CHAPTER 5
Preparing for Production

Joachim Zentici

Confirming that something works in the laboratory has never been a sure sign it will work well in the real world, and machine learning models are no different. Not only is the production environment typically very different from the development environment, but the commercial risks associated with models in production are much greater. It is important that the complexities of the transition to production are understood and tested and that the potential risks have been adequately mitigated.

This chapter explores the steps required to prepare for production (highlighted in the context of the entire life cycle in Figure 5-1). The goal is to illustrate, by extension, the elements that must be considered for robust MLOps systems.

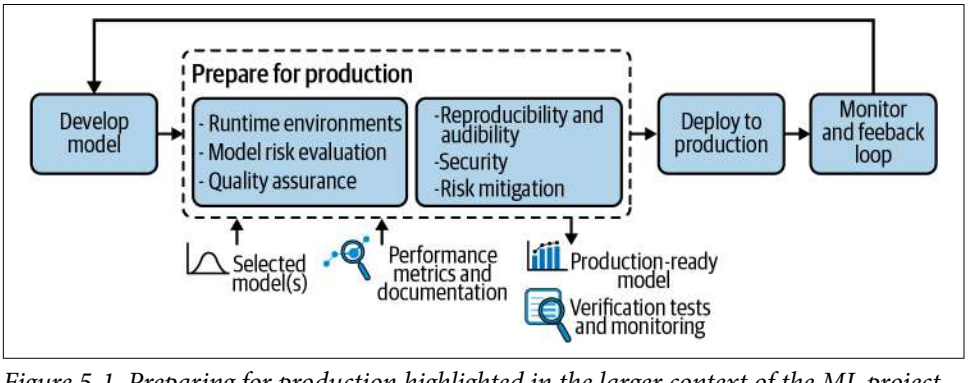

Figure 5-1. Preparing for production highlighted in the larger context of the ML project life cycle

Runtime Environments

The first step in sending a model to production is making sure it's technically possible. As discussed in Chapter 3, ideal MLOps systems favor rapid, automated deployment over labor-intensive processes, and runtime environments can have a big effect on which approach prevails.

Production environments take a wide variety of forms: custom-built services, data science platforms, dedicated services like TensorFlow Serving, low-level infrastructure like Kubernetes clusters, JVMs on embedded systems, etc. To make things even more complex, consider that in some organizations, multiple heterogeneous production environments coexist.

Ideally, models running in the development environment would be validated and sent as is to production; this minimizes the amount of adaptation work and improves the chances that the model in production will behave as it did in development. Unfortunately, this ideal scenario is not always possible, and it's not unheard of that teams finish a long-term project only to realize it can't be put in production.

Adaptation from Development to Production Environments

In terms of adaptation work, on one end of the spectrum, the development and production platforms are from the same vendor or are otherwise interoperable, and the dev model can run without any modification in production. In this case, the technical steps required to push the model into production are reduced to a few clicks or commands, and all efforts can be focused on validation.

On the other end of the spectrum, there are cases where the model needs to be reimplemented from scratch—possibly by another team, and possibly in another programming language. Given the resources and time required, there are few cases today where this approach makes sense. However, it's still the reality in many organizations and is often a consequence of the lack of appropriate tooling and processes. The reality is that handing over a model for another team to reimplement and adapt for the production environment means that model won't reach production for months (maybe years), if at all.

Between these two extreme cases, there can be a number of transformations performed on the model or the interactions with its environment to make it compatible with production. In all cases, it is crucial to perform validation in an environment that mimics production as closely as possible, rather than in the development environment.

Tooling considerations

The format required to send to production should be considered early, as it may have a large impact on the model itself and the quantity of work required to productionalize it. For example, when a model is developed using scikit-learn (Python) and production is a Java-based environment that expects PMML or ONNX as input, conversion is obviously required.

In this case, teams should set up tooling while developing the model, ideally before the first version of the model is finished or even started. Failure to create this pipeline up front would block the validation process (and, of course, final validation should not be performed on the scikit-learn model, as it's not the one that will be put into production).

