The Ultimate Guide to MLOps

NimbleBox.ai

Table of Contents

Preface

This guide glosses over what you need to know about Machine Learning Operations (MLOps) as a field. While it may not be an exhaustive list, the guide provides recommendations and rules that are best practices for transitioning from a business use case to an ML solution.

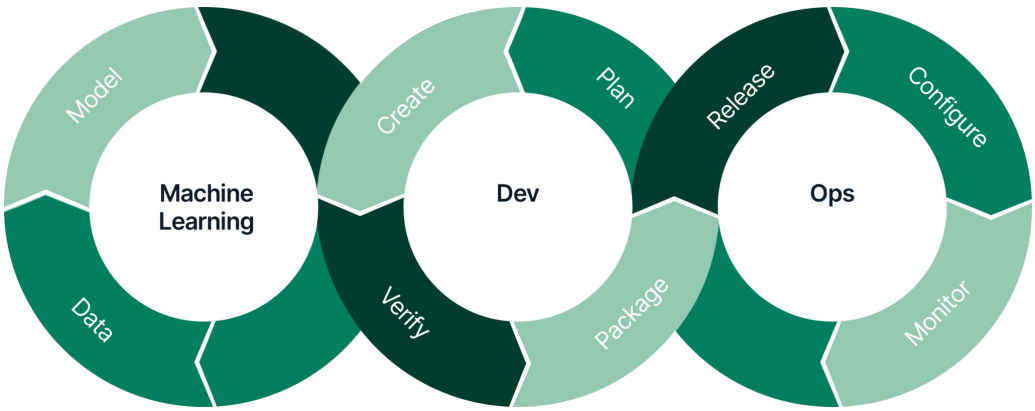
What is MLOps?

Machine Learning Operations or MLOps is a framework of practices that serve as pipelines and rules for successfully maintaining and deploying Machine Learning models.

MLOps, at the intersection of machine learning practices and DevOps, provides a technical and managerial backbone wherein machine learning models can be scaled and automated easily.

Emerged alongside the data science revolution, machine learning models in production should have four qualities: scalable, reproducible, testable, and evolvable.

The end-to-end management of your machine learning workflow — from data to monitoring, is answered by following the simple framework of MLOps.



Defining The Business Use Case

Typically, machine learning has been a later-stage addition to business applications. However, machine learning solutions are implemented to increase metrics, decrease time, and reduce errors while streamlining inefficiencies of a more extensive business process.

Figuring out a machine learning solution without a clear understanding of what you want to achieve is a dangerous adventure that you should avoid.

