science-In-Cloud/17-Introduction/README.md>
- ▶ <https://github.com/microsoft/Data-Science-For-Beginners/blob/main/6-Data-Science-In-Wild/20-Real-World-Examples/README.md>

▶ Continual learning

- ▶ <https://fullstackdeeplearning.com/course/2022/lecture-6-continual-learning/>

▶ On-line learning

- ▶ <https://github.com/online-ml/river>

Resources and libraries

► Deployment

- [Flask](#)
- [FastAPI](#)
- [Tensorflow serving](#)
- <https://pytorch.org/serve/>
- [TensorFlow.js](#)
- [TensorFlow Lite](#)
- <https://pytorch.org/mobile/home/>
- [Streamlit](#)
- [Gradio](#)
- <https://github.com/voila-dashboards/voila>
- <https://github.com/datapane/datapane>
- <https://github.com/roboflow/inference>

Resources and libraries

▶ Container

- ▶ <https://github.com/bentoml/BentoML>
- ▶ <https://github.com/replicate/cog>
- ▶ <https://github.com/basetenlabs/truss>

▶ Optimization before deployment

- ▶ https://www.tensorflow.org/model_optimization
- ▶ <https://pytorch.org/blog/introduction-to-quantization-on-pytorch/>
- ▶ <https://huggingface.co/docs/optimum/index>

▶ Testing

- ▶ <https://madewithml.com/courses/mlops/testing/>

Resources and libraries

► Monitoring and testing

- <https://github.com/evidentlyai/evidently>
- <https://github.com/online-ml/river>
- <https://github.com/whylabs/whylogs>
- <https://github.com/deepchecks/deepchecks>
- <https://github.com/SeldonIO/alibi-detect>
- <https://greatexpectations.io/>

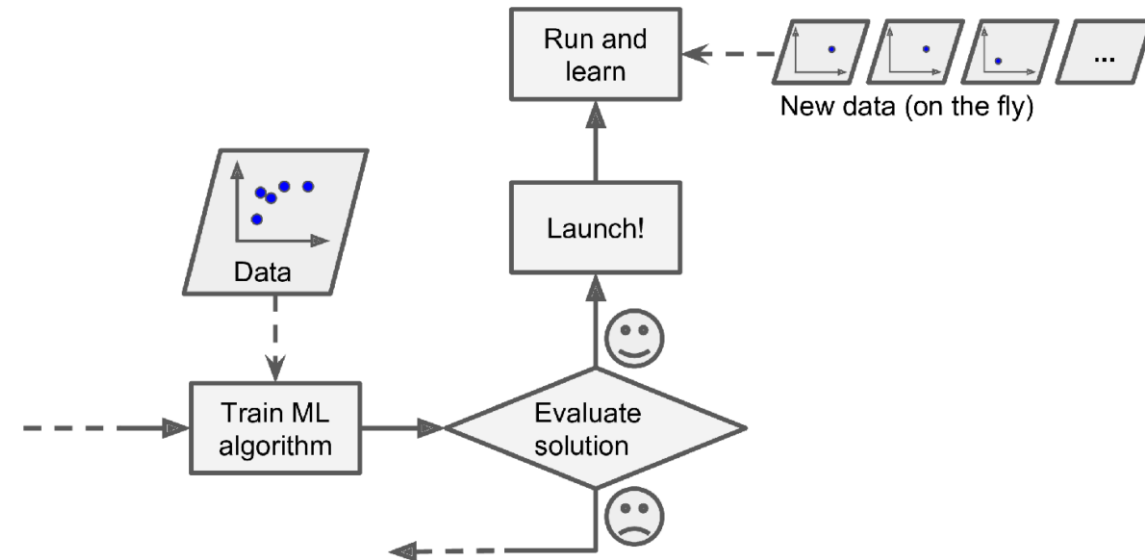
► Data shift

- <https://dcai.csail.mit.edu/2024/imbalance-outliers-shift/>
- <https://direct.mit.edu/books/edited-volume/3841/Dataset-Shift-in-Machine-Learning>
- https://d2l.ai/chapter_linear-classification/environment-and-distribution-shift.html

