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.

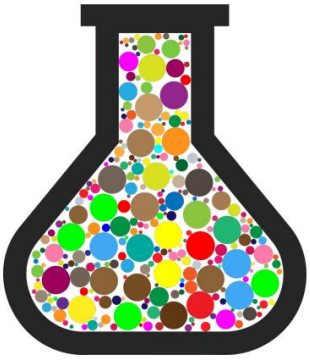
Understanding machine learning and deep learning concepts is essential, but if you're looking to build an effective AI career, you need production engineering capabilities as well. Machine learning engineering for production combines the foundational concepts of machine learning with the functional expertise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

Week 3: Model Management and Delivery

Contents

Why experiment tracking?

- ML projects have far more branching and experimentation
- Debugging in ML is difficult and time consuming
- Small changes can lead to drastic changes in a model's performance and resource requirements
- Running experiments can be time consuming and expensive



- In many ways, machine learning can be considered an experimental science since experimenting and analyzing results is at the heart of ML development.
- We need rigorous processes and reproducible results, which has created a need for experiment tracking. We'll discuss that now.
- **Experiments are fundamental to data science and machine learning**.
- ML in practice is more of an experimental science than a theoretical one.
- Tracking the results of experiments, especially in production environments, is critical to being able to make progress towards your goals.
- Debugging in ML is often fundamentally different than debugging in software engineering because it's **often about a model not converging or not generalizing, instead of some functional error like a segfault.**
- Keeping a clear record of the changes of the model and data over time can be a big help when you're trying to **hunt down the source of the problem.**
- Even small changes like changing the width of a layer or the learning rate can make a big difference in both the model's performance and the resources required to train the model.
- Again, tracking even small changes is important.
- Don't forget that running experiments, which means training your model over and over again, can be very time-consuming and expensive.
- This is especially true for large models and large datasets, especially when you're using expensive accelerators like GPUs to speed things up. Making the maximum use of each experiment is important.

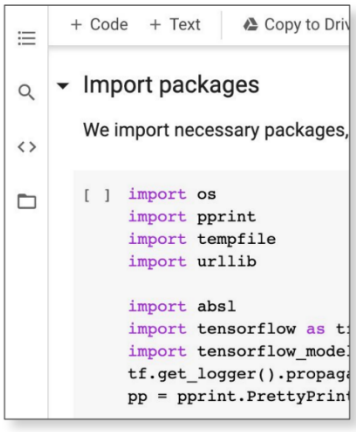
What does it mean to track experiments?

- Enable you to duplicate a result
- Enable you to meaningfully compare experiments
- Manage code/data versions, hyperparameters, environment, metrics
- Organize them in a meaningful way
- Make them available to access and collaborate on within your organization

- Let's step back and think for a minute about what it means to track experiments.
- First, you want to keep track of all the things that you need in order to **duplicate** a result.
- Some of us have had the unfortunate experience of getting a good result and then making a few changes that were maybe not well tracked and then finding it hard to get back to the setup that produced that good result.
- Another important goal is being able to meaningfully **compare** results. This helps guide you when you're trying to decide what to do next in your experiment.
- Without good tracking, it can be hard to make comparisons of more than a small number of experiments.
- It's important to track and manage all of the things that go into each of your experiments, including your **code**, your **hyperparameters**, and the **execution environment**, which includes things like the **versions** of the libraries that you're using, and the **metrics** that you're measuring.
- Of course, it helps to organize them in a meaningful way.
- Many people start out taking free-form notes, which is fine for a small number of simple experiments, but quickly becomes a mess.
- Finally, because you're probably working in a team with other people, good tracking helps when you want to share your results with your team.
- That usually means that as a team, you need to share common tooling and be consistent.
- **At a basic level, especially when you're just starting out on a project, most or all of your experiments might be in a notebook.**
- Notebooks are powerful and friendly tools for ML data and model development and allow for a nice iterative development process, including in-line visualizations.
- However, notebook code is usually not directly promoted to production as it is often not well structured.
- One of the reasons it's not usually promoted is that notebooks aren't just product code.
- They often contain notebook **magics**, special **annotations** that only work in the notebook environment, code to **check the value** of things and code to **generate visualizations** which you rarely want to do in a production workflow.

Simple Experiments with Notebooks

- Notebooks are great tools
- Notebook code is usually not promoted to production
- Tools for managing notebook code
  - nbconvert (.ipynb -> .py conversion)
  - nbdime (diffing)
  - jupytext (conversion+versioning)
  - neptune-notebooks (versioning+diffing+sharing)



- When you're experimenting with notebooks, you want to make sure to track those experiments and there are some tools that help with that.
- These include **nbconvert** (which among other things, can be used to extract the Python from a notebook), **nbdime** (which enables diffing and merging of Jupyter notebooks), **jupytext** (which converts and synchronizes pairs of notebooks with a matching Python file and much more), and **neptune-notebooks** (which helps with versioning, diffing and sharing notebooks).

Smoke testing for Notebooks

```
jupyter nbconvert --to script train_model.ipynb python train_model.py;
python train_model.py
```

- For example, to make sure that the extracted Python code from your notebook will actually run, you can use commands like this to extract the code from your notebook and then try running it.
- If it fails, then there were things happening in your notebook that your code depended on, like perhaps notebook magics.

Not Just One Big File

- Modular code, not monolithic
- Collections of interdependent and versioned files
- Directory hierarchies or monorepos
- Code repositories and commits

- But as you move from simple, small experiments into production level experiments, you'll quickly outgrow the pattern of putting everything in a notebook.
- You should plan to **write modular code, not monolithic code,** and the earlier in the process, the better
- Because you'll tend to do many core parts of your work over and over again, you'll develop reusable modules that will become high level tools, often specific to your environment, infrastructure, and team.
- Those will save you a lot of time and be much more robust and maintainable.
- They'll also make it easier to understand and reproduce experiments.
- **The simplest form** of these is just a **directory hierarchy** especially if your whole team is working in a monorepo.
- But in more advanced and distributed workflow, **code repositories and versioning with commits** are powerful and widely available tools for managing large projects, including experiments.
- In these cases, you probably want to **keep experiments separate** if you're using a shared monorepo with your team so that your commits don't version the rest of the team's repo.

Tracking Runtime Parameters

Config files

```
data:
    train_path: '/path/to/my/train.csv'
    valid_path: '/path/to/my/valid.csv'

model:
    objective: 'binary'
    metric: 'auc'
    learning_rate: 0.1
    num_boost_round: 200
    num_leaves: 60
    feature_fraction: 0.2
```

Command line

```
python train_evaluate.py \
    --train_path '/path/to/my/train.csv' \
    --valid_path '/path/to/my/valid.csv' \
    -- objective 'binary' \
    -- metric 'auc' \
    -- learning_rate 0.1 \
    -- num_boost_round 200 \
    -- num_leaves 60 \
    -- feature_fraction 0.2
```

- As you perform experiments, you're often changing runtime parameters including your model's hyperparameters.
- It's important to include the values of those parameters in your experiment tracking and how you set them will determine how you do that.
- **A simple and robust method is to use configuration files** and change those values by editing those files. The files can be **versioned** along with your code for tracking.
- Another option is to **set your parameters on the command line**. But this requires additional code to save those parameter values and associate them with your experiment.
- This means including code along with your experiment to save those values in a data store somewhere.
- This is an additional burden, but it also makes those values easily available for analysis and visualization rather than having a parse them out of a specific commit of a config file.
- Of course, if you do use config files, you can also include the code along with your experiment to save those values in a data store somewhere, which gives you the best of both worlds.

Log Runtime Parameters

```
parser = argparse.ArgumentParser()
parser.add_argument('--number_trees')
parser.add_argument('--learning_rate')
args = parser.parse_args()

neptune.create_experiment(params=vars(args))
...
# experiment logic
...
```

- This is an example of what the code to save your runtime parameter values might look like in this case, where you're setting your runtime parameters on the command line and this example uses the Neptune AI API

Tools for Experiment Tracking

Data Versioning

- Data reflects the world, and the world changes
- Experimental changes include changes in data
- Tracking, understanding, comparing, and duplicating experiments includes data

- Now let's look at some of the tooling that's available to help you implement experiment tracking.
- Along with your code and your runtime parameters, you also **need to version your data**.
- Remember that your data reflects a snapshot of the world at the time when the data was gathered and of course the world changes.
- If you're adding new data or purging old data or cleaning up your data, it will change the results of your experiments.
- So just like when you make changes in your code, or your model, or your hyperparameters, you need to track versions of your data.
- You might also change your feature vector as you experiment to add delete or change features and that needs to be versioned.
- So if you're going to be able to track, understand, compare and duplicate your experimental results you need to version your data.

Tools for Data Versioning

- Neptune
- Pachyderm
- Delta Lake
- Git LFS
- DoIt
- lakeFS
- DVC
- ML-Metadata

- Fortunately there are **good tools for data versioning**.
- These include **Neptune** which includes data versioning, experiment tracking and model registry.
- **Pachyderm** which lets you continuously update data in the master branch of your repo while experimenting with specific data commits in a separate branch or branches.
- **Delta Lake** which runs on top of your existing data lake and provides data versioning including rollbacks and full historical audit trails.
- **Git LFs** which is an extension to git, and replaces large files such as audio samples, videos, datasets and graphics with text pointers inside Git.
- **DoIt** which is a SQL database that you can fork, clone, branch, merge, push and pull just like a Git repository.
- **LakeFS**, which is an open source platform that provides a Git like branching and committing model the scales to petabytes of data.
- **DVC**, which is an open source version control system for machine learning projects and runs on top of Git.
- And **ML-Metadata**, which is a library for recording and retrieving metadata associated with ML developer and data scientist workflows, including datasets. ML MD or ML-Metadata is an integral part of TFX, but it's designed so that it can also be used independently.

Experiment tracking to compare results

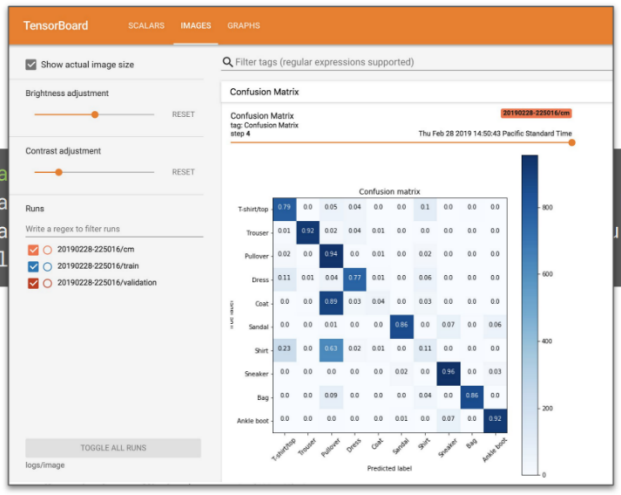
Name (50 visualized)	Tags	acc	Sweep	optimizer	epoch	batch_size	n_train	n_valid	n_conv_lay	loss	GPU
batch 64 4 GPU	4GPU b_64_c	0.4305	-	rmsprop	49	64	5000	800	1	1.632	-
batch 64 (V2, 5K train	2GPU b_64_c	0.4343	-	rmsprop	49	64	5000	800	1	1.63	-
50K examples (b 64)		0.4042	-	rmsprop	49	64	50000	8000	1	1.76	-
batch 32 4 GPU	4GPU b_32_c	0.4032	-	rmsprop	49	32	5000	800	1	1.714	-
batch 64 1 GPU	1GPU GCP	0.4465	-	rmsprop	49	64	5000	800	5	1.615	1
batch 128 (5K train)	2GPU b_128	0.4181	-	rmsprop	49	128	5000	800	1	1.658	-
batch 256 4 GPU	4GPU keras	0.3882	-	rmsprop	49	256	5000	800	1	1.751	-

- The typical ML workflow involves running lots of experiments.
- Most developers find that looking at results in the context of other results is much more meaningful than looking at a single experiment alone.
- But as you can see, **looking across lots of experiments at once can be a bit confusing at first.**

- As you gain experience with the tools, you'll get more comfortable and it will be easier to focus on what you're looking for.
- There are lots of inputs changing and lots of different possible outputs.
- Some runs will inevitably fail early.
- Different experimentation styles lead to different workflows, but it's a good idea to log every metric that you might care about.
- **Tag** experiments with a few consistent tags which are meaningful to you and add **notes**.
- Developing these habits can keep things much more organized later.

Example: Logging metrics using TensorBoard

```
logdir = "logs/ima
tensorboard_callba
cm_callback = kera
model.fit(... call
```




```
histogram_freq=1)
usion_matrix)
```

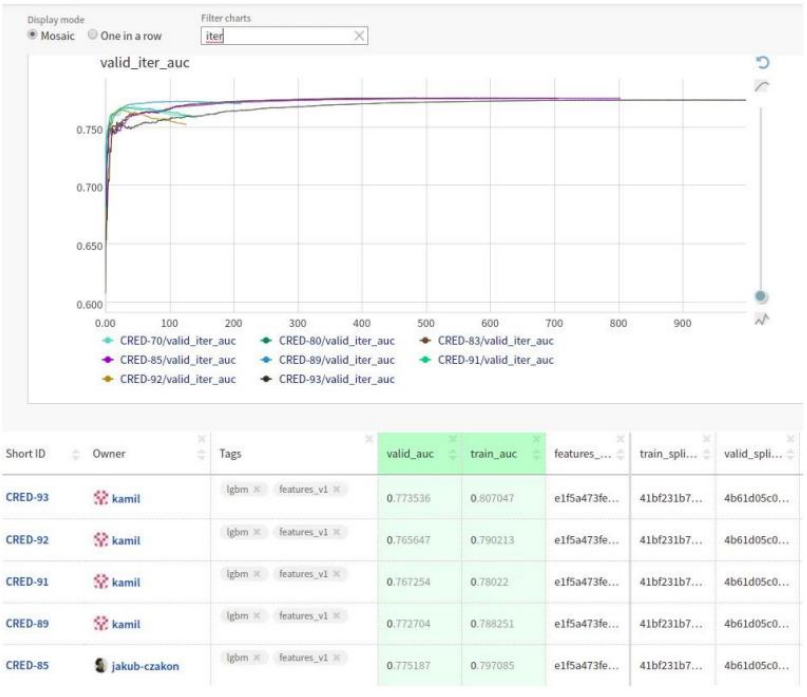
- TensorBoard is an amazing tool for **analyzing your training**, which makes it very useful for understanding your experiments.
- For example, you can use a TensorBoard callback to log metrics and logs the confusion matrix at the end of every epoch.
- When you display the results, you get a clear view of how your model is doing, in this case, by examining a confusion matrix.
- By default the dashboard shows the image summary for the last log step or epoch.
- You can use the slider to view earlier confusion matrices.
- Notice how the matrix changes significantly as training progresses with darker squares, coalescing along the diagonal and the rest of the matrix tending towards zero and white. This means that your classifier is improving as training progresses.
- The ability to visualize the results as the model is training and not just when the training is complete can also give you insights into your experiments.