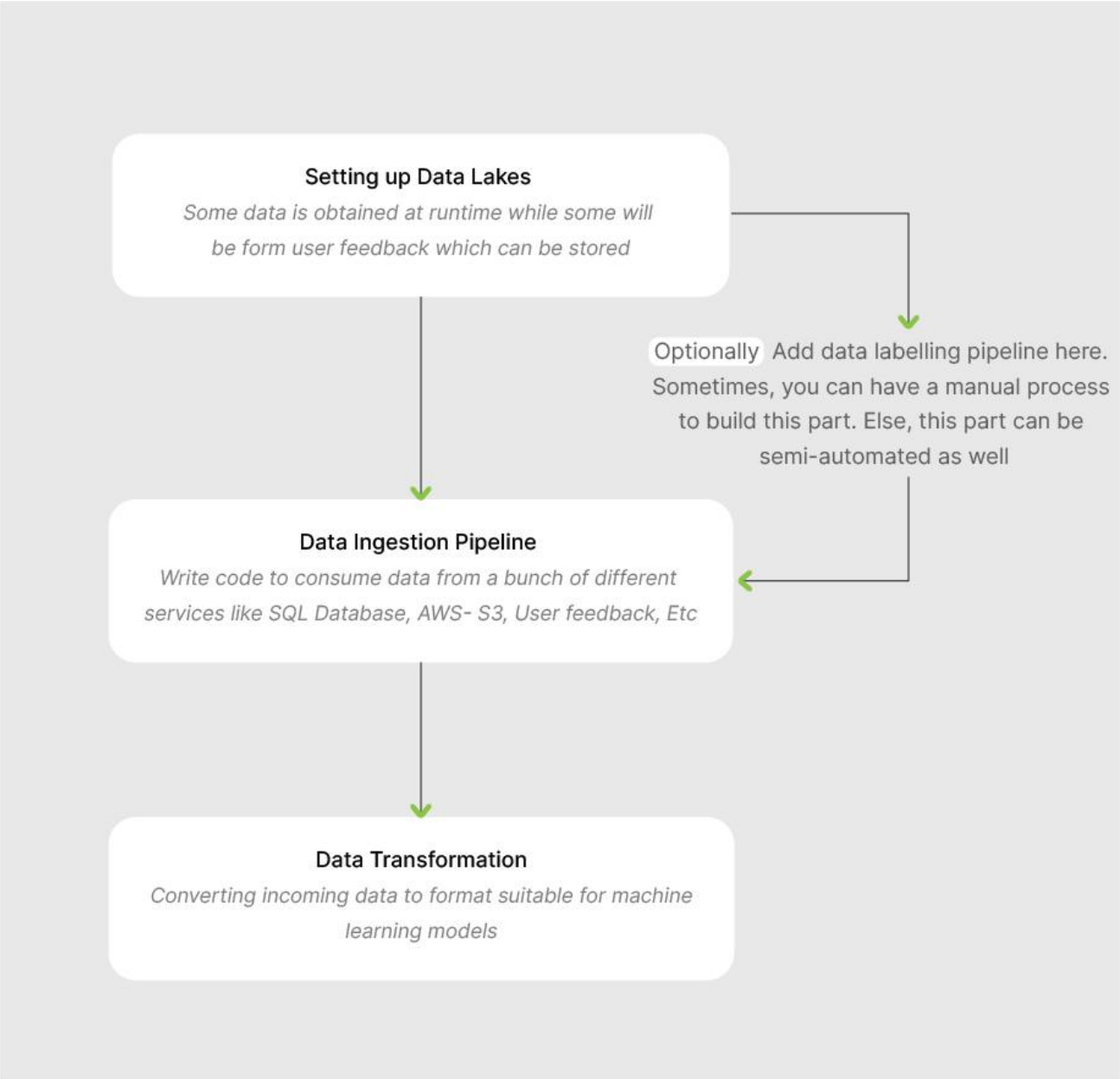
As a leader, you need to develop the success criteria of the business use case you're trying to solve. By involving stakeholders from different verticals, create a final specification with the outcome you want to achieve. This exercise will help you and your team at a later stage when things get hazy, and clarity is needed.

Building machine learning solutions for the part of the process that is fully functional is a smart move for two reasons:

1. Having a fallback
2. Maintaining a baseline against which you can compare the optimized outcome.

Developing individual models with specific tasks truly harnesses machine learning. So often, teams consider success criteria related to the particular problem they're targeting, leading to siloed code development — faster but adding to the technical debt in the long run.

Building Your Data Pipeline



Data plays a vital role in machine learning algorithms as training a model requires data (lots of it!). So if you put in rubbish data, don't be surprised to see garbage output.

Having a promising data pipeline goes a long way. Not only does it help in scaling your application, having a singular source of truth for everything also helps in reducing misunderstandings.

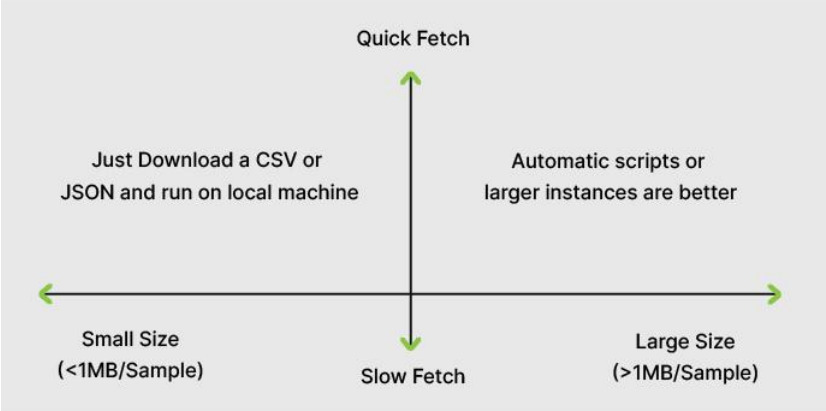
Your objective is to have precise control over data with the ability to fetch anything in minimal lines of code. For example, in the image below, Tesla asked its database the following: “Images of ‘stop signs’ in open and hidden.”