Performance considerations

Another common reason conversion may be required is for performance. For example, a Python model will typically have higher latency for scoring a single record than its equivalent converted to C++. The resulting model will likely be dozens of times faster (although obviously it depends on many factors, and the result can also be a model that is dozens of times slower).

Performance also comes into play when the production model must run on a low-power device. In the specific case of deep neural networks, for example, trained models can become extremely large with billions or hundreds of billions of parameters. Running them on small devices is simply impossible, and running them on standard servers can be slow and expensive.

For these models, an optimized runtime is not enough. To obtain better performance, the model definition must be optimized. One solution is to use compression techniques:

- With quantization, the model can be trained using 32-bit floating-point numbers and used for inference at a lower precision so that the model requires less memory and is faster while accuracy is mostly preserved.
- With pruning, one simply removes weights (or even entire layers) from the neural network. This is a rather radical approach, but some methods allow for the preservation of accuracy.
- With distillation, a smaller "student" network is trained to mimic a bigger, more powerful network. Done appropriately, this can lead to better models (as compared to trying to train the smaller network directly from the data).

These methods are efficient if the initial model is trained in a way that reduces information loss while performing them, so these operations are not simply conversions of the trained model post hoc, but rather orient the way the model is trained. These

methods are still very recent and quite advanced but already commonly used in natural language processing (NLP) pretrained models.

Data Access Before Validation and Launch to Production

Another technical aspect that needs to be addressed before validation and launch to production is data access. For example, a model evaluating apartment prices may use the average market price in a zip code area; however, the user or the system requesting the scoring will probably not provide this average and would most likely provide simply the zip code, meaning a lookup is necessary to fetch the value of the average.

In some cases, data can be frozen and bundled with the model. But when this is not possible (e.g., if the dataset is too large or the enrichment data needs to always be up to date), the production environment should access a database and thus have the appropriate network connectivity, libraries, or drivers required to communicate with the data storage installed, and authentication credentials stored in some form of production configuration.

Managing this setup and configuration can be quite complex in practice since, again, it requires appropriate tooling and collaboration (in particular to scale to more than a few dozen models). When using external data access, model validation in situations that closely match production is even more critical as technical connectivity is a common source of production malfunction.

Final Thoughts on Runtime Environments

Training a model is usually the most impressive computation, requiring a high level of software sophistication, massive data volumes, and high-end machines with powerful GPUs. But in the whole life cycle of a model, there is a good chance that most of the compute is spent at inference time (even if this computation is orders of magnitude simpler and faster). This is because a model is trained once and can be used billions of times for inference.

Scaling inference on complex models can be expensive and have significant energy and environmental impact. Lowering the complexity of models or compressing extremely complex models can lower the infrastructure cost of operating machine learning models.

It's important to remember that not all applications require deep learning, and in fact, not all applications require machine learning at all. A valuable practice to control complexity in production is to develop complex models only to provide a baseline for what seems achievable. What goes into production can then be a much simpler model, with the advantages of lowering the operating risk, increasing computational performance, and lowering power consumption. If the simple model is close enough

to the high complexity baseline, then it can be a much more desirable solution for production.

Model Risk Evaluation

Before exploring how validation should be done in an ideal MLOps system, it's important to consider the purpose of validation. As discussed in Chapter 4, models attempt to mimic reality, but they are imperfect; their implementation can have bugs, as can the environment they are executing in. The indirect, real-world impact a model in production can have is never certain, and the malfunctioning of a seemingly insignificant cog can have tremendous consequences in a complex system.

The Purpose of Model Validation

It is, to some extent, possible (not to mention absolutely necessary) to anticipate the risks of models in production and thus design and validate so as to minimize these risks. As organizations become more and more complex, it is essential to understand that involuntary malfunctions or malicious attacks are potentially threatening in most uses of machine learning in the enterprise, not only in financial or safety-related applications.

Before putting a model in production (and in fact constantly from the beginning of the machine learning project), teams should ask the uncomfortable questions:

- What if the model acts in the worst imaginable way?
- What if a user manages to extract the training data or the internal logic of the model?
- What are the financial, business, legal, safety, and reputational risks?