Organizing model development

- Search through & visualize all experiments
- Organize into something digestible
- Make data shareable and accessible
- Tag and add notes that will be meaningful to your team

- As you experiment, you'll be looking at each result as it becomes available and starting to compare results.
- Organizing your experimental results from the start is important to help you understand your own work when you revisit it later and help your team understand it as well.
- You want to make sure that it's easy to share and accessible, so that you and your team can collaborate especially when working on larger projects.
- **Tagging** each experiment and adding your notes will help both you and your team and help **avoid having to rerun experiments more than once.**

Tooling for Teams

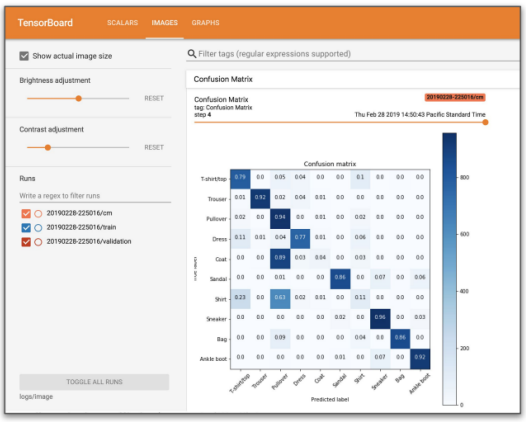


- **Tooling**, which enables sharing can really help.
- For example, if you can use the experiment management tool provided by Neptune AI to send a link that shares a comparison of the experiments, it makes it easy for you and your team to track and review progress, discuss problems and inspire new ideas.

Tooling for Teams

Vertex TensorBoard

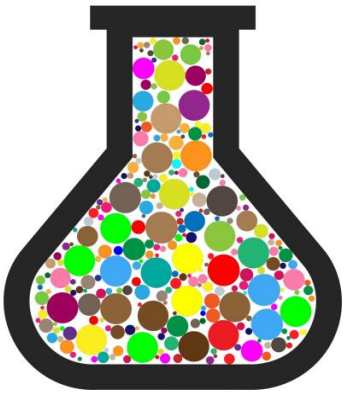
- Managed service with enterprise-grade security, privacy, and compliance
- Persistent, shareable link to your experiment dashboard
- Searchable list of all experiments in a project



- Vertex TensorBoard and similar cloud based tools can also offer significant advantages.
- First, like many infrastructure decisions, there are significant advantages to using a managed service, including security, privacy and compliance.
- But one of the most important features is having a **persistent shareable link** to your dashboards, which you can share with your team and not have to worry about setting it up and maintaining it.
- **Having a searchable list of all the experiments in a project can also be incredibly useful.**

Experiments are iterative in nature

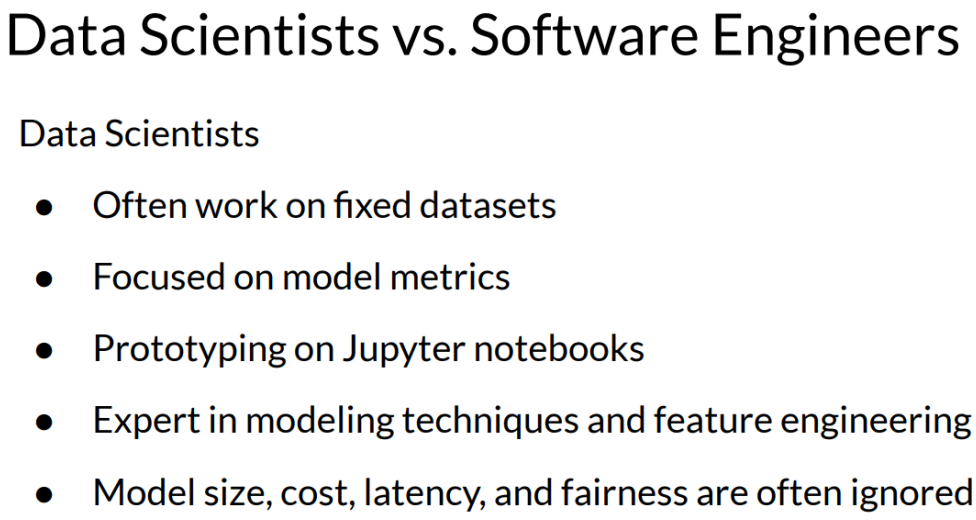
- Creative iterations for ML experimentation
- Define a baseline approach
- Develop, implement, and evaluate to get metrics
- Assess the results, and decide on next steps
- Latency, cost, fairness, etc.



- Tools like that are a big help and a huge improvement over spreadsheets and notes.
- However you can take your machine learning project to the next level with creative iterations.
- In every project, there is a phase where a business specification is created that usually includes a schedule, budget and the goals of the project.
- The goals are usually a set of KPIs, business metrics or if you are incredibly lucky, actual machine learning metrics.
- You and your team should choose what you think might be achievable as business goals that align with the project and start by defining a **baseline approach**.
- **Implement your baseline and evaluate to get a first set of metrics**.
- Often you'll learn a lot from those first baseline results. They may be close to meeting your goals, which tells you that this is likely to be an easy problem.

- Or your results may be so far off that you'll start to wonder about the **strength of the predictive signal in your data** and start considering more complex modelling approaches.
- And don't forget that although there is a tendency to focus on modeling metrics, and unfortunately, much of the tooling also has that focus.
- Since you're doing production ML, you also need to meet your business goals for latency, cost, fairness, privacy, GDPR and so forth.

Introduction to MLOps



- Almost everything that we've talked about and we'll talk about in these courses, can be considered MLOps in a broad sense.
- But now let's take a closer look at MLOps in a **more narrow sense** and develop an understanding of **different levels of maturity of MLOps processes and infrastructure**.
- First, let's understand the two key roles within a typical engineering team. Data scientist and software engineer.
- Thinking about these roles will help you to understand why production ML requires data scientists to evolve into **domain experts who can both develop predictive models and build production machine learning solutions**.
- You'll also learn how AI components are parts of larger systems and explore the challenges in engineering in AI enabled system.
- Thinking first about data scientists, especially those coming from a research or academic background, they often focus on fixed data sets and focus on optimizing model metrics such as accuracy while doing prototyping in notebooks.
- Their training usually makes them experts in modeling and feature engineering and model size, but cost, latency and fairness are often **not a central focus** of their work.

Data Scientists vs. Software Engineers

Software Engineers

- Build a product
- Concerned about cost, performance, stability, schedule
- Identify quality through customer satisfaction
- Must scale solution, handle large amounts of data
- Detect and handle error conditions, preferably automatically
- Consider requirements for security, safety, fairness
- Maintain, evolve, and extend the product over long periods

- Software engineers however, are much more focused on building products.
- So concerns such as cost and performance, stability, scalability, maintainability and schedule are much more important to them.
- They identify strongly with customer satisfaction and recognize infrastructure needs such as scalability.
- They have a strong focus on quality, testing and detecting and mitigating errors, and are keenly aware of the need for security, safety and fairness.
- They also however, tend to view their product as basically static with changes being primarily the result of bug fixes or new features.
- Changes in the data as the world around them changes, are **not** typically a primary concern when simply doing software development.

Growing Need for ML in Products and Services

- Large datasets
- Inexpensive on-demand compute resources
- Increasingly powerful accelerators for ML
- Rapid advances in many ML research fields (such as computer vision, natural language understanding, and recommendations systems)
- Businesses are investing in their data science teams and ML capabilities to develop predictive models that can deliver business value to their customers

- Increasingly, data science and ML are becoming core capabilities for solving complex real world problems, transforming industries and delivering value.
- Currently, the ingredients for applying ML have already been made available or accessible with large datasets, inexpensive on demand compute resources, and increasingly powerful accelerators for ML such as GPUs and TPUs on cloud platforms like AWS, Azure and Google Cloud.

- There have been rapid advances in ML research in computer vision, natural language understanding and recommendation systems where there is an increased demand for applying ML to offer new capabilities.
- With all that in mind, many businesses have already started investing in their data science teams and ML capabilities to develop predictive models that can deliver business value to their customers.
- All of this drives evolution of product focused engineering practices for ML, which is the basis for the development of MLOps.

Key problems affecting ML efforts today

We've been here before

- In the 90s, Software Engineering was siloed
- Weak version control, CI/CD didn't exist
- Software was slow to ship; now it ships in minutes
- Is that ML today?

Today's perspective

- Models blocked before deployment
- Slow to market
- Manual tracking
- No reproducibility or provenance
- Inefficient collaboration
- Unmonitored models

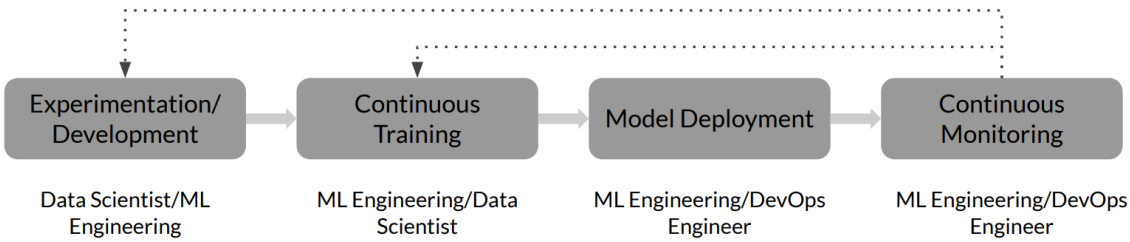
- There are a number of problems affecting AI efforts today, so there is an increasing focus on improving our machine learning workflows.
- But it turns out that **software engineering faced a similar situation in the recent past**.
- There's no denying that in the nineties, software engineering too was once siloed.
- Back then even basic version control was fairly weak compared to what is currently used today, and things that you take for granted like CI/CDs didn't even exist.
- This resulted in software taking a long time to ship.
- So thinking about what it was like back then, is that where ML is today in today's perspective for many teams, it can take months to deploy their models to production once they've trained them.
- In some cases being that slow to market means **missing opportunities**, or deploying models that are already **decayed**.
- Moreover, **traditional data science projects lack tracking** and then there are problems like manual tracking and models **not being reproducible**, and data lacking **provenance**, and a general lack of good tools and processes for **collaboration** between different teams.
- And once models do get deployed to production, there's often **no monitoring** for them.

Bridging ML and IT with MLOps

- **Continuous Integration (CI):** Testing and validating code, components, data, data schemas, and models
- **Continuous Delivery (CD):** Not only about deploying a single software package or a service, but a system which automatically deploys another service (model prediction service)
- **Continuous Training (CT):** A new process, unique to ML systems, that automatically retrains candidate models for testing and serving
- **Continuous Monitoring (CM):** Catching errors in production systems, and monitoring production inference data and model performance metrics tied to business outcomes

- DevOps is an engineering discipline which focuses on developing and managing software systems.
- It was developed over decades of experience and learning in the software development industry.
- Some of the potential benefits that it offers include **reducing development cycles**, **increasing deployment velocity and ensuring dependable releases of high quality software.**
- Like DevOps, MLOps is an ML engineering culture and practice that aims at **unifying ML system development or dev and ML system operation or ops.**
- Unlike DevOps, ML systems present **unique challenges** to core DevOps principles like continuous integration, which for ML means that you not only just **test** and **validate** code and components, but also do the same for **data schemas and models**.
- Continuous delivery on the other hand, isn't just about deploying a single piece of software or service, but as a system more precisely, an ML pipeline that deploys a model to a prediction service automatically.
- As ML emerges from research, disciplines like software engineering, DevOps and ML need to converge forming MLOps.
- So with that comes the need to employ a novel DevOps automation techniques dedicated for training and monitoring machine learning models.
- That includes **continuous training**, a new property that is **unique to ML systems**, which automatically re-trains models for both testing and serving.
- And once you have models in production, it's important to catch errors and monitor inference data and performance metrics with continuous monitoring.

ML Solution Lifecycle



- Now, let's consider the major phases in the life cycle of an ML solution.

- Usually a data scientist or an ML engineer start by shaping data and developing an ML model and continue by experimenting until you get results which meet your goals.
- After that, you typically go ahead and set up pipelines for continuous training unless you already use the pipeline structure for your experimenting in model development, which I would encourage you to consider.
- Then you turn to model deployment, which involves more of the operations and infrastructure aspects of your production environment and processes.
- And then continuous monitoring of your model systems and data from your incoming requests.
- The data from those incoming requests will become the basis for further experimentation and continuous training.
- So, as you go from continuous training to model deployment, the tasks evolve into something that traditionally a DevOps engineer would be responsible for.
- That means that you need a DevOps engineer who understands ML deployment and monitoring.
- And now let's move on to a new practice for collaboration and communication between data scientists and operation professionals which is known as MLOps.

Standardizing ML processes with MLOps

- ML Lifecycle Management
- Model Versioning & Iteration
- Model Monitoring and Management
- Model Governance
- Model Security
- Model Discovery

- MLOps provides capabilities that will help you build, deploy and manage machine learning models that are critical for ensuring the integrity of business processes.
- It also **provides a consistent and reliable means to move models from development to production by managing the ML Lifecycle**.
- Models generally need to be iterated and versioned. To deal with an emerging set of requirements, the models change based on further training or real world data that's closer to the current reality.
- MLOps also includes creating versions of models as needed and maintaining model version history.
- And as the real world and its data continuously change, it's critical that you manage **model decay**.
- With MLOps, you can ensure that by monitoring and managing the model results continuously, you can make sure that accuracy, performance and other objectives and key requirements are acceptable.
- MLOps platforms also generally provide capabilities to audit compliance, access control, **governance** testing and validation and change and access logs.
- The logged information can include details related to access control like who is publishing models, why modifications are done and when models were deployed or used in production.

- You also need to secure your models from both attacks and unauthorized access.
- MLOps solutions can provide some functionality to protect models from being corrupted by infected data, being made on unavailable by denial of service contracts or being inappropriately accessed by unauthorized users.
- Once you've made sure your models are secure, trustable and good to go, it's often a good practice to establish a platform where they can be easily discovered by your team. MLOps can do that by providing **model catalogs for models produced as well as a searchable model marketplace**.
- These model discovery solutions will provide information to track the data origination, significance, model architecture and history and other metadata for a particular model.

MLOps Level 0

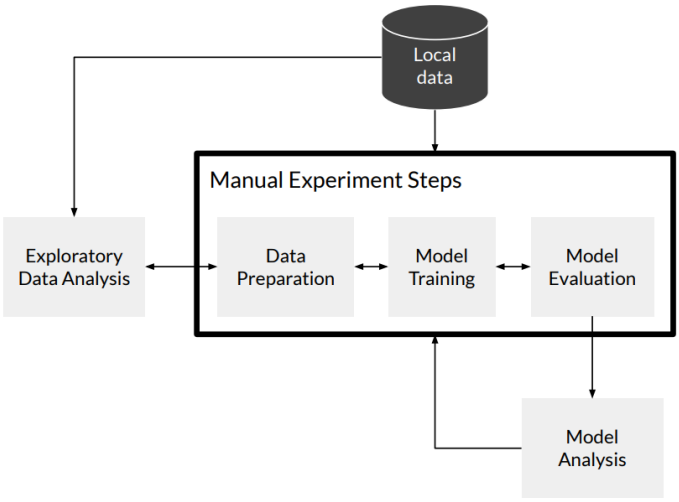
What defines an MLOps process' maturity?