Tesla has a query language built just to fetch the data that they need to train their model. While this might be an overkill for a small organization, the ability to get the exact data that you need to train your model improves the iteration time of your process.

For DevOps teams:

It is instrumental in having a spectrum of fetching speeds and file sizes to set up data lakes that store and retrieve data. Generally, it is preferred to have datasets that are quick to download because processing speeds are a bottleneck in machine learning. A good database is built for querying information to solve a particular problem instead of dumping a large amount of data. With that said, databases can be tricky to create sometimes. They should balance being easy for DevOps to build and manage but not completely

ignore the significant dependencies after machine learning model development begins.



For DevOps and Data Engineering teams:

Once the databases are finalized, you would want to build ingestion pipelines for them. In the best-case scenario, your machine learning models would pull data from an existing pipeline to keep the overlap between machine learning engineers and DevOps minimum. Unfortunately, this becomes a significant bottleneck when independent new features are released. In addition, the non-existent knowledge overlap between DevOps and machine learning teams can be a showstopper.

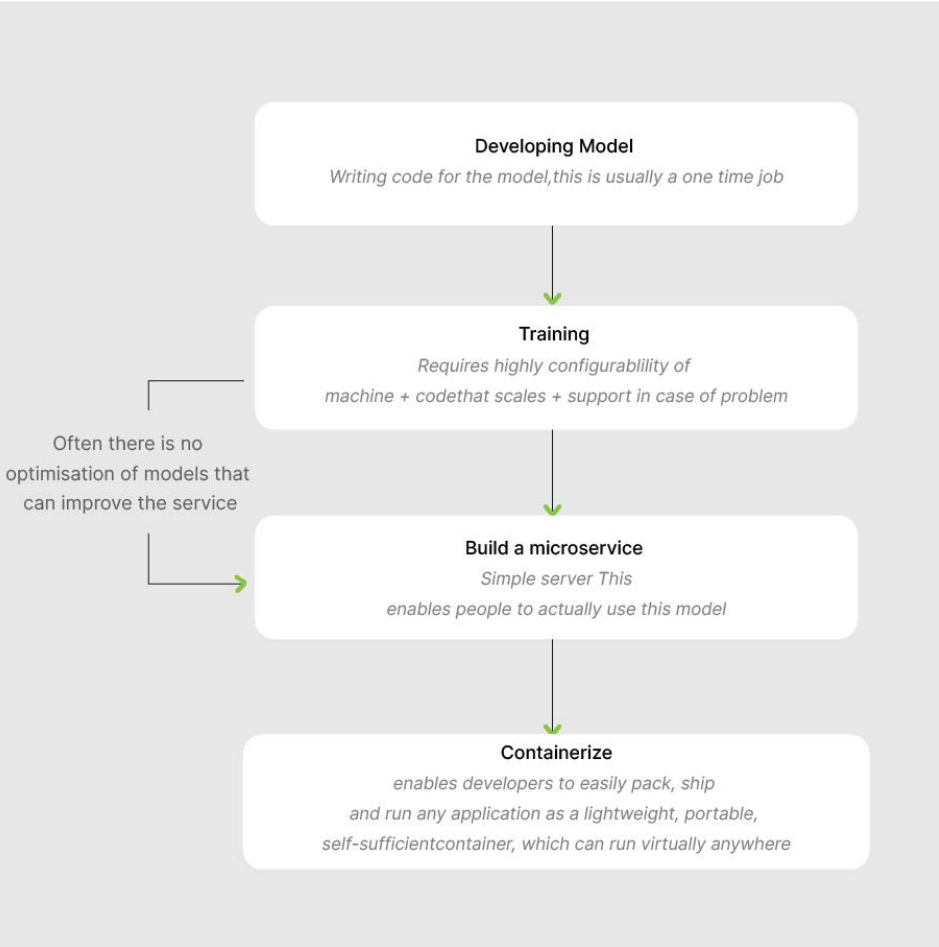
For Data Engineering teams:

Once your databases are finalized and ingestion pipelines are set, you should ideally be able to fetch all the data you need. At this point, you will need to convert this data (typically in JSON, CSV, API formats) into something that your model can understand. Machine learning is mathematically treated as "(x,f,y)" triplets where x is the input," y" is the output, and" f" is the function that is trained with the objective" f(x)~y". So keeping this in mind is a good guiding principle.