Determine how to serve

► Online Learning

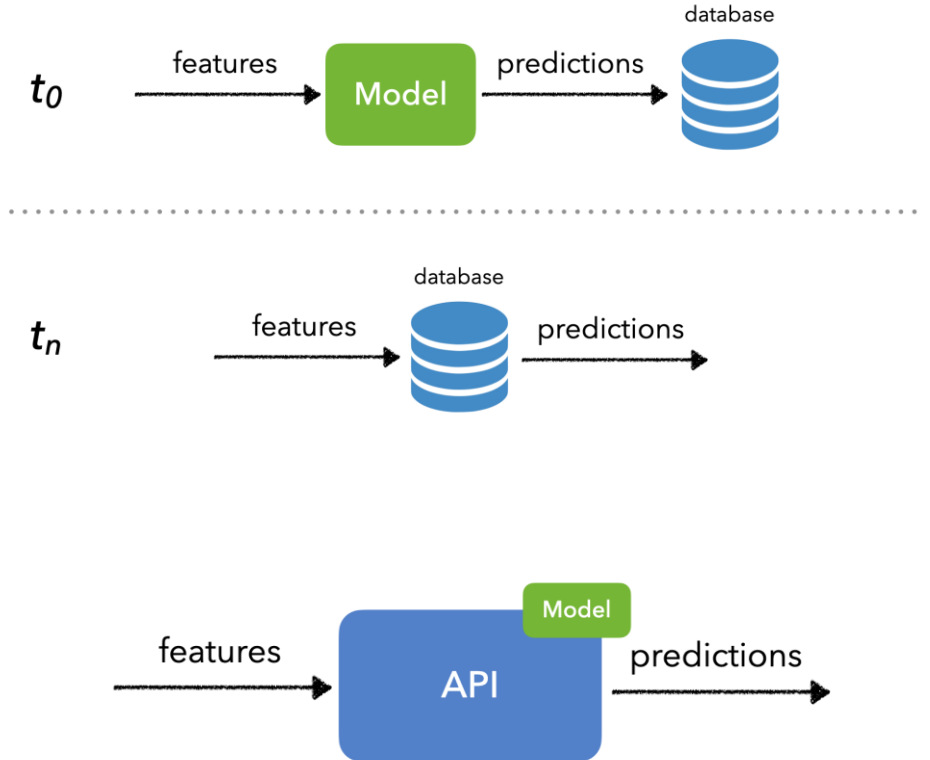
- In *online learning*, you train the system incrementally by feeding it data sequentially, either individually or *mini-batches*. Each learning step is fast and cheap, so the system can learn about new data on the fly
- It can receive data as a continuous flow and need to adapt to change rapidly
- One important parameter of online learning systems is how fast they should adapt to changing data: this is called the *learning rate*



Determine how to serve

► Online Learning

- If bad data is fed to the system, the system's performance will gradually decline. For example, bad data could come from a malfunctioning sensor on a robot, or from someone spamming a search engine to try to rank high in search results. To reduce this risk, you need to monitor your system closely and promptly switch learning off
- Similarly, you need to determine whether your system provide stream or batch serving