- The level of **automation** of ML pipelines determines the maturity of the MLOps process
- As maturity increases, the available velocity for the training and deployment of new models also increases
- Goal is to automate training and deployment of ML models into the core software system, and provide monitoring

- Let's look now at how MLOps processes evolve and mature as teams become more and more sophisticated.
- Fundamentally, the level of automation of the data, modeling, deployment, and maintenance systems determines the maturity of the MLOps process.
- With increased maturity, the available velocity for the training and deployment of new models is also increased.
- The **objective of an MLOps team** is to automate the training and deployment of ML models into the core software system and provide robust and comprehensive monitoring.
- Ideally, this means automating the complete ML workflow with as little manual intervention as possible.
- **Triggers** for automated model training and deployment can be calendar events, messaging or monitoring events, as well as changes in data, model training code and application code or detected model decay.

MLOps level 0: Manual process

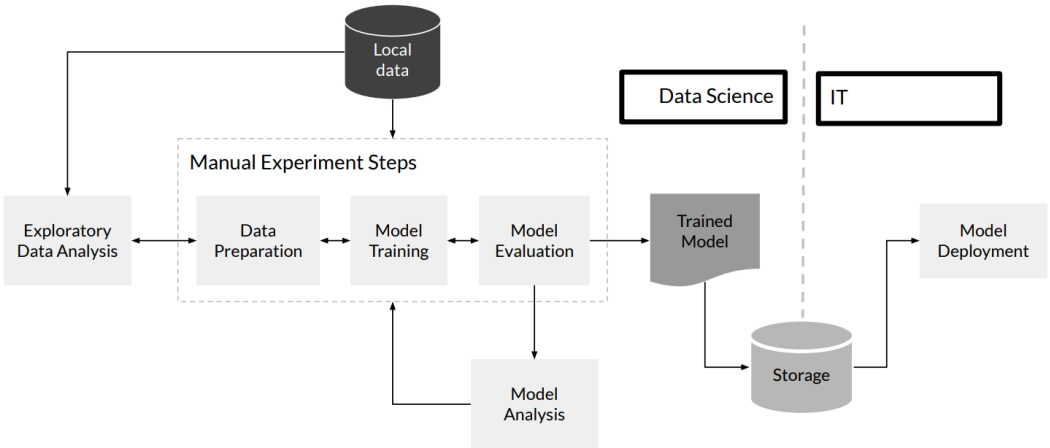
Manual, script-driven, interactive



- Many teams have data scientists and ML researchers who can build state-of-the-art models but their process for building and deploying ML models is entirely manual.
- This is considered the basic level of maturity or level 0.
- This is generally **script or notebook driven** for the most part, and every training step is manual, including data analysis, data preparation, model training, and validation.
- It requires manual execution of each step, and manual transition from one step to another.
- This process is usually driven by experimental code that is written and executed in notebooks by data scientists interactively until a workable model is produced.

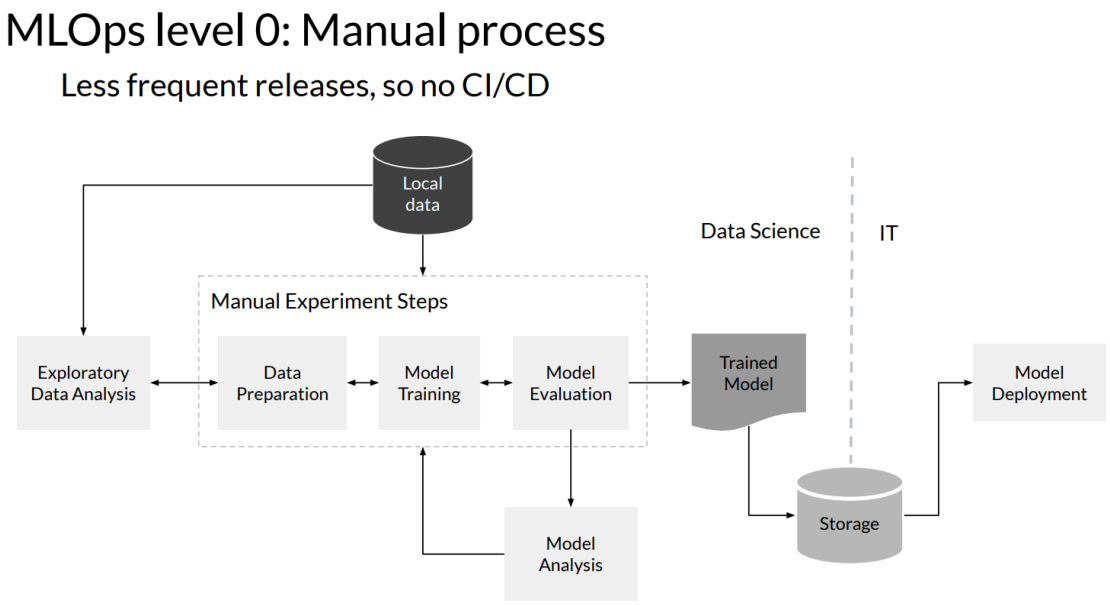
MLOps level 0: Manual process

Disconnection between ML and operations



- This creates a disconnect between the ML and operations teams.
- Among other things, this opens the door for potential training-serving skew.
- To better understand what's going on here, let's assume data scientists hand over a trained model to the engineering team to deploy it on their infrastructure per serving or batch prediction.

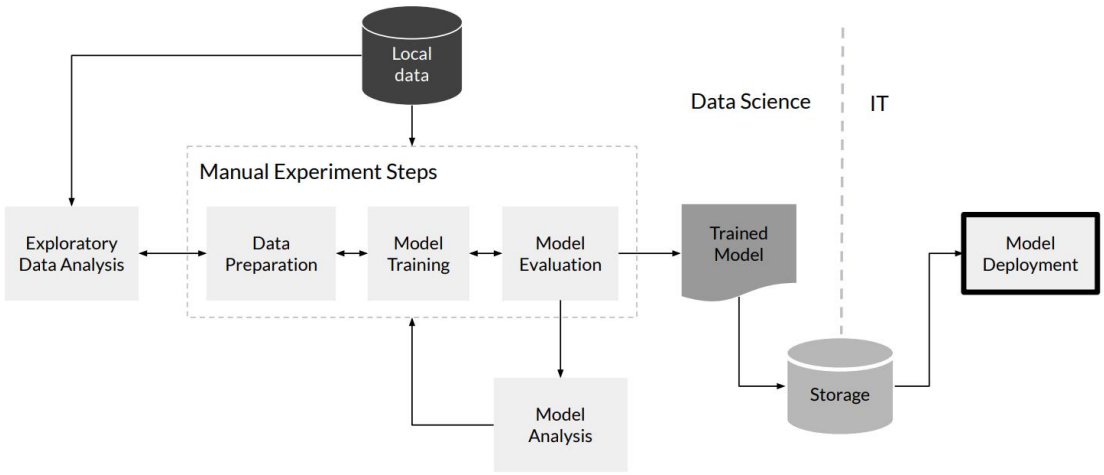
- This form of manual handoff could include putting the trained model in a file system somewhere, checking the model object into a code repository, or uploading it to a model registry.
- Then, engineers who deploy the model need to make the required input features available in production, potentially for low latency serving, which can lead to training serving skew.



- A level 0 process assumes that your data science team manages a few models that **don't change frequently** because of either changes in model the implementation, or retraining the model with new data, or both.
- A new model version is probably only deployed a couple of times a year.
- Because of fewer code changes, continuous integration or CI, and often even unit testing is totally ignored.
- Usually, testing the code is part of the notebooks or script execution.
- The scripts and notebooks that implement the experiment steps are often source controlled, and they produce **artifacts such as trained models, evaluation metrics, and visualizations**.
- Also, because there aren't many model versions that need to be deployed, continuous deployment or CDs isn't even considered.

How do you scale?

Deployment and lack of active performance monitoring



- **A level 0 process is concerned only with deploying the trained model as a prediction service**.
- For example, a microservice with a REST API, rather than deploying the entire ML system.
- Also, here you do not track or log the model predictions and actions which are required in order to detect model performance degradation and other model behavioral drifts.

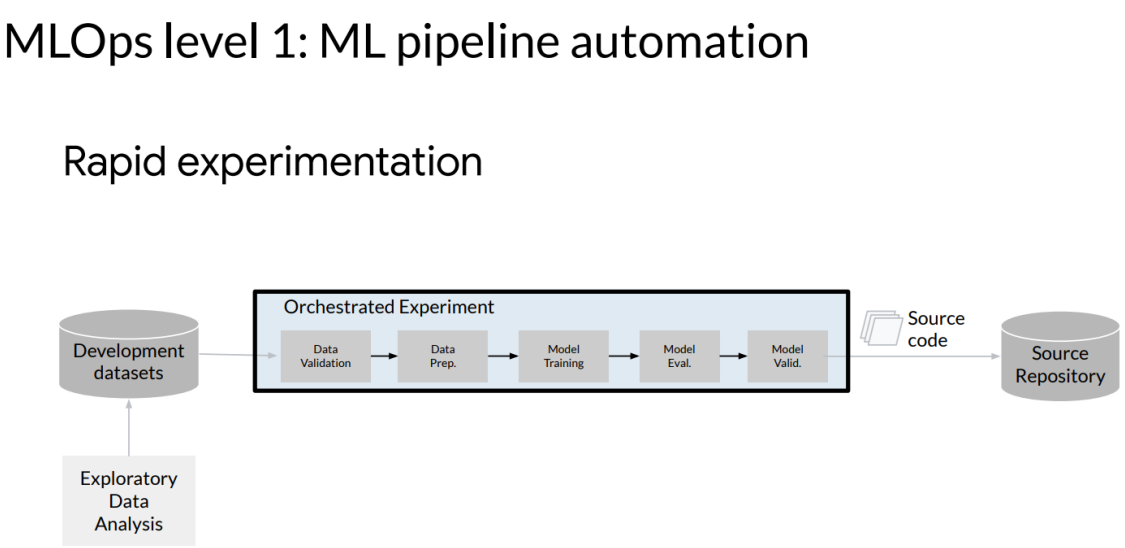
Challenges for MLOps level 0

- Need for actively monitoring the quality of your model in production
- Retraining your production models with new data
- Continuously experimenting with new implementations to improve the data and model

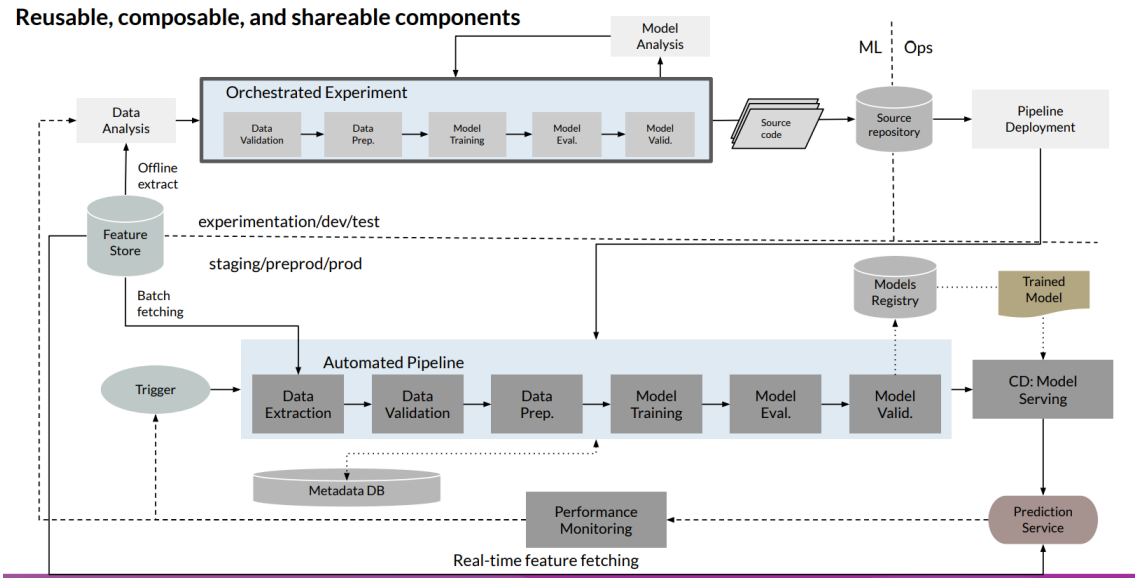
- MLOps level 0 is common in many businesses that are beginning to apply ML to their use cases.
- This **manual**, data science driven process might be sufficient when models are rarely changed or retrained.
- In practice, **models often break when they are deployed in the real world**.
- Models fail to adapt to changes in the dynamics of the environment or changes in the data that describes the environment.
- To address these challenges and to maintain your model's accuracy and production, you need, first, to **address the lack of active performance monitoring**.
- Active monitoring your model lets you detect performance degradation and model decay.
- It acts as a cue that it's time for new experimentation and retraining of the model on new data.
- Then there's the problem of continuously adapting your model to the latest trends.
- To overcome this, you need to **retrain your production models often with the most recent data** to capture the evolving and emerging patterns.
- For example, if your app recommends fashion products using ML, its recommendation should adapt to the latest trends and products.

- Of course, that requires you to have new data and to label it somehow, and at level 0, those are usually manual processes.