Converting the human-readable data into something that your model can consume is called **data transformation**. Some examples of data transformation are image cropping and resizing, text tokenization, etc.

Having some data visualization pipeline is a good idea whether you use a Jupyter Notebook or a wholly managed Grafana dashboard.

Model Development



Today, model development is akin to alchemy (the chemistry of the middle ages). Models are only as well-understood as the person explaining them — it is still unclear why or how they work. Furthermore, unlike conventional computer programs, debugging models may entail putting more data through it or performing dedicated feature engineering (a somewhat ephemeral practice).

The effort required in researching and developing a new model version is more than simply throwing more computational power at the problem, resulting in organizations being forced to continue implementing a sub-par solution. Something that would have taken a couple of hours to complete usually takes more than 54 hours. Often

overlooked until a crisis hits, the pressure to add new features adds to it.

Roughly, you can break down all the ML problems into a table like this:

	Stateless two same calls made successively return same thing	**Stateful** two same calls made successively return two different things
Causal Generation, Control	Fixed data modeling like text generation	Dynamic system like social network modeling
Non-Causal Classification, Prediction	Classification, Segmentation	Recommendation Algorithms

Stateless

When something does not have any change in behavior over successive calls, we can call it stateless.
This is the general property found in a majority of machine learning models. So, for example, a model that classifies a transaction as `fraud` should not tag something as `valid` later on.

Stateful

When the behaviour changes with every call, we call it a stateful model.
This is the general property in recommendation algorithms — ranging from items on an e-commerce platform or videos on streaming sites. Every successive call shows you a different set of items. Debugging these models is especially hard because you have to let the system run or create a dedicated pipeline for testing.

Causality

When you want your models to take a particular action (control in terms of reinforcement learning or generation, loosely), models take those actions based on what they have seen previously.
For example, consider a chess agent looking at a board and making a move. Of course, the mathematics around this may be a little deceiving, i.e., models not being able to understand that the current board is in a particular position because of its actions. But, in general, this domain is good whenever you try to forecast or predict the future.

Non-Causal

When the model does not have the property of causality (cause and effect). For example, a fraud transaction classifier does not need to forecast anything. Instead, it needs to look at a transaction and tell what it thinks.
Some other dimensions to split a model across are:

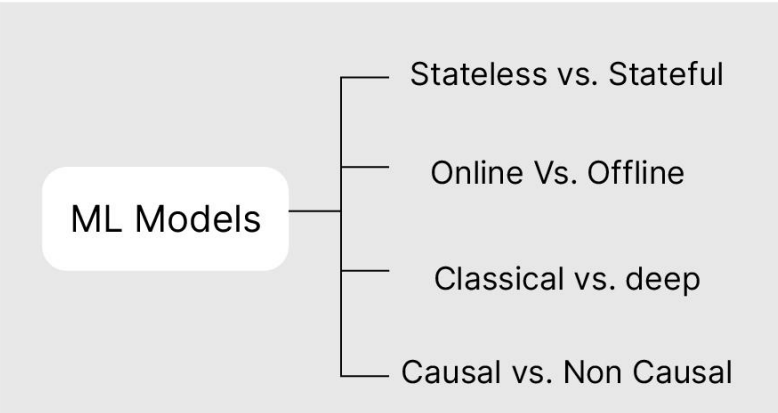
Online vs. Offline models

Models that are pinned online and are available at some IP address are known as online models. They are usually serving API endpoints but can be a remote procedure call. For example, a recommendation algorithm needs to be perennially available and thus is an example of online models.
Offline models (batched models) are usually loaded when the data has to be inferred. They are typically put up with other big data pipelines like Hadoop Clusters. For example, demand forecasting is an offline model since you can choose when you want to see the forecast models.

Classical vs. Deep Learning models

Classical models (CMLs) use statistical analysis or probabilistic models to satisfy an objective. They are often an excellent first step to creating a baseline and automating specific tasks where the data is sufficient.

Deep Learning models use neural networks to satisfy an objective. They are often difficult to explain and require an understanding of the basics.



To summarise, these are the four considerations to make when building an ML model for your use case. This will determine your debugging strategy and infrastructure.

Currently, AutoML is not at the stage where it can train random things. However, recent improvements in NLP like GPT/BERT/T5 have a considerable role in the next generation of AutoML. Unsupervised training will enable any digital data to be compressed and labeled, thus enabling powerful general-purpose performance.