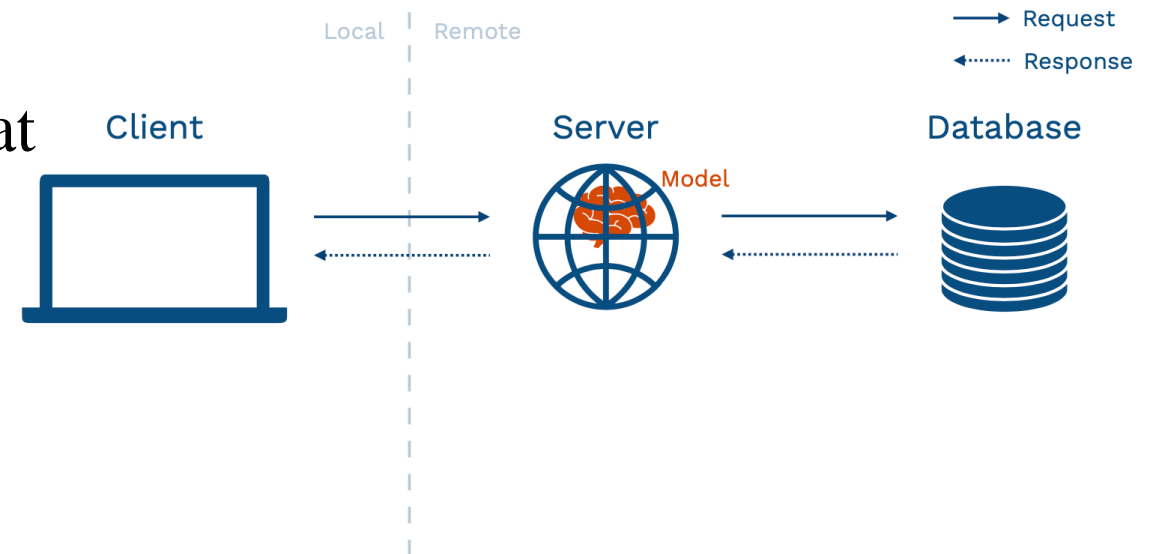


Build a Prototype To Interact With

- ▶ Here are some best practices for prototype deployment:
 1. **Have a basic UI:** The goal at this stage is to play around with the model and collect feedback from other folks
 2. **Put it behind a web URL:** An URL is easier to share. Furthermore, you will start thinking about the tradeoffs you'll be making when dealing with more complex deployment schemes
 3. **Do not stress it too much:** You should not take more than a day to build a prototype
- ▶ A model prototype won't be your end solution to deploy
 - ▶ Firstly, a prototype has limited frontend flexibility, so eventually, you want to be able to build a fully custom UI for the model
 - ▶ Secondly, a prototype does not scale to many concurrent requests. Once you start having users, you'll hit the scaling limits quickly

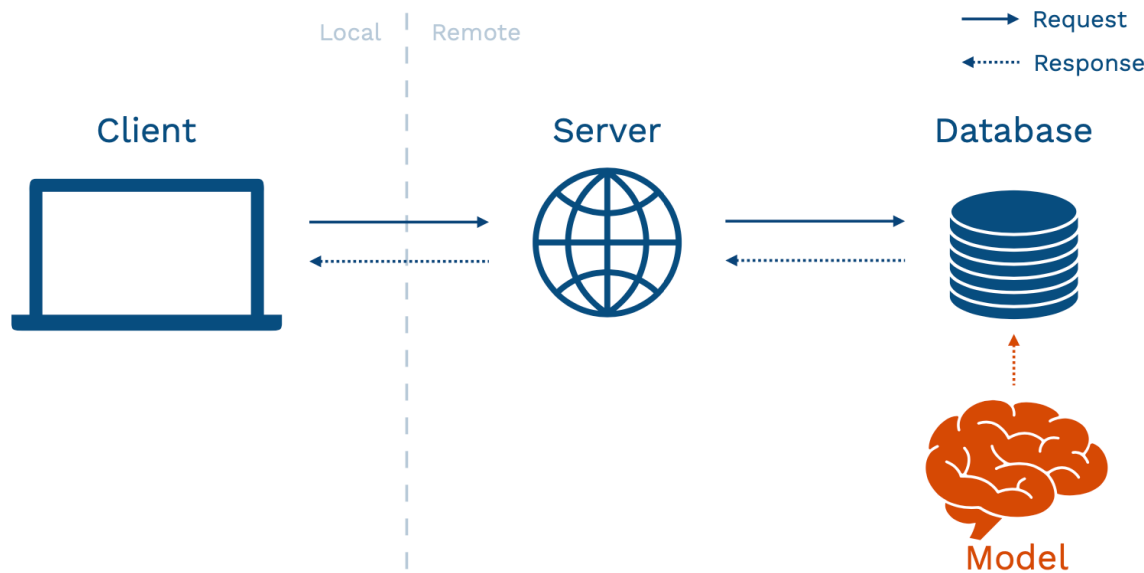
Build a Prototype To Interact With

1. Your web server may be written in a different language
2. Models may change more frequently than server code
 - ▶ If you have a well-established application and a nascent model, you do not want to redeploy the entire application every time that you make an update to the model
3. Large models can eat into the resources for your web server. That might affect the user experience for people using that web server, even if they are not interacting with the model
4. Your model and application may have different scaling properties