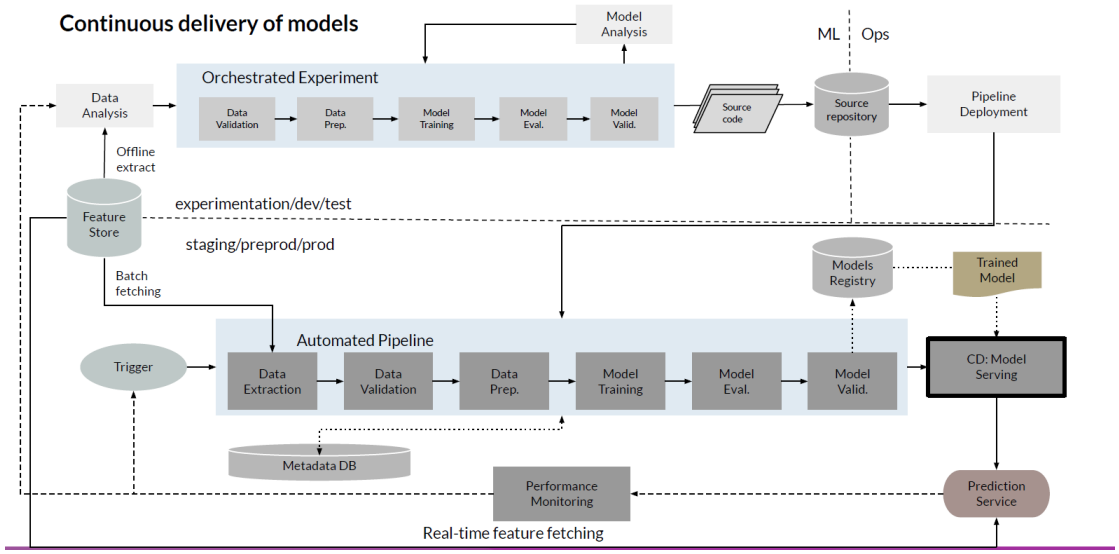
MLOps Levels 1 & 2



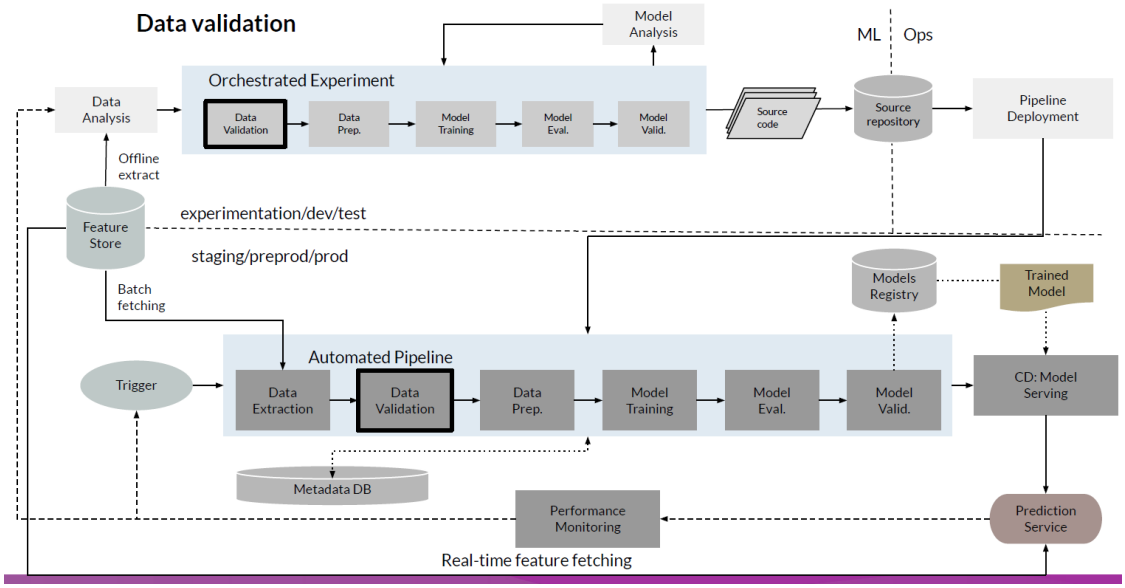
- Let's look at MLOps levels one and 2, which introduce pipeline automation.
- One of the key goals of level one is to perform **continuous training of the model**, by **automating the training pipeline**.
- This lets you achieve continuous delivery of trained models to your model prediction service.
- To automate the process of using new data to retrain models in production, you need to introduce **automated data and model validation** steps to the pipeline, as well as **pipeline triggers** and **metadata management**.
- There is a need to have repeatable training in your ML workflows, so let's look at some of the characteristics of pipeline automation.
- Since the steps of the experimentation are orchestrated, the transition between steps is automated.
- That enables you to rapidly iterate on your experiments and makes it **easier to move the whole pipeline to production.**



- Now, let's expand this out quite a bit to include the different environments – Dev, test, staging, pre-production, and production.
- Note that the architecture shown here is typical, but different teams will implement this differently depending on their needs and infrastructure choices.
- In this architecture, models are automatically retrained using fresh data based on live pipeline triggers, which I will discuss soon.
- The pipeline implementation that is used in the development or experimentation environment is also used in the pre-production and production environment, which is a key aspect of MLOps practice for **unifying** the DevOPS effort.
- To construct ML pipelines, components need to be reusable, composable, and potentially shareable across pipelines.
- Therefore, while exploratory data analysis code can still live in notebooks, the **source code for components must be modularized**.
- In addition, **components** should ideally be **containerized**.
- You do this in order to **decouple the execution environment from the custom code runtime**.
- It's also done to make code reproducible between development and production environments.
- This essentially isolates each component in the pipeline, making them their own version of the runtime environment, and have different languages and libraries.
- Note if the exploratory data analysis is done using production components and a production style pipeline, it greatly simplifies the transition of that code to production.

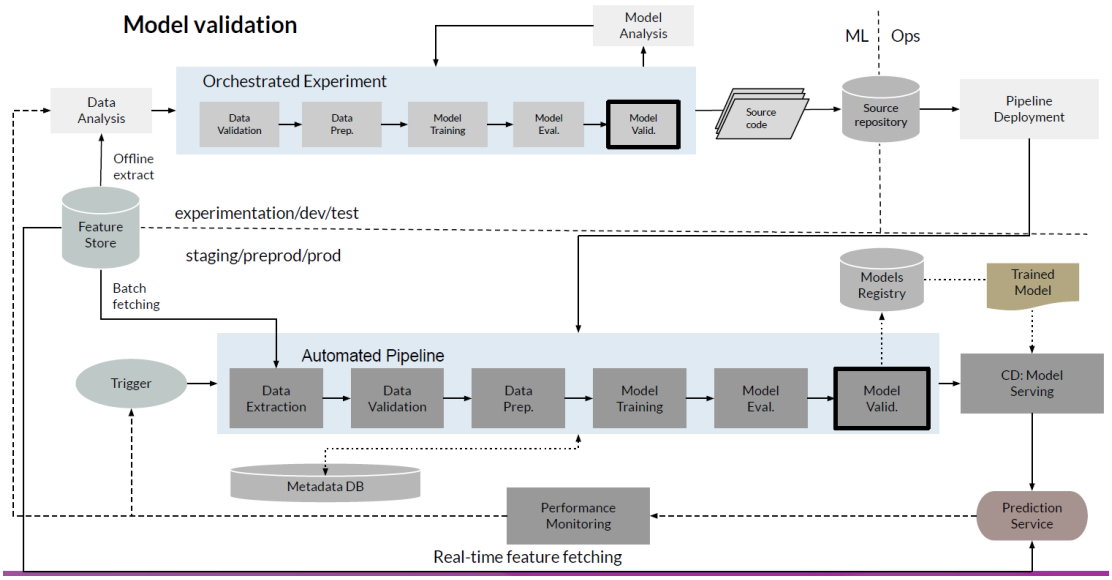


- An ML pipeline in production continuously delivers new models that are trained on new data to prediction services.
- Note that when I say **continuously**, I mean in an **automated** process in which new models might be delivered on a **schedule** or based on a **trigger**.
- The model deployment step is automated, which delivers the trained and validated model for use by a prediction service for online or batch predictions.
- In level zero, you simply deployed the trained model to production.
- But here, you **deploy a whole training pipeline**, which automatically and recurrently runs to serve the trained model as a prediction service.



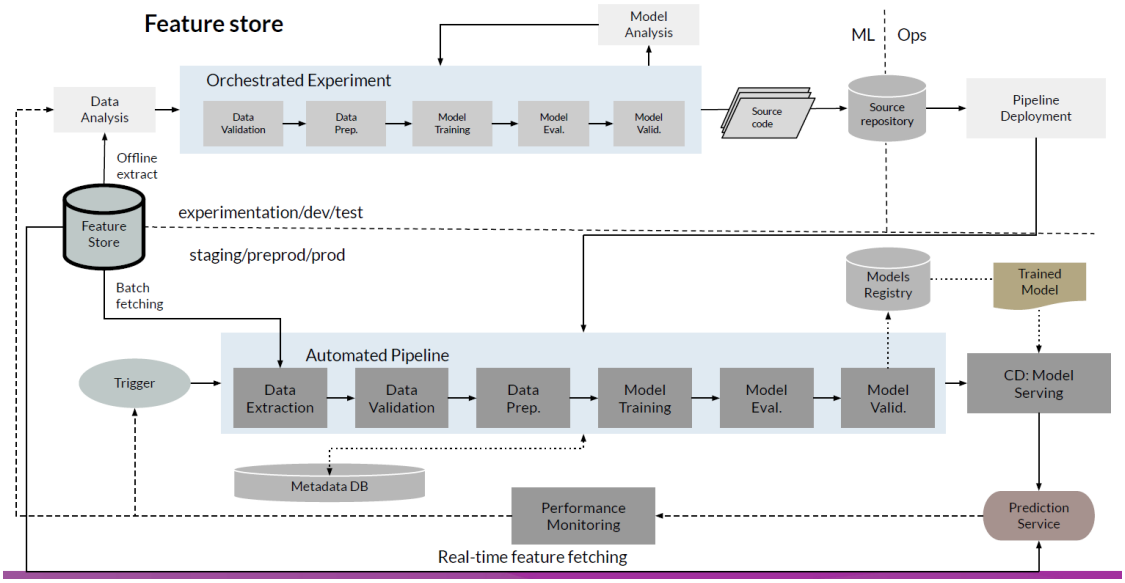
- When you deploy your pipeline to production, one or more of the triggers automatically executes the pipeline.
- The pipeline expects new live data to produce a new model version that is trained on new data.
- So **automated data validation and model validation steps** are required in the production pipeline.

- First, let's talk about why data validation is necessary before model training to decide whether you should retrain the model, or stop the execution of the pipeline.
- This decision is automatically made **only if the data is deemed valid**.
- For example, **data schema skews** are considered anomalies in the input data, which means that the downstream pipeline steps including data processing and model training, receives data that doesn't comply with the expected schema.
- In this case, you should stop the pipeline and raise a notification so that the team can investigate.
- The team might release a fix, or an update to the pipeline to handle these changes in the schema.
- **Scheme skews include receiving unexpected features, not receiving all the expected features or receiving features with unexpected values**.
- Then there are **data value skews** which are significant changes in the statistical properties of data, and you need to trigger retraining of the model to capture these changes.

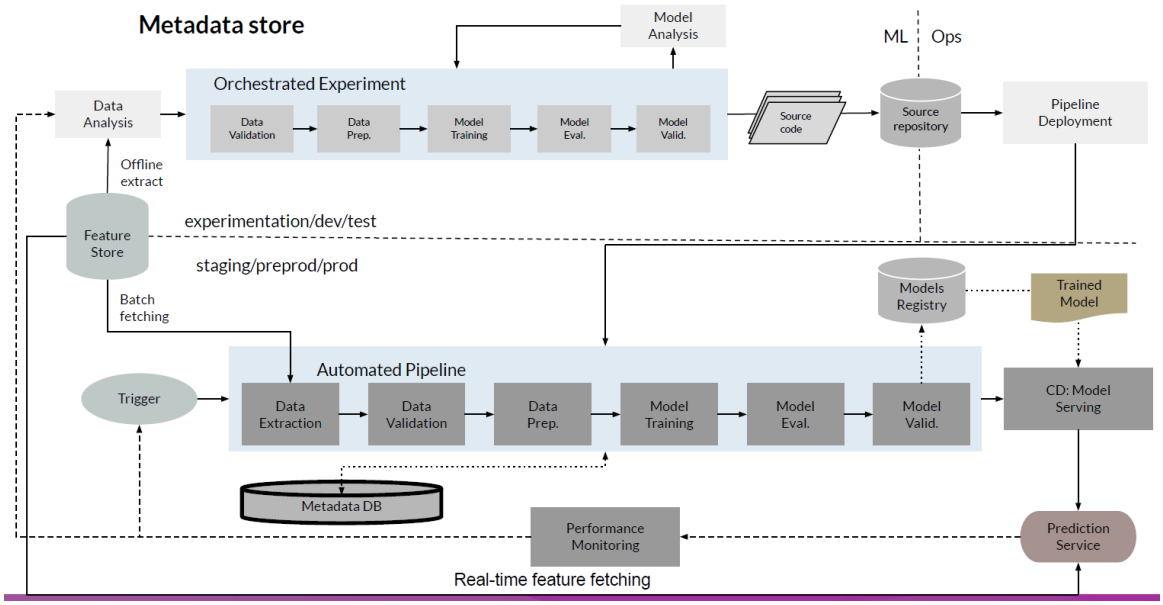


- Model validation is another step, which runs after you successfully train the model given the new data.
- Here, you evaluate and validate the model before it's promoted to production.
- Model validation makes sure that the new model performs better than the current one before promoting it to production. Also, Model Validation ensures that model performance is consistent on various segments of the data.
- This offline model validation step may involve first producing **evaluation metric values**, using the trained model on a test data set, to assess the model's predictive quality.
- Then, the next step would be to **compare** the evaluation metric values produced by your newly trained model to the current model.
- For example, the current production model, or a baseline model, or any other model which meets your business requirements.
- Here, you **make sure that the new model performs better** than the current model before promoting it to production.
- Also, you ensure the performance of the model is **consistent on various segments or slices of the data.**

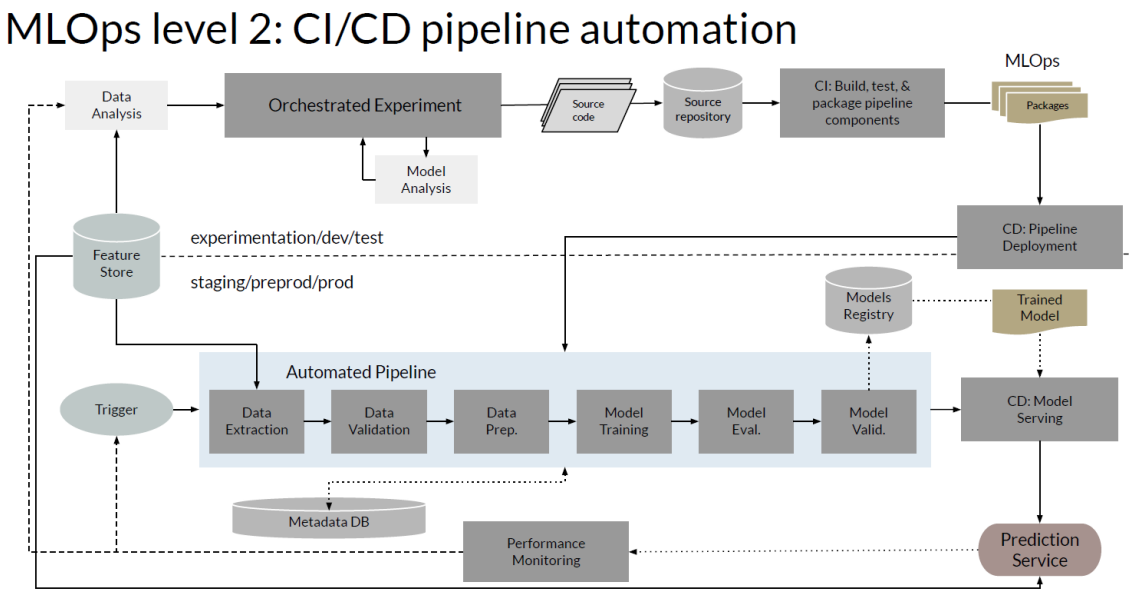
- Your newly trained customer churn model might produce an overall better predictive accuracy compared to the previous model, but the accuracy values per customer region might have a large variance.
- Finally, infrastructure compatibility and consistency with the prediction service API, are some other factors that you need to consider before finally deploying your models.
- In other words, will the new model run on the current infrastructure?
- In addition to offline model validation, a newly deployed model undergoes online model validation in either a **canary** deployment or an **A/B** testing setup during the transition to serving prediction for online traffic. You'll learn more about this in later sessions.