Your objective for developing a machine learning model should be to achieve metrics like accuracy, the number of clicks to open links or churn rate. You've determined this metric in the first step of solving using machine learning, i.e., defining your business use-case (or machine learning problem).

The model should be explainable either directly through its features or indirectly through its behavior. It would help if you also optimized the model. Often, models consume three times more compute resources than required.

The final objective is to serve the model using asgi/wsgi and a web server (uvicorn, for example).

Most of the code is written to support the model in machine learning. However, developers rarely code the model. Instead, the primary job is writing efficient code for pre-and post-processing and wrapping it in business logic. Often, models are written only once in their lifetime, and new code typically means an entirely new model.

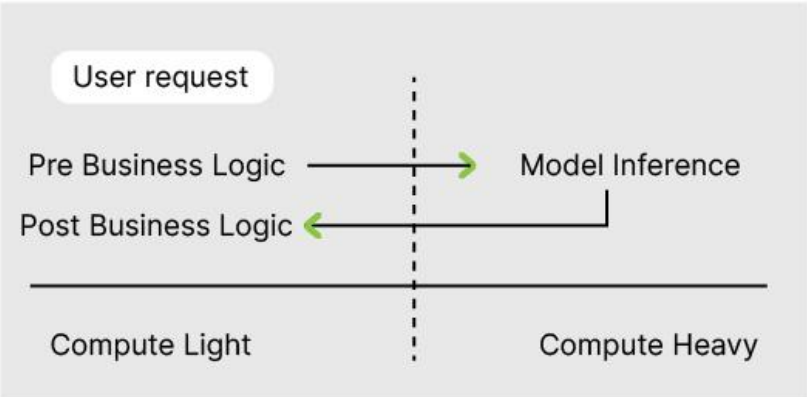
Writing the first set of models in CML is a great way to get started — for them to be shipped and the first set of experiences to be captured. It also helps refine the feature extraction process through code and make changes in the backend database, if required.

ML engineers and data scientists often use Jupyter or VSCode to create models that go into production. You can simplify and automate using scripts, extensions, or plugins for Jupyter. Ideally, it would be best to have your process sorted when moving towards a repetitive task. Avoid hardcoding values wherever possible, and use simple configurations. Optimizing for low lines of code is also an effective strategy to maintain flexibility.

Building an evident and good validation set is essential to understand the chosen features and the problem being solved. In addition, you must understand the validation set so well that you can determine the exact case when the model fails. This will also guide your development process.

Training a model is ultimately a hardware scaling problem. The better machines you have, the faster your training and validation will happen, and the faster you can get it into production. Therefore, throw more compute resources at the problem than attempting a more innovative solution. While it may seem counterproductive, it is truly "The Bitter Lesson" (search this up online!).

With that said, the cost is an essential factor. So be wise about spending — You and your team must operate large computers or any expensive machine once the engineering is better.