Separate Your Model From Your UI

- ▶ The first pattern to pull your model from your UI is called **batch prediction**
 - ▶ You get new data in and run your model on each data point. Then, you save the results of each model inference into a database
 - ▶ For example, if there are not a lot of potential inputs to the model, you can re-run your model on some frequency. You can have reasonably fresh predictions to return to those users that are stored in your database. Examples of these problems include the early stages of building recommender systems and internal-facing tools like marketing automation



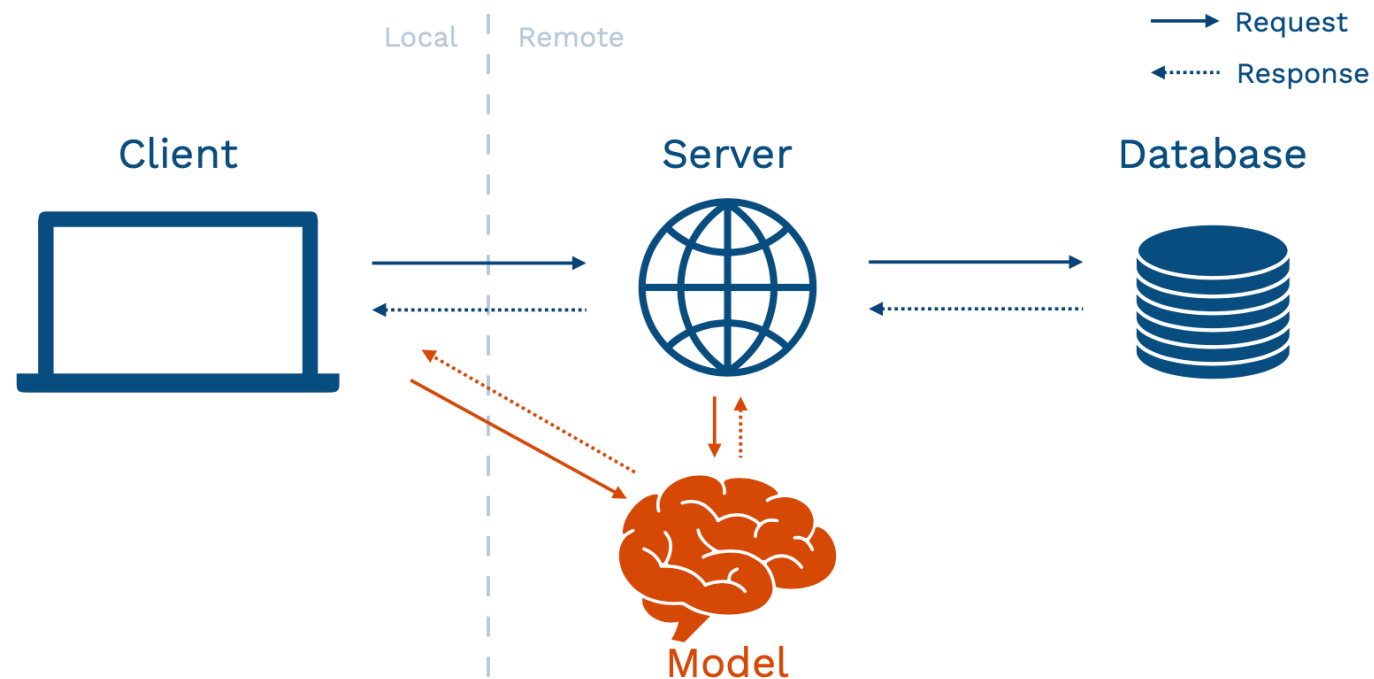
Separate Your Model From Your UI

► Pros and Cons

- ✓ Batch prediction scales easily because databases have been engineered for such a purpose
- ✓ Even though it looks like a simple pattern, it has been used in production by large-scale production systems for years
- ✓ It is fast to retrieve the prediction since the database is designed for the end application to interact with
- ✗ Batch prediction doesn't scale to complex input types. If the universe of inputs is too large to enumerate every single time you need to update your predictions, this won't work
- ✗ Users won't be getting the most up-to-date predictions from your model. If the feature that goes into your model changes every hour, minute, but you only run your batch prediction job every day, the predictions your users see might be slightly stale.
- ✗ Models frequently become "stale." If your batch jobs fail for some reason, it can be hard to detect these problems

Model-as-Service

- ▶ We run the model online as its own service. The service is going to interact with the backend or the client itself by making requests to the model service and receiving responses back

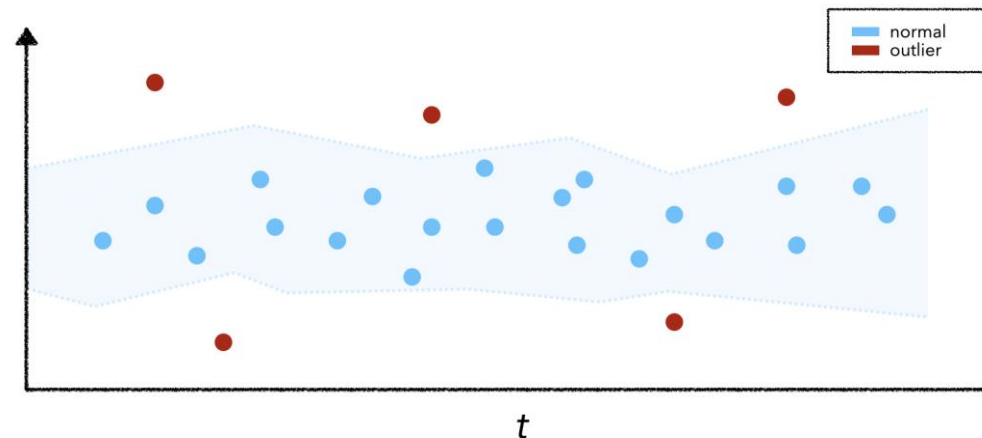


Model-as-Service

- ▶ The pros of this pattern are:
 - ✓ **Dependability** - model bugs are less likely to crash the web application