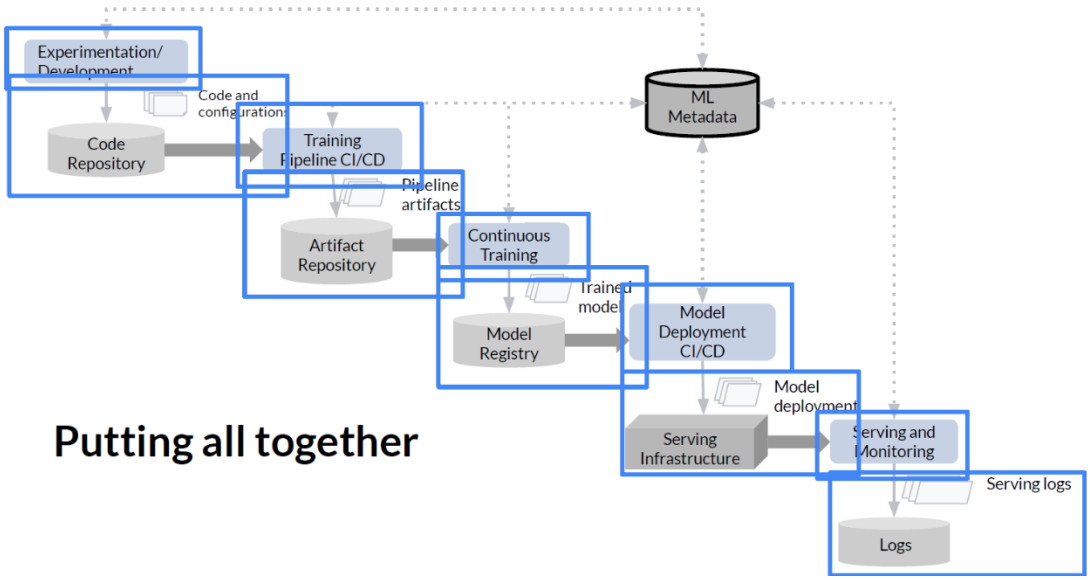
- An optional additional component for level one MLOps is a **feature store**.
- A feature store is a centralized repository where you **standardize the definition, storage, and access of features for training and serving**.
- Ideally, a feature store will provide an **API** for both high throughput batch serving and low latency, real time serving for the feature values, and support both training and serving workloads.
- A feature store helps you in many ways.
- First, it lets you **discover and reuse available feature sets instead of recreating the same or similar feature sets**, avoiding having similar features that have different definitions by maintaining features and their related metadata.
- Moreover, you can potentially serve up-to-date feature values from the feature store, and avoid training serving skew by using the feature store as a data source for experimentation, continuous training, and online serving.
- This approach **makes sure that the features used for training are the same ones used during serving**.
- For example, when it comes to experimentation, data scientists can get an offline extract from the feature store to run their experiments.
- For continuous training, the automated training pipeline can fetch a batch of the up-to-date feature values of the data set.
- For online prediction, the prediction service can fetch **feature values such as customer demographic features, product features, and current session aggregation features.**



- Another key component is the **metadata store**, where information about each execution of the pipeline is recorded in order to help with data and artifact lineage, reproducibility, and comparisons.
- It also helps you debug errors and anomalies.
- Each time you execute the pipeline, the metadata store tracks information such as the pipeline and component versions which were executed, the start and end time and date, and how long the pipeline took to complete each of the steps, and the input and output artifacts from each step, and more.
- Basically, what this means is that you could rely on **pointers to the artifacts** produced by each step of the pipeline, like the **location of prepared data, validation anomalies, computed statistics**, and so on to seamlessly resume execution in case of an interruption.
- So, **tracking these intermediate outputs, helps you resume the pipeline from the most recent step** if the pipeline stopped due to a failed step without having to restart the pipeline as a whole.



- The truth is that at the current stage of the development of MLOps best practices, level two is still somewhat speculative.
- This diagram presents one of the current architectures, which is focused on enabling rapid and reliable update of the pipelines in production.
- This requires robust automated CI/CD to enable your data scientists and ML engineers to rapidly explore new ideas around **feature engineering, model architecture, and hyperparameters**.
- They can implement these ideas and automatically build, test, and deploy new pipeline components to the target environment.
- These MLOps setup includes components like source code control, test and build services, deployment services, a model registry, a feature store, a metadata store, and a pipeline orchestrator.
- Since this is a lot to take in, let's look at the different stages of the machine learning, continuous integration and continuous delivery pipeline in a simplified and more digestible form.



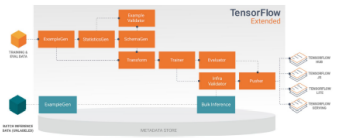
- To summarize, let's use this picture to understand the steps of the level two life cycle.
- First, you have experimentation and development. This is where you iteratively try out new algorithms and new modeling, and the experiment steps are orchestrated.
- The output of this stage is the source code of the ML pipeline steps that are then pushed to a source repository.
- Next, comes the CI/CD stage for the training pipeline itself.
- Here, you build the source code and run various tests. The outputs of this stage are pipeline entities like software packages, executables and artifacts to be deployed in the later stage.
- The output of this stage is a deployed pipeline with a new implementation of the model. Then you train your models. Here, the pipeline is automatically executed in production based on a schedule or in response to a trigger.
- The output of this stage is a trained model, and that is pushed to the model registry. Once the model's been trained, the goal of the pipeline is now to deploy the model using continuous delivery.
- You do this by serving the trained model as a prediction service for predictions. The output of this stage is a deployed model prediction service.

- Finally, once all your models have been trained and deployed, it is the role of the monitoring service to collect statistics on the model performance based on live data.
- The output of this stage is the data collected in the logs from the operation of the serving infrastructure, including the prediction request data which will be used to form the new data set and retrain your model

Developing Components for an Orchestrated Workflow

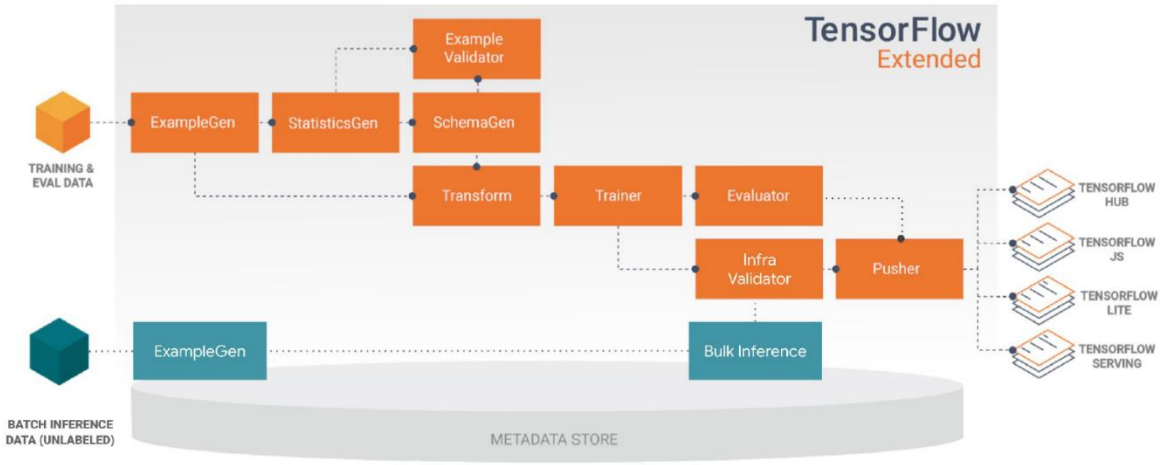
Orchestrate your ML workflows with TFX

- Pre-built and standard components, and 3 styles of custom components
- Components can also be containerized
- Examples of things you can do with TFX components:
  - Data augmentation, upsampling, or downsampling
  - Anomaly detection based on confidence intervals or autoencoder reproduction error
  - Interfacing with external systems like help desks for alerting and monitoring
  - ... and more!



- One of the key parts of an MLOps infrastructure is the **training pipeline**.
- Let's look now at developing training pipelines using TFX, including ways to adapt your pipelines to meet your needs with custom components.
- TFX is an open-source framework that you can use to create ML pipelines.
- TFX enables you to implement your model training workflow in a variety of execution environments, including containerized environments like Kubernetes.
- TFX pipelines, organize your workflow into a series of components where each component performs a step in your ML workflow.
- TFX standard components provide proven functionality to help you get started building in ML workflow easily.
- You can also include **custom components** in your workflow, including creating components which run in containers and can use any language or library that you can run in a container, such as performing data analysis using R.
- Custom components let you extend your ML workflow by creating components that are tailored to meet your needs, such as data augmentation, up sampling or down sampling , anomaly detection, or interfacing with external systems such as help desks for alerting and monitoring, and much more.

Hello TFX



- This is what a starter or Hello World, TFX pipeline typically looks like.
- The things in orange here and green are components, and in this case, these are standard components which come with the TFX out of the box, but they could just as easily be custom components that you created.
- The components in orange are a **training** pipeline and the ones in green are an **inference** pipeline for doing batch inference.

Anatomy of a TFX Component

Component Specification

- The component's input and output contract

Executor Class

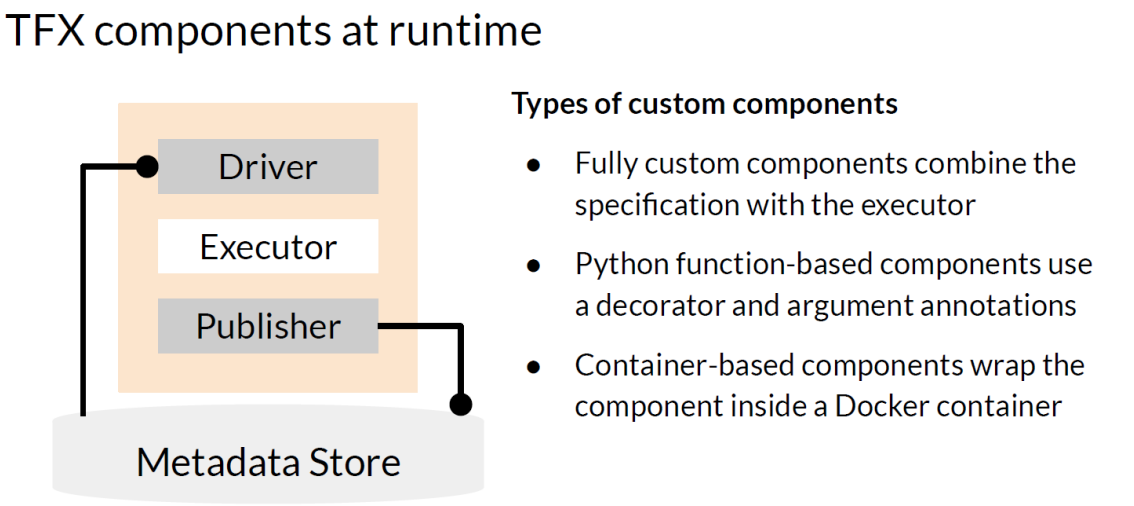
- Implementation of the component's processing

Component Class

- Combines the specification with the executor to create a TFX component

- So, by mixing standard components and custom components, you can build an ML workflow that meets your needs, while taking advantage of the best practices built into the TFX standard components.
- Now, let's look at how TFX components are put together.
- They're essentially composed of a component specification and executer class which are packaged in a component class.
- The specification here defines the components, input and output contract.
- This contract specifies the components, input and output artifacts and the parameters that are used for the component execution.
- A components executor class provides the implementation for the work performed by the component. It's the main code for a component

- Finally we have the component class which combines the component specification with the executor for use as a component in the TFX pipeline.
- Note that this is the implementation style which is used by the TFX standard components and full custom style components, but there are also two other styles of components for creating custom components, which I'll discuss next.



- When a pipeline runs a TFX component, the component is executed in three phases.
- First, the driver uses the component specification to retrieve the required artifacts from the metadata store and pass them into the component.
- Next, the executor performs the components work.
- Finally, the publisher uses the component specification and the results from the executor to store the components' outputs in the metadata store.
- Most custom component implementations do not require you to customize the driver with the publisher.
- Typically, modification to the driver and publisher should only be necessary if you want to change the interaction between your pipelines components and the metadata store.
- If you only want to change the inputs, outputs or parameters for your component, you only need to modify the component specification.
- There are three types of custom components: Python function based custom components, container based custom components and fully custom components.
- Fully custom components, which were discussed in the previous slides, lets you build components by defining the component specification, executer and component interface classes.
- This approach lets you reuse and extend a standard component to meet your needs.
- Python function based components are the easiest to build (easier than container based components and fully custom components).
- They only require Python function for the executor with a decorator and annotations.
- On the other hand, container-based components provide the flexibility to integrate code written in any language into your pipeline, assuming that you can execute that code in a Docker container.
- To create a container-based component, you create a component definition that is very similar to a Docker file and cause a wrapper function to instantiate it.
- Next, we'll learn how you can make use of each of the custom component types.

Python function-based components

```
@component
def MyValidationComponent(
    model: InputArtifact[Model],
    blessing: OutputArtifact[Model],
    accuracy_threshold: Parameter[int] = 10,
    ) -> OutputDict(accuracy=float):
  '''My simple custom model validation component.'''

  accuracy = evaluate_model(model)
  if accuracy >= accuracy_threshold:
    write_output_blessing(blessing)

  return {
    'accuracy': accuracy
  }
```

- The Python function-based component style makes it easy for you to create TFX custom components by saving you the effort of defining component specification class, executor class and component interface class.
- In this style you write a function that is decorated and annotated with type hints.
- The type hints describe the InputArtifacts, OutputArtifacts and parameters of your component.
- Writing a custom component for simple model validation in this style is very straightforward, as is shown in this example.
- The component specification is defined in the Python functions arguments using type annotations that describe if an argument is an InputArtifact, OutputArtifact or a Parameter.
- The function body defines the components executor.
- The component interface is defined by the component decorator to your function.
- By decorating your function with the component decorator and defining the function arguments with type annotations, you could create a component without the complexity of building a component specification and executor and a component interface.

Container-based components

```
from tfx.dsl.component.experimental import container_component
from tfx.dsl.component.experimental import placeholders
from tfx.types import standard_artifacts

grep_component = container_component.create_container_component(
    name='FilterWithGrep',
    inputs={'text': standard_artifacts.ExternalArtifact},
    outputs={'filtered_text': standard_artifacts.ExternalArtifact},
    parameters={'pattern': str},
    ...
)
```

- Next let's look at how to create a container-based component.
- Container-based components are backed by containerized command line arguments or programs rather, and in some ways are similar to creating a Docker file.
- To create one, you need to specify a Docker container image that includes your components dependencies.
- Then you call the create container component function and pass the component definition, including the components, inputs, outputs and parameters.

Container-based components

```
grep_component = container_component.create_container_component(
    ...
    image='google/cloud-sdk:278.0.0',
    command=[
        'sh', '-exc',
        '''
          ...
        ''',
        '--pattern', placeholders.InputValuePlaceholder('pattern'),
        '--text', placeholders.InputUriPlaceholder('text'),
        '--filtered-text',
placeholders.OutputUriPlaceholder('filtered_text'),
    ],
)
```

- There are other parts of the configuration, like the container image name, and optionally the image tag.
- Finally, for the body of the component, you have the command parameter which defines the container entry point command line.