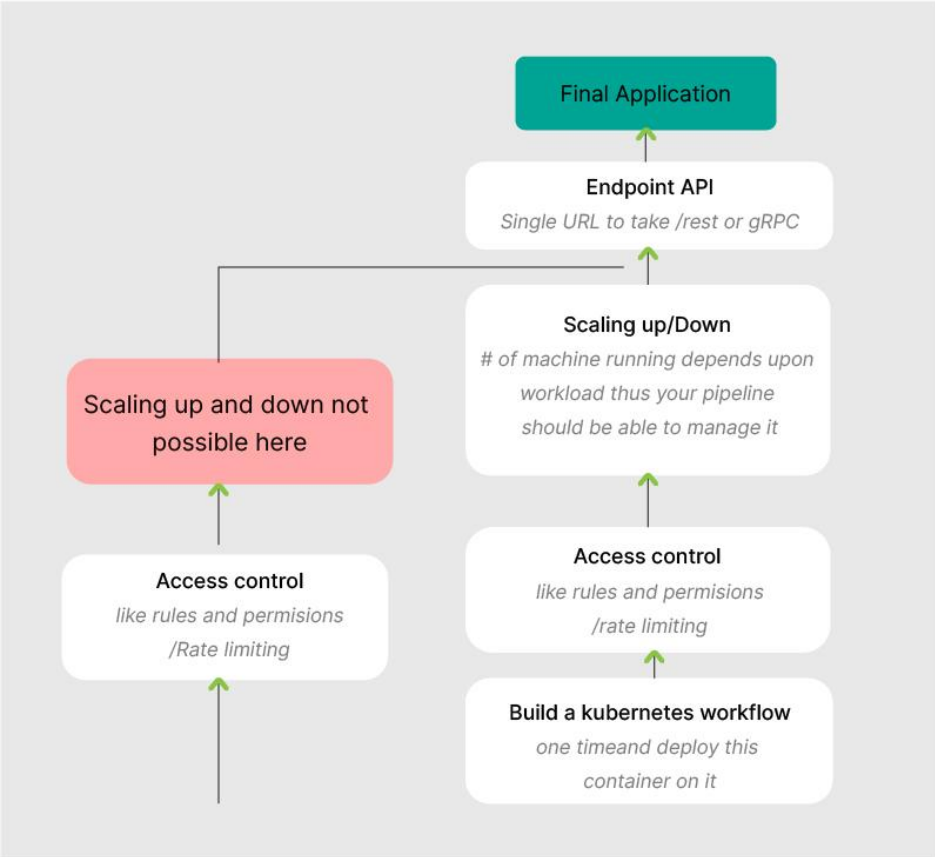


Additionally, try to keep business logic separate from the actual model execution as much as possible. The model inference is a compute-heavy problem and thus should be placed as such hardware-wise. Implementing this separation at the code level can reap faster development speeds.

In machine learning, the primary objective is to iterate over as many ideas as possible in the shortest timespan possible. Using the right tool for the job and optimizing the iteration speed is the key.This means automatic exploratory data analysis (EDA) for any kind of data, basic CML algorithms already generated before the user starts working.

Once it is ready, you will need to wrap the model in a server and reduce the overhead around it.

Closing the Loop with Monitoring



Monitoring is the most engineering-heavy aspect for any organization since it's a user-facing part. In the era of microservices, everything has to be an API endpoint. This, however, is easier said than done. The ultimate objective of monitoring is to capture better data over time and fix problems in the machine learning model, similar to fixing bugs in a giant codebase. The proximate purpose is backtesting.

Building and maintaining a Kubernetes cluster is relatively tedious and time-consuming. Moreover, there are always improvements that can be made — using Golang instead of Python or Rust instead of Java. Each model is deployed as an autoscaling pod on the network with proper load balancing. So is it relatively more straightforward for ML applications?

Unfortunately not. You not only want to serve requests but also store and analyze them, which requires a large team to build and maintain. You also need to provide a reliable service without any deviation in performance.

At this point, you have a model containerized and ready to be put into the network. Unfortunately, the intricacies of Kubernetes and DevOps are beyond this guide's scope, and hopefully, you may have experienced their complexities first-hand yourself.

A good deployment consists of the following (in increasing order of complexity to implement):

1. **Continuous Integration/Continuous Deployment (CI/CD)**: To produce the model, you must have a promising CI/CD pipeline set up to quickly revert to the previous model. It is not uncommon for social media platforms to revert their models hours after a big release.
2. **Monitoring**: The ability to watch and change the behavior of the models is as powerful as watching the messages and API requests. These two factors can explain the conduct of any model and aid your team in debugging.
3. **A/B Testing**: A powerful, hands-on approach to running the network and over service is A/B testing. Social media companies mentioned earlier can detect erroneous behavior because of their robust A/B testing pipelines.
4. **CI/CD with humans-in-loop**: Encompassing the ability to test the model on live samples before putting them in production helps improve in-the-wild errors and is a great way to test

models on known problems. Self-driving machine learning models are robustly checked against each known issue and manual human validation.

The above list is not exhaustive. However, at the current pace of technological innovation, improvisations with better tools and self-understanding are not far-fetched.

Extra: Deep Learning Models

There is no problem discovered yet that cannot be solved using a sufficiently large neural network. This represents a different ball game compared to things before it and thus justifies writing a whole section on it. First, however, we need to discuss classical machine learning (CML) more before explaining deep learning.

CML models rely on a statistical analysis of features (often human-built) and are easy to explain and hard to improve because of their limited processing ability.