 - ✓ **Scalability** - you can choose optimal hardware for the model and scale it appropriately
 - ✓ **Flexibility** - you can easily reuse a model across multiple applications
 - ✗ Since this is a separate service, you add a network call when your server or client interacts with the model. That can **add latency** to your application
 - ✗ It also **adds infrastructural complexity** because you are on the hook for hosting and managing a separate service
- ▶ **The model-as-service pattern is still a sweet spot for most ML-powered products** since you really need to be able to scale independently of the application in most use cases

Outlier

- ▶ With drift, we're comparing a window of production data with reference data as opposed to looking at any one specific data point. While each individual point may not be an anomaly or outlier, the group of points may cause a drift
- ▶ It's not very easy to detect outliers because it's hard to constitute the criteria for an outlier. Therefore the outlier detection task is typically unsupervised and requires a streaming algorithm to identify potential outliers
- ▶ Typically, outlier detection algorithms fit to the training set to understand what normal data looks like and then we can use a threshold to predict outliers



Data drift in Evidently

- ▶ For small data with ≤ 1000 observations in the reference dataset:
 - ▶ For numerical features ($n_unique > 5$): two-sample Kolmogorov-Smirnov test
 - ▶ For categorical features or numerical features with $n_unique \leq 5$: chi-squared test
 - ▶ For binary categorical features ($n_unique \leq 2$), use the proportion difference test for independent samples based on Z-score
- ▶ For larger data with > 1000 observations in the reference dataset:
 - ▶ For numerical features ($n_unique > 5$): Wasserstein Distance
 - ▶ For categorical features or numerical with $n_unique \leq 5$: Jensen–Shannon divergence