- As with Docker files, this isn't executed within a shell unless you specify that in your command line.
- The command line can use placeholder objects that are replaced at compilation time with the input, output or parameters.
- The placeholder objects can be imported from tfx.dsl.component.experimental.placeholders, and in this example, the component code uses *gsutil* to upload the data to Google Cloud storage, so the container image needs to have *gsutil* installed and configured as a dependency here.
- This approach is more complex than building a Python function-based component, since it **requires packaging your code as a container image.**
- This approach is the **most suitable for including non-Python code** in your pipeline, or for building Python components with complex runtime environments or dependencies.

Fully custom components

- Define custom component spec, executor class, and component class
- Component reusability
  - Reuse a component spec and implement a new executor that derives from an existing component

- Returning to fully custom components, this style lets you build components by directly defining the component specification, executor class and component class.
- This approach also lets you reuse and extend a standard component or other preexisting component to meet your needs.
- For example, if an existing component, which could be a custom component, is defined with the same inputs and outputs as the custom component that you're developing, you can simply override the executor class of the existing component.
- This means that you can reuse a component specification and implement a new executor that derives from an existing component.
- In this way you can reuse functionality built into existing components and implement only the functionality that is required.
- The primary use of this component style is to **extend existing components**.
- Otherwise, if you don't need a containerized component, you should probably use the Python function style instead.
- However, developing a good understanding of the style of the fully custom component will help you to better understand all TFX components.
- So let's take a closer look at how to create a fully custom component.

Defining input and output specifications

```
class HelloComponentSpec(types.ComponentSpec):
  INPUTS = {
      # This will be a dictionary with input artifacts, including URIs
      'input_data': ChannelParameter(type=standard_artifacts.Examples),
  }
  OUTPUTS = {
      # This will be a dictionary which this component will populate
      'output_data': ChannelParameter(type=standard_artifacts.Examples),
  }
  PARAMETERS = {
      # These are parameters that will be passed in the call to create an instance of this component
      'name': ExecutionParameter(type=Text),
  }
```

- Developing a fully custom component first requires you to define a component's spec, which contains a set of input and output artifacts specifications for the new component.
- Second, you must define any non-artifact execution parameters that are needed for the new component.
- So, there are three main parts of the component specification the inputs, outputs and parameters.
- Inputs and outputs are wrapped in **channels**, essentially dictionaries of typed parameters for the input and output artifacts.
- The parameters is a dictionary of additional execution parameter items which are passed into the executor and are not metadata artifacts.

Implement the executor

```
class Executor(base_executor.BaseExecutor):
  def Do(self, input_dict: Dict[Text, List[types.Artifact]],
         output_dict: Dict[Text, List[types.Artifact]],
         exec_properties: Dict[Text, Any]) -> None:
    ...
```

- Next, you need to create an executor class. Basically this is a subclass of base_executor.BaseExecutor with its Do function overwritten.
- In the Do function, the arguments input_dict, output_dict and exec_properties are passed in, and those map to the inputs, outputs, and parameters that are defined in the components spec.
- For exec properties, the values can be fetched directly through a dictionary look up.

Implement the executor

```
class Executor(base_executor.BaseExecutor):
    ...
    split_to_instance = {}
    for artifact in input_dict['input_data']:
      for split in json.loads(artifact.split_names):
        uri = os.path.join(artifact.uri, split)
        split_to_instance[split] = uri
    for split, instance in split_to_instance.items():
      input_dir = instance
      output_dir = artifact_utils.get_split_uri(
          output_dict['output_data'], split)
      for filename in tf.io.gfile.listdir(input_dir):
        input_uri = os.path.join(input_dir, filename)
        output_uri = os.path.join(output_dir, filename)
        io_utils.copy_file(src=input_uri, dst=output_uri, overwrite=True)
```

- Continuing with implementing the executor for artifacts in the input_dict and output_dict, there are convenient functions available in the artifact utilities class of TFX that can be used to fetch and artifacts instance or its URI.

Make the component pipeline-compatible

```
from tfx.types import standard_artifacts
from hello_component import executor

class HelloComponent(base_component.BaseComponent):
  SPEC_CLASS = HelloComponentSpec
  EXECUTOR_SPEC = ExecutorClassSpec(executor.Executor)
```

- Now that the most complex part is complete, the next step is to assemble these pieces into a component class to enable the component to be used in a pipeline.
- There are several steps. First you need to make the component class a subclass of base_component.BaseComponent or a different component if you're extending an existing component.
- Next, you assign class variables, spec_class and executor_spec with a component spec and executor classes respectively, that you just defined.

Completing the component class

```
class HelloComponent(base_component.BaseComponent):
  ...
  def __init__(self,
               input_data: types.Channel = None,
               output_data: types.Channel = None,
               name: Optional[Text] = None):
    if not output_data:
      examples_artifact = standard_artifacts.Examples()
      examples_artifact.split_names = input_data.get()[0].split_names
      output_data = channel_utils.as_channel([examples_artifact])

    spec = HelloComponentSpec(input_data=input_data, output_data=output_data, name=name)
    super(HelloComponent, self).__init__(spec=spec)
```

- The final step to completing the custom component is to finish implementing the *init*, which will initialize the component.
- Here you define the **constructor function** by using the arguments to the function to construct an instance of the component spec class, then invoke the super function with that value along with an optional name.
- When an instance of the component spec is created, type checking logic in the base_component.BaseComponent class will be invoked to ensure that the arguments passed in are compatible with the types to find in the component spec class.

Assemble into a TFX pipeline

```
def _create_pipeline():
  ...
  example_gen = CsvExampleGen(input_base=examples)

  hello = component.HelloComponent(
      input_data=example_gen.outputs['examples'],
      name='HelloWorld')

  statistics_gen = StatisticsGen(
      examples=hello.outputs['output_data'])
  ...
  return pipeline.Pipeline(
      ...
      components=[example_gen, hello, statistics_gen, ...],
      ...
  )
```

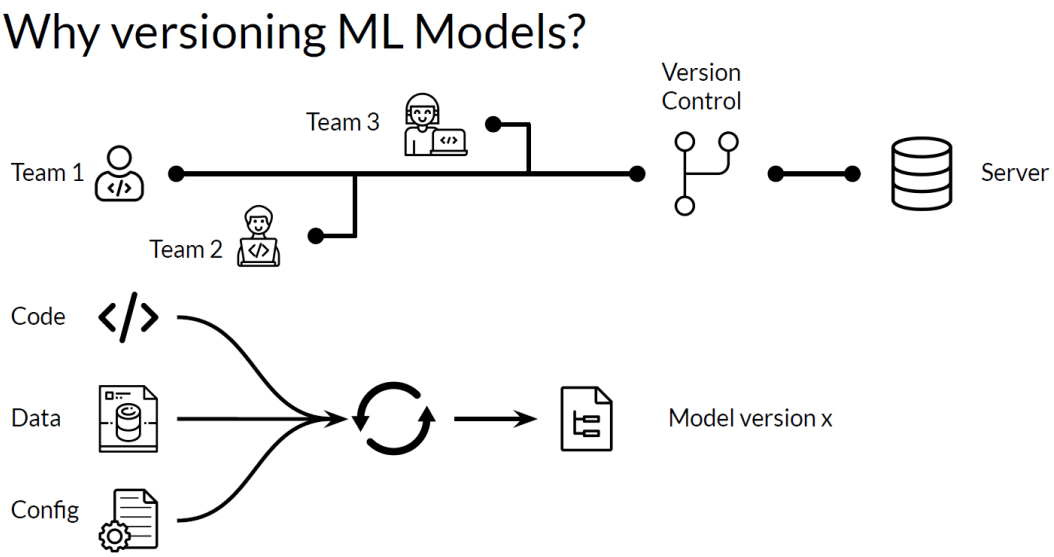
- The last step is to plug the new custom component into a TFX pipeline.
- Besides adding an instance of the new component, you need to wire the upstream and downstream components to it.
- You can generally do this by referencing the outputs of the upstream components in the new component and referencing the outputs of the new component in the downstream components.

- Also, another thing to keep in mind is that you need to add the new component instance to the components list when constructing the pipeline.

Vertex Pipelines

- Pipelines help you automate and reproduce your ML workflow.
- Vertex AI integrates the ML offerings across Google Cloud into a smooth development experience.
- Previously, models trained with AutoML and custom models were accessible via separate services.
- Vertex AI combines both into a single API, along with other new products.
- Vertex AI also includes a variety of MLOps products, like Vertex Pipelines.

Managing Model Versions



- Now let's turn to another important topic in MLOps, managing model versions.
- Let's look at why version control is so important and some of the challenges of versioning models.
- In normal software development, especially with teams, organizations rely on version control software to help teams manage and control changes to their code.
- But imagine if you didn't have that. How would you enable multiple developers to stay in sync?
- How do you roll back when there are problems? How would you do continuous integration?
- Well, just like with software development, when you're developing models, you have all these needs and more.
- Generating models is an iterative process. During development, you typically generate several models and compare one against the other to evaluate the performance of each model.
- **Each model version may have different code, data, and configuration**.
- You need to keep track of all of this to properly reproduce results.
- This is where model versioning is important.
- Versioning will improve collaboration at different levels, from individual developers, to teams, all the way up to organizations.

How ML Models are versioned?

How software is versioned?

Version: MAJOR.MINOR.PATCH

- MAJOR: Contains incompatible API changes
- MINOR: Adds functionality in a backwards compatible manner
- PATCH: Makes backwards compatible bug fixes

ML Models versioning

- No uniform standard accepted yet
- Different organizations have different meanings and conventions

- How should you version your models? First, let's think about how you version software.
- In most cases, you version software with **a combination of three numbers**.
- These numbers are the major version, the minor version, and a patch number of the release.
- The **major version** usually increases when you make incompatible API changes.
- The minor version number is increased when you add functionality in a backwards compatible way
- The patch number is increased when you make backward-compatible **bug fixes**.
- Can you take a similar approach with your models?
- As of now, there's no uniform standard which is widely accepted across the industry to version models.
- Different companies have adopted their own conventions for versioning.
- As a developer in their organization, you need to understand how the version their models.

A Model Versioning Proposal

Version: MAJOR.MINOR.PIPELINE

- MAJOR: Incompatibility in data or target variable
- MINOR: Model performance is improved
- PIPELINE: Pipeline of model training is changed

- Here's one possible approach that I'd like to propose, which is simple to understand and is in line with normal software versioning.
- Let's use a combination of three numbers and to note these as major, minor, and pipeline versions.
- The major version will increment when you have an **incompatible data change**, such as a **schema change or target variable change that can render the modeling compatible with its prior versions** when it's used for predictions.

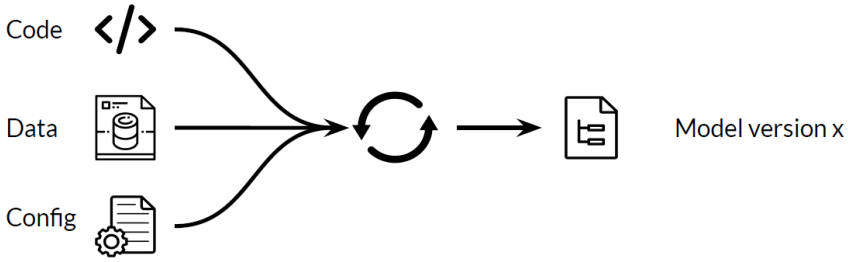
- The minor version will increment when you believe that you've improved or enhanced the model's output.
- Finally, the pipeline version will correspond to an update in the training pipeline, **but it need not improve or even change the model itself.**
- Other versioning styles include arbitrary grouping, black-box functional models, and pipeline execution versioning, but we will not discuss them here.
- Rather, I'd like to **focus on how to retrieve older models, leverage model lineage, and use model registries** to simplify production workflow.

Retrieving older models

- Can ML framework be leveraged to retrieve previously trained models?
- ML framework may internally be versioning models

- One way to test a versioning style is to ask, can you leverage a model's frameworks capability to retrieve previously trained models?
- You may have an intuition that for an ML framework to retrieve older models, the framework must be internally versioning the models through some versioning technique.

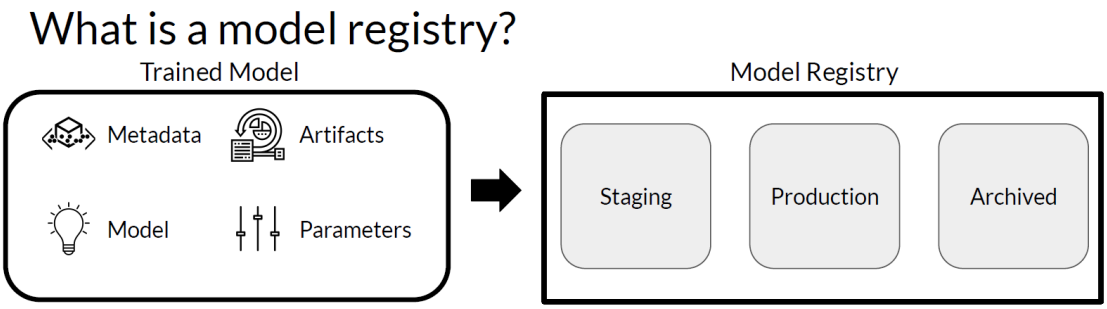
What is model lineage?



- Artifacts: information needed to preprocess data and generate result (code, data, config, model)
- ML orchestration frameworks may store operations and data artifacts to recreate model
- Post training artifacts and operations are usually not part of lineage

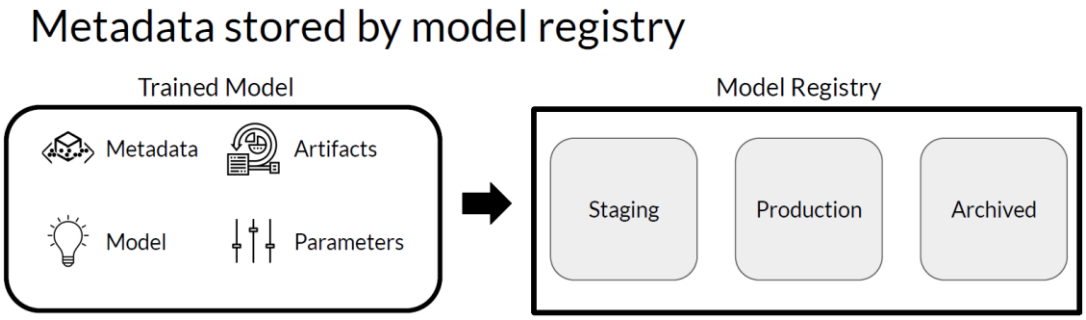
- Different ML frameworks may use different techniques to retrieve previously trained models.
- One technique is by making use of **model lineage**.
- Model lineage is a set of relationships among the artifacts that resulted in the trained model.
- To build model artifacts, you have to be able to track the code that builds them, and the data, including pre-processing operations, that the model was trained and tested with.
- ML orchestration frameworks like TFX will store this model lineage for many reasons including recreating different versions of the model when necessary.
- Note that model lineage usually only includes those artifacts and operations that were part of model training.

- Post-training artifacts and operations are usually not part of the lineage.