At the core of machine learning sits the idea of "transforming vectors of features to get some other features." In CML, these transformations are calculated based on some fixed algorithm that looks at the data and fits it to an algorithm. For example, in the largest repository of CML algorithms (scikit-learn), this is what you get:

1. **Classification:** Identifying which category an object belongs to. It is used for spam detection or image recognition.
2. **Regression**: Predicting a continuous-valued attribute associated with an object, typically used for drug response or stock prices.
3. **Clustering**: Automatic grouping of similar objects into sets. It is used for customer segmentation or grouping experiment outcomes.

4. **Dimensionality Reduction**: Reducing the number of random variables to consider. It is used for visualization or increased efficiency.
5. **Model Selection**: Comparing, validating, and choosing parameters and models. It is used for improved accuracy via parameter tuning.
6. **Preprocessing**: Feature extraction and normalization.

CML is good when you want to quickly create a good enough baseline that can be shipped to get the first set of data and experience right. The tooling around the explainability of models is also a significant advantage that you can use to identify why a model behaves the way it does and for debugging and feature improvement.

Neural networks are dated to 1943, when McCulloch and Pitts introduced the first mathematical model. There is a lot of new research on this topic but comparing it to CML:

1. **Versatility**: Deep Learning Models (DLM) are relatively versatile and can handle any data (both modality and structure-wise) with new architectures like transformers — powering new ideas and products like GPT-3 from OpenAI, Github Copilot, Tesla's AutoPilot, or Deepmind AlphaFold.
2. **Size**: DLMs are considerably larger than classical models, resulting in higher latencies and expensive machines. Frameworks like ONNX, onnxruntime, Intel OpenVINO™ are enabling huge improvements from the software side, while platforms like NVIDIA CUDA are improving from the hardware perspective.

3. **Tooling**: Two well-known tools with a substantial open-source following, backed by Meta and Google, respectively, are PyTorch and TensorFlow. Pytorch is simple to use, has both: functional and OOPs-based APIs, while TensorFlow has a tremendous overall ecosystem with TFX.
4. **Learning Capacity**: Large DLMs, where the model size is significantly greater than the dataset size, have a reasonably large learning capacity to learn complicated structures quickly, though training large networks is complex.
5. **Explainability**: This is the Achilles' heel of DLMs, where they often appear to be doing foolish things because of unknown reasons, the reason for which they aren't not recommended for production use-cases with critical applications.
   However, having a promising MLOps pipeline can enable you to use deep learning models in production, such as Tesla's AutoPilot.

In general, use DLM when you:

1. Want to build a compelling and versatile model
2. Have a lot of data
3. Are unable to get more performance gains from CML
4. Require personalization.

MLOps Cheatsheet

1. **Defining the business use-case/ML problem**

   Objective: Coming up with clear definitions and numbers you want to achieve

   Do's:

   - Have a precise success criterion by creating a final specification.
   - Add ML to processes that are built — with options to fallback or compare against.
   - The metrics of success should be linked to your business rather than the problem.
   - Involve all stakeholders to have a concise understanding of the expected outcome.

2. **Building your data pipeline**

   Objective: Having precise control over data + ability to fetch anything with minimal code.

   Do's:

   - Set up data lakes, ingestion pipelines, and a process for data transformation.
   - Ensure the data engineering team understands concepts to operate independently of DevOps team.

3. **Model development**

   Objective: Creating an explainable and optimized model that caters to a business objective, achieving a specific metric like accuracy, churn rate, etc.

   Do's:

   - Write the first set of models in classical machine learning (CML) to ship instantly.
   - Build a very clear and good validation set.
   - Throw more compute resources at the problem than attempting a smarter solution. (Be mindful of the cost incurred!)
   - Build a quick proof-of-concept, automate the training, avoid hardcoded values, and optimize for low lines of code to maintain flexibility.
   - Avoid focusing on model specifics, leading to getting lost in the math.
   - Avoid large computational resources until engineering is sound.

4. **Monitoring → Data again**

   Objective: Capturing better data over time and fixing problems in the model while providing reliable service without too much deviation in performance.

   Do's:

   - Deploy the model without concerning about data logging — you may develop this alongside.
   - Create a centralized set of panels that can help you monitor more and better metrics.

NimbleBox.ai

NimbleBox.ai is a full-stack MLOps platform designed to help data scientists and machine learning practitioners around the world discover, create, and launch multi-cloud applications on an intuitive browser-based platform.

The platform is purpose-built for large datasets and supports all major machine learning frameworks, enabling developers to train and deploy at scale while maintaining accuracy.