- Central repository for storing trained ML models
- Provides various operations of ML model development lifecycle
- Promotes model discovery, model understanding, and model reuse
- Integrated into OSS and commercial ML platforms

- Now let's take a look at model registries.
- **A model registry is a central repository for storing trained models.**
- **Model registries provide an API** for managing trained models throughout the model development lifecycle.
- Model registries are essential in supporting model discovery, model understanding, and model reuse, including in large-scale environments with hundreds or thousands of models.
- As a result, model registries have become an integral part of many open-source and commercial ML platforms.



- Model versions
- Model serialized artifacts
- Free text annotations and structured properties
- Links to other ML artifact and metadata stores

- Next, let's look at the metadata which is stored in most of the open-source and commercial model registries.
- **Metadata** usually includes the different model versions available.
- Some model registries provides storage for serialized model artifacts.
- In order to improve the model discoverability within the model registry, it's important to store some **free text annotations and other structured or searchable properties** of the models.

- In order to promote the model lineage, registry sometimes include links to other ML artifact and metadata stores.

Capabilities Enabled by Model Registries

- Model search/discovery and understanding
- Approval/Governance
- Collaboration/Discussion
- Streamlined deployments
- Continuous evaluation and monitoring
- Staging and promotions

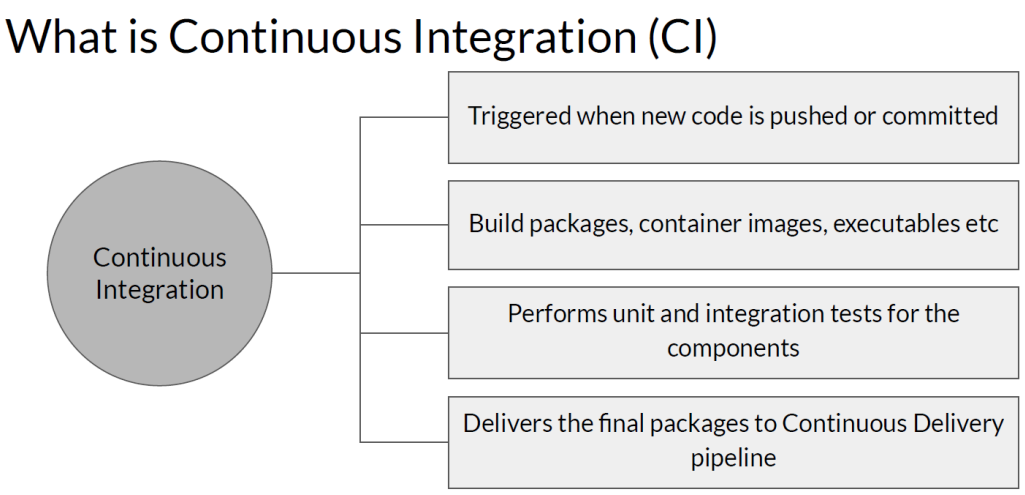
- Model registries are really very useful things to have.
- Model registries promote model search and discoverability within your organization, and that can help improve the understanding of the model among your team.
- Model registries can help enforce a set of approval guidelines which need to be followed when uploading models, which can help improve governance.
- By sharing models with your team, you're improving the chances of collaboration among your co-workers.
- Model registries can also help streamline deployments.
- Model registries can even provide a platform for continuous evaluation, monitoring, and staging, and promotions.

Examples of Model Registries

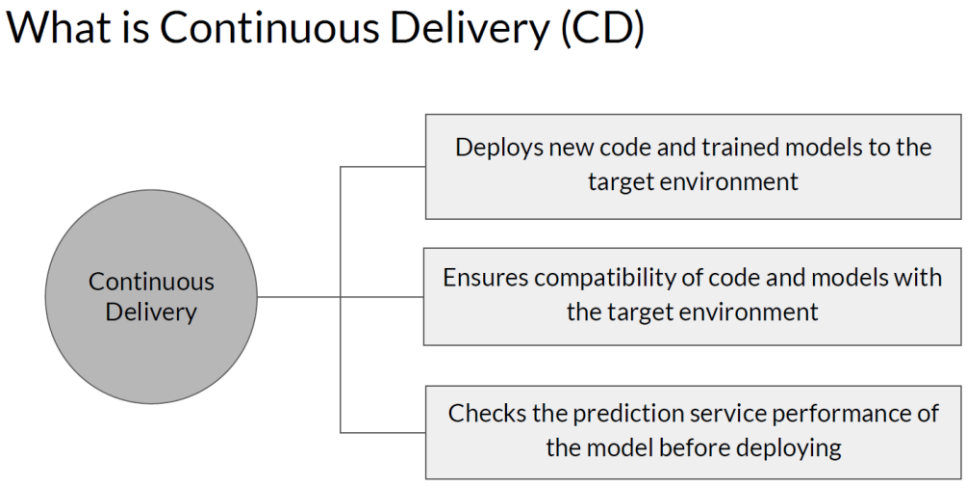
- Azure ML Model Registry
- SAS Model Manager
- MLflow Model Registry
- Google AI Platform
- Algorithmia

- Here's a list of some of the currently available model registries.
- As you can see there are quite a few, and each has somewhat different feature sets.
- I won't do a comparison here. This is just to make you aware of some of the offerings that are available.
- For example, there's Azure ML model registry, SAS Model Manager, MLflow Model Registry, the Google AI platform, and Algorithmia.

Continuous Delivery

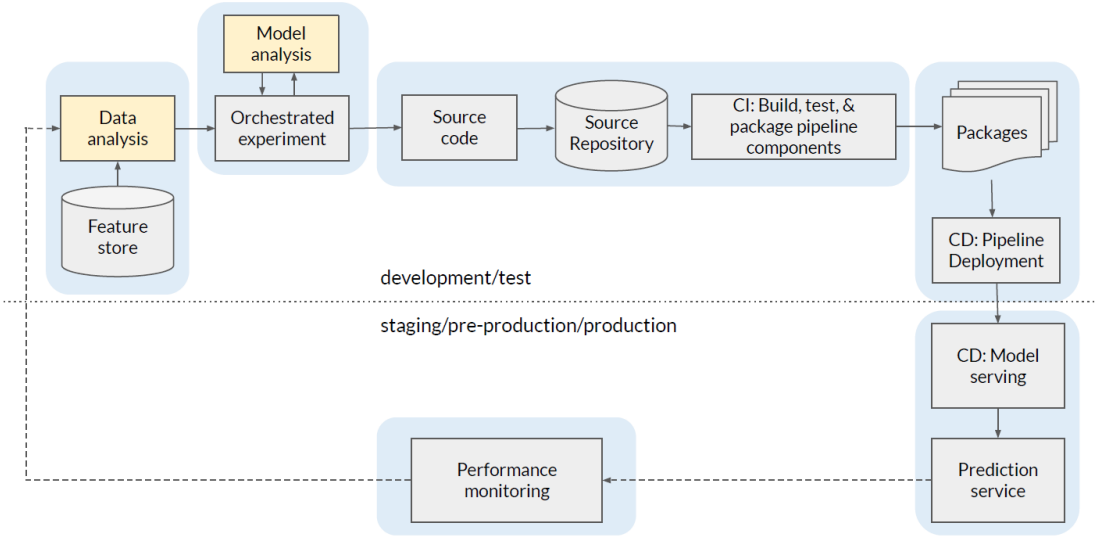


- In more mature MLOPs processes and where more than a few models need to be managed, it's important to implement a **robust deployment process**.
- This is especially true when model predictions are **served online** as part of a user facing application.
- Let's discuss robust deployment using continuous delivery now.
- First, before deploying, you need to make sure that your code works, which you should determine through **comprehensive unit testing**.
- This is automated with continuous integration or CI.
- **CI triggers whenever new code is committed or pushed to your source code repository**.
- It mainly performs building, packaging and testing for the components.
- The quality of the testing will be determined by the coverage and quality of your unit test suite.
- If all tests pass, it delivers the tested code and packages to a continuous delivery pipeline.



- Next, continuous delivery or CD deploys new code and trained models to the **target environment**.
- It also ensures **compatibility** of code and models with the target environment, and for an ML deployment, it should check the prediction performance of the model to make sure that the new model can be served successfully.

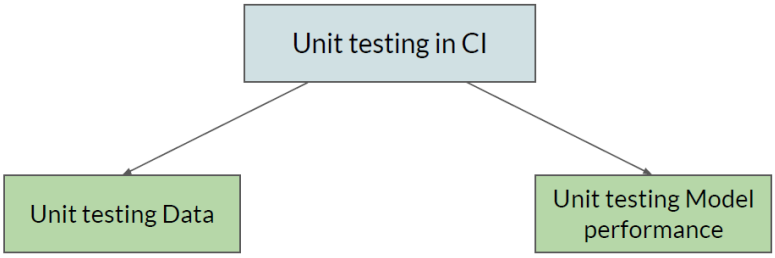
CI/CD Infrastructure



- The full continuous integration, continuous delivery process and infrastructure is referred to as **CI/CD.**
- It includes two different forms of data analysis and model analysis.
- During experimentation, data analysis and model analysis are usually manual processes which are performed by data scientists.
- Once a model and code have been promoted to a production training pipeline, data and model analysis should be performed **automatically**.
- As part of the promotion of the code to production, source code is committed to a source code control and CI is initiated.
- CD, then deploys the production code to a production training pipeline and models are trained.
- Trained models are then deployed to an online serving environment or batch prediction service.
- During serving, the performance monitoring collects the performance metrics of the model from **live data**.

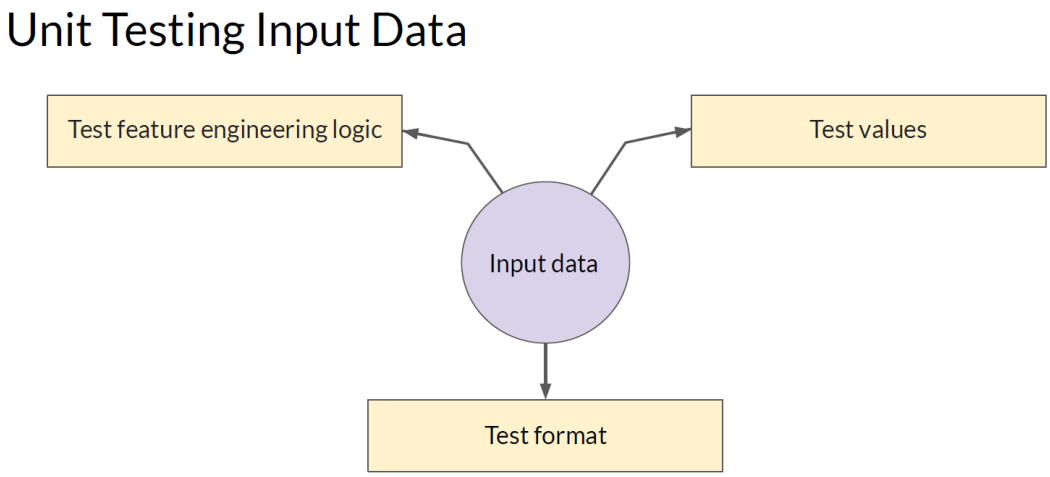
Unit Testing in CI

- Testing that each component in the pipeline produces the expected artifacts.

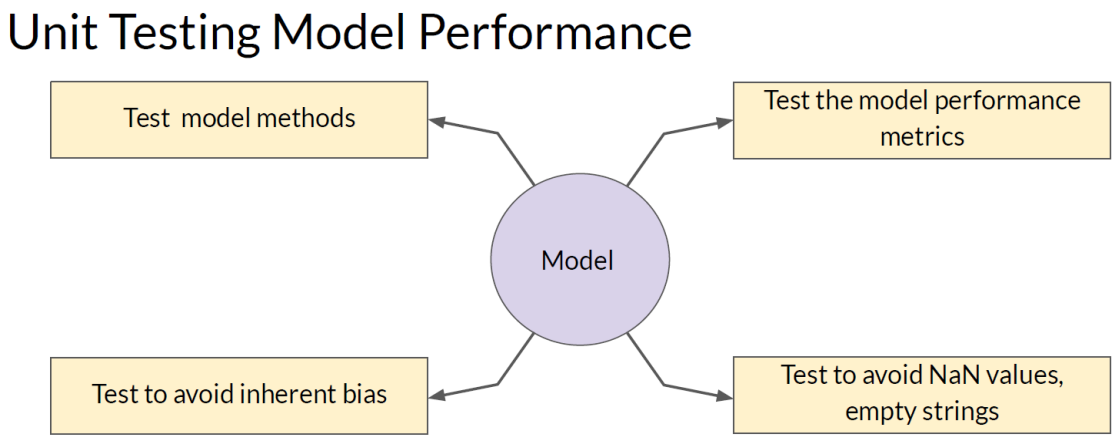


- Let's look at the **two main tests that are performed during continuous integration – unit testing and integration testing**.
- In unit testing, you test each component to make sure that they're producing correct outputs.

- In addition to unit testing our code, which follows the standard practice for software development, there are **two additional types of unit tests** when doing CI for machine learning.
- The **unit tests for our data and the unit tests for our model**.

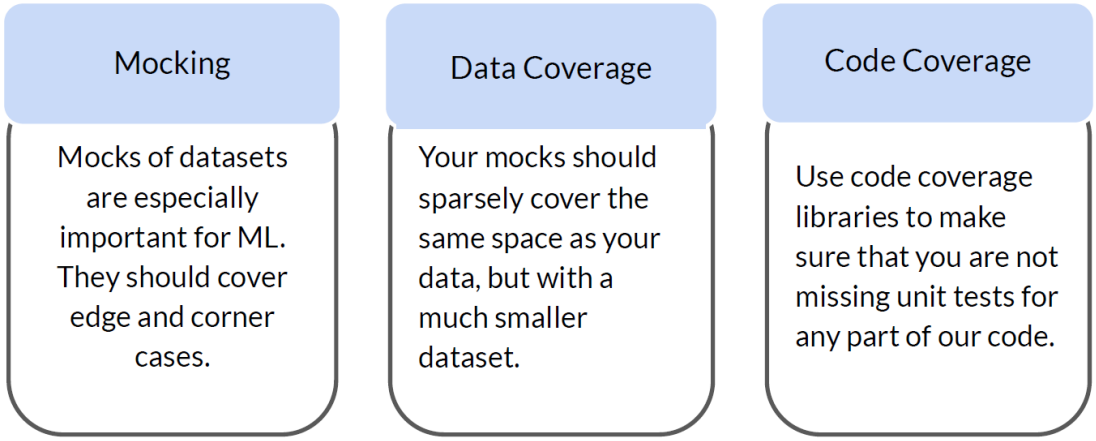


- Unit testing for data is **not** the same as performing data validation on your raw features.
- It's **primarily concerned with the results of your feature engineering.**
- You can write unit tests to **check if engineered features are calculated correctly**.
- It includes tests to check whether they are **scaled** or **normalized** correctly, one hot vector values are correct, embedding are generated and used correctly, etc.
- And you will also do tests to confirm if columns and data are the **correct types**, in the right range, not empty, and so forth.



- Your modelling code should also be written in a modular way which allows it to be testable.
- You need to write unit tests for the functions you use inside your modeling code to **check if the functions return their output in the correct shape and type**.
- For **numerical** features, it includes testing for NaN (or not a number), and for **string** features, it includes testing for empty strings and so forth.
- You also need to add tests to make sure that the **accuracy, error rates, AUC ROC, etc. are above a performance baseline that you specify**.
- Even if the trained model has acceptable accuracy, you need to test it against **data slices** to make sure that the model is accurate for key subsets of the data in order to avoid bias.

ML Unit Testing Considerations



- While you should perform standard unit testing of your code, there are some additional considerations for ML
- These include the design of your **mocks** which is especially important for ML unit testing.
- They should be designed **to cover your edge and corner cases**, which requires you to think about each of your features and your domain, and identify where those edge and corner cases are.
- Ideally, your mock should occupy roughly the same region of your feature space as your actual data, but much more sparsely, of course, since your mock data set should be much smaller than your actual data set in most cases.
- If you've created good mocks and good tests, you should have good code coverage, but just to be sure take advantage of one of the available libraries to test and track your code coverage.

Infrastructure validation

When to apply infrastructure validation

- Before starting CI/CD as part of model training
- Can also occur as part of CI/CD as a last check to verify that the model is deployable to the serving infrastructure

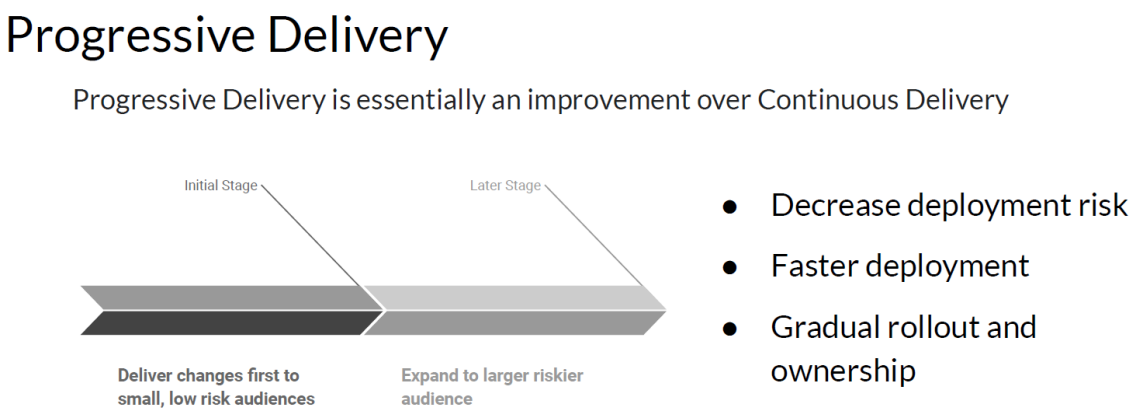
TFX InfraValidator

- TFX InfraValidator takes the model, launches a sand-boxed model server with the model, and sees if it can be successfully loaded and optionally queried
- InfraValidator is using the same model server binary, same resources, and same server configuration as production

- **Infrastructure validation** acts as an early warning layer before pushing a model into production to avoid issues with models that might not run or might perform badly when serving requests in production.
- It focuses on the **compatibility** between the model server binary, and the model which is about to be deployed.
- It's a good idea to include infrastructure validation in your training pipeline, so that as you train models, you can avoid problems early.

- You can also run it as part of your CI/CD workflow, which is especially important if you didn't run it during model training.
- Let's take a look at an example of running infrastructure validation as part of a training pipeline.
- In a TFX pipeline the InfraValidator component takes the model, launches a sandbox model server with the model, and sees if it can **successfully be loaded and optionally queried**.
- If the model behaves as expected, then it is referred to as **blessed** and considered ready to be deployed.
- InfraValidator focuses on the compatibility between the model server binary (for example, TensorFlow serving) and the model to deploy
- Despite the name InfraValidator, it is the user's responsibility to configure the environment correctly.
- InfraValidator only interacts with the model server in the user configured environment to see if it works as expected.
- Configuring this environment correctly will ensure that your inferred validation passing or failing will be indicative of whether the model would be servable in the production serving environment.

Progressive Delivery



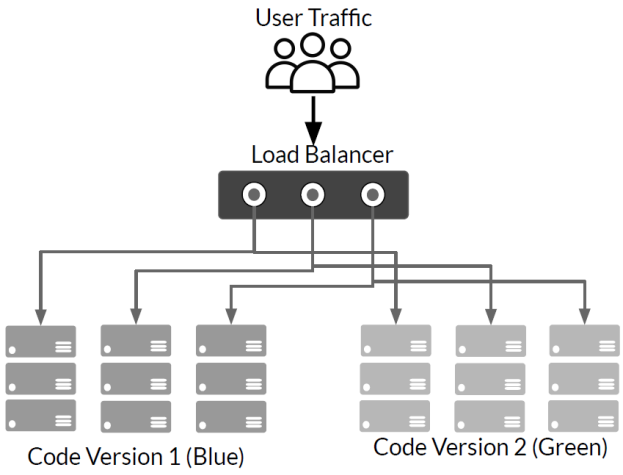
- Now let's take the next step and look at an even more advanced style of deployment, progressive delivery.
- **Progressive delivery** is a software development process that is built upon the core tenets of continuous integration and continuous delivery but is essentially an **improvement over CI/CD**.
- It includes many modern software and development processes including canary deployments, A/B testing, bandits, and observability.
- It focuses on **gradually rolling out new features in order to limit potential negative impact and gauge user response to new product features**.
- The process involves **delivering changes first to small, low-risk audiences**, and then expanding to larger and riskier audiences thereby validating the results.
- The benefits of progressive delivery include, well, it offers controls and safeguards like feature flags to increase speed and decrease deployment risk.
- This can often lead to faster deployments and implement a gradual process for both rollout and ownership.

Complex Model Deployment Scenarios

- You can deploy multiple models performing same task
- Deploying competing models, as in A/B testing
- Deploying as shadow models, as in Canary testing

- Progressive delivery usually involves having **multiple versions deployed at the same time** so that comparisons in performance can be made.
- This practice comes from software engineering, especially for online services.
- Each of the models performs the same tasks so that they can be compared.
- That includes deploying competing models as in A/B testing and deploying to shadow environments to limit the deployment risk as in canary testing.

Blue/Green deployment

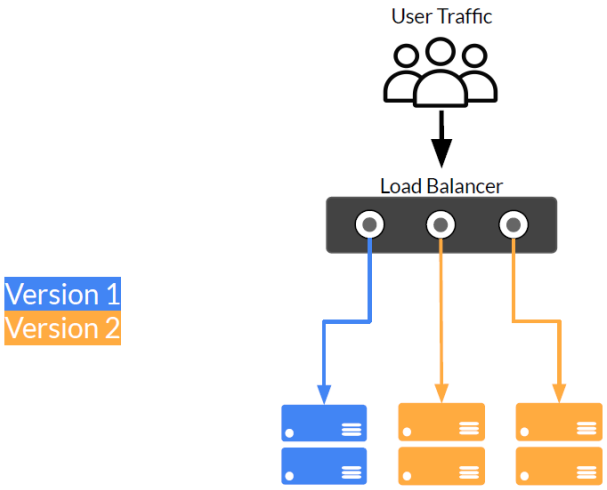


- No downtime
- Quick rollback & reliable
- Smoke testing in production environment

The diagrams are illustrations based on:
https://dev.to/mostlyjason/intro-to-deployment-strategies-blue-green-canary-and-more-3a3#

- A simple form of progressive delivery is **blue/green deployment** where there are **two production serving environments**.
- Requests flow through a load balancer which directs traffic to the **currently live environment which is called blue**.
- Meanwhile, a new version is deployed to the **green** environment which acts as a **staging setup** where a series of tests are conducted to ensure performance and functionality.
- After passing the tests, all traffic is directed to the green deployment.
- **If there are any problems, traffic can be moved back to blue.**
- This means that there's **no downtime** during deployment, rollback is easy, and there is a high degree of reliability, and it includes smoke testing before going live.

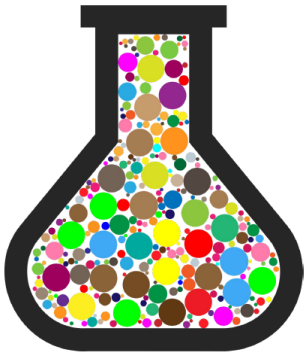
Canary deployment



- A canary deployment is similar to a blue/green deployment, but instead of switching the **entire** incoming traffic from blue to green all at once, **traffic is switched gradually**.
- As traffic begins to use the new version, the performance of the new version is monitored.
- If necessary, the deployment can be stopped and reversed with no downtime and minimal exposure of users to the new version.
- Eventually, all the traffic is being served using the new version.
- Canary deployment is cheaper than a blue-green deployment as it does not require two production environments

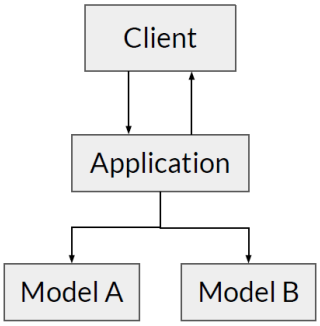
Live Experimentation

- Model metrics are usually not exact matches for business objectives
- Example: Recommender systems
  - Model trained on clicks
  - Business wants to maximize profit
  - Example: Different products have different profit margins



- Progressive deployment is closely related to live experimentation.
- Live experimentation is used to test models to measure the actual business results delivered, or data as closely associated with business results as you can actually measure.
- This is necessary because model metrics which you use to optimize your models during training are usually not exact matches for business objectives.
- For example, consider recommender systems.
- You train your model to maximize the click-through rate, which is how your data is labeled.
- **But what the business actually wants to do is maximize profit**.
- This is closely related to click-through but it's not an exact match, since some clicks will result in more profit than others. For example, different products have different profit margins.

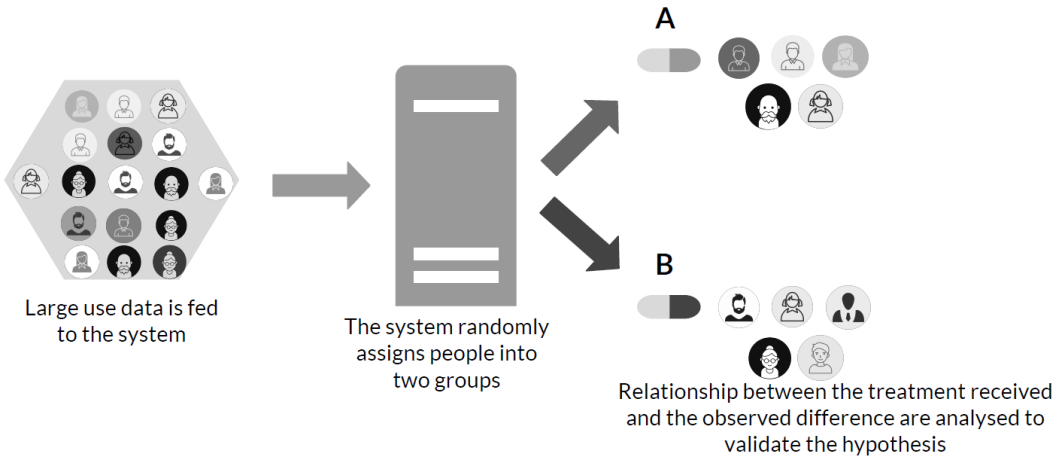
Live Experimentation: A/B Testing



- Users are divided into two groups
- Users are randomly routed to different models in environment
- You gather business results from each model to see which one is performing better

- One simple form of live experimentation is A/B testing.
- In A/B testing, you have at least two different models, or perhaps n different models, and you compare the business results between them to select the model that gives the **best business performance.**
- You do that by dividing users into two or n groups, and you route users to a randomly selected model.
- Notice that it's important here that the user continues to use the **same model for their entire session** if they make multiple requests.
- You then gather the results from each model to select the one that gives the best results.

Live Experimentation: A/B Testing



- This is actually a widely used tool in many areas of science, not just machine learning.
- A/B testing is the process of comparing two variations of the same system usually by testing the response to variant A versus variant B and concluding which of the two variants is more effective.
- Often, A/B testing is used for testing medicines with one of the variants being a placebo.

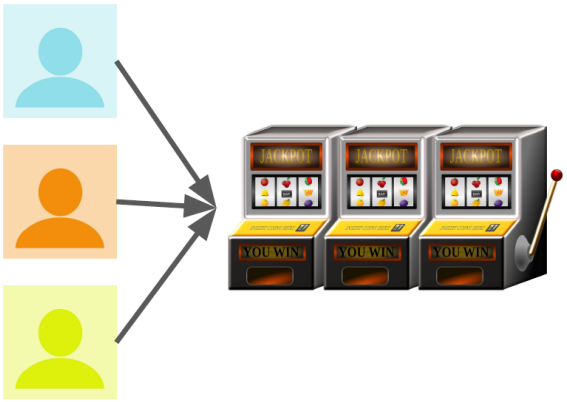
Live Experimentation: Multi-Armed Bandit (MAB)

- Uses ML to learn from test results during test
- Dynamically routes requests to winning models
- Eventually all requests are routed to one model
- Minimizes use of low-performing models

- An even more advanced approach is multi-armed bandits. The multi-armed bandit approach is similar to A/B testing but uses ML to **learn from test results which are gathered during the test**.
- As it learns which models are performing better, it **dynamically routes more and more requests to the winning models.**
- What this means is that eventually, all of the requests will be routed to a single model or a smaller group of similarly performing models.
- One of the major benefits of that is that it **minimizes the use of low-performing models by not waiting for the end of the test to select the winner**.
- The multi-arm bandit approach is a **reinforcement learning architecture which balances exploration and exploitation**.

Live Experimentation: Contextual Bandit

- Similar to multi-armed bandit, but also considers context of request
- Example: Users in different climates



- An even more advanced approach is **contextual bandit**.
- The contextual bandit algorithm is an extension of the multi-arm bandit approach where you also **factor in the customer's environment or other context** of the request when choosing a bandit.
- The context affects how reward is associated with each bandit, so **as contexts change, the model should learn to adapt its bandit choice**.
- For example, consider recommending clothing choices to people in different climates.
- A customer in a hot climate will have a very different context than a customer in a cold climate.
- Not only do you want to find the maximum reward, you also want to reduce the reward loss when you're exploring different bandits.
- When judging the performance of a model, the metric that measures the reward loss is called **regret, which is the difference between the cumulative reward from the optimal policy and the model's cumulative sum of rewards over time**.
- The lower the regret, the better the model. Contextual bandits helps with minimizing regret.

References

- Machine Learning Experiment Tracking
- Machine Learning Experiment Management: How to Organize Your Model Development Process
- https://neptune.ai/blog/mlops
- https://github.com/visenger/awesome-mlops
- Architecture for MLOps using TFX, Kubeflow Pipelines, and Cloud Build
- Machine Learning Model Management: What It Is, Why You Should Care, and How to Implement It
- What is Continuous Delivery?
- Progressive Delivery
- Colab – Kubeflow Pipelines
- Colab – Developing TFX Custom Components
- Colab – Model Versioning with TF Serving
- Colab – CI/CD pipelines with GitHub Actions