For high-risk applications, it is essential that the whole team (and in particular the engineers in charge of validation) be fully aware of these risks so that they can design the validation process appropriately and apply the strictness and complexity appropriate for the magnitude of the risks.

In many ways, machine learning risk management covers model risk management practices that are well established in many industries, such as banking and insurance. However, machine learning introduces new types of risks and liabilities, and as data science gets democratized, it involves many new organizations or teams that have no experience with more traditional model risk management.

The Origins of ML Model Risk

The magnitude of risk ML models can bring is hard to model for mathematical reasons, but also because the materialization of risks arises through real-world consequences. The ML metrics, and in particular the cost matrix, allow teams to evaluate the average cost of operating a model in its "nominal" case, meaning on its cross-validation data, compared to operating a perfect magical model.

But while computing this expected cost can be very important, a wide range of things can go wrong well beyond expected cost. In some applications, the risk can be a financially unbounded liability, a safety issue for individuals, or an existential threat for the organization. ML model risk originates essentially from:

- Bugs, errors in designing, training, or evaluating the model (including data prep)
- Bugs in the runtime framework, bugs in the model post-processing/conversion, or hidden incompatibilities between the model and its runtime
- Low quality of training data
- High difference between production data and training data
- Expected error rates, but with failures that have higher consequences than expected
- Misuse of the model or misinterpretation of its outputs
- Adversarial attacks
- Legal risk originating in particular from copyright infringement or liability for the model output
- Reputational risk due to bias, unethical use of machine learning, etc.

The probability of materialization of the risk and its magnitude can be amplified by:

- Broad use of the model
- A rapidly changing environment
- Complex interactions between models

The following sections provide more details on these threats and how to mitigate them, which should ultimately be the goal of any MLOps system the organization puts in place.

Quality Assurance for Machine Learning

Software engineering has developed a mature set of tools and methodologies for quality assurance (QA), but the equivalent for data and models is still in its infancy, which makes it challenging to incorporate into MLOps processes. The statistical methods as

well as documentation best practices are well known, but implementing them at scale is not common.

Though it's being covered as a part of this chapter on preparing for production, to be clear, QA for machine learning does not occur only at the final validation stage; rather, it should accompany all stages of model development. Its purpose is to ensure compliance with processes as well as ML and computational performance requirements, with a level of detail that is proportionate to the level of risk.

In the case where the people in charge of validation are not the ones who developed the model, it is essential that they have enough training in machine learning and understand the risks so that they can design appropriate validation or detect breaches in the validation proposed by the development team. It is also essential that the organization's structure and culture give them the authority to appropriately report issues and contribute to continuous improvement or block passage to production if the level of risk justifies it.

Robust MLOps practices dictate that performing QA before sending to production is not only about technical validation. It is also the occasion to create documentation and validate the model against organizational guidelines. In particular, this means the origin of all input datasets, pretrained models, or other assets should be known, as they could be subject to regulations or copyrights. For this reason (and for computer security reasons in particular), some organizations choose to allow only whitelisted dependencies. While this can significantly impact the ability of data scientists to innovate quickly, though the list of dependencies can be reported and checked partly automatically, it can also provide additional safety.

Key Testing Considerations

Obviously, model testing will consist of applying the model to carefully curated data and validating measurements against requirements. How the data is selected or generated as well as how much data is required is crucial, but it will depend on the problem tackled by the model.

There are some scenarios in which the test data should not always match "real-world" data. For example, it can be a good idea to prepare a certain number of scenarios, and while some of them should match realistic situations, other data should be specifically generated in ways that could be problematic (e.g., extreme values, missing values).

Metrics must be collected on both statistical (accuracy, precision, recall, etc.) as well as computational (average latency, 95th latency percentile, etc.) aspects, and the test scenarios should fail if some assumptions on them are not verified. For example, the test should fail if the accuracy of the model falls below 90%, the average inference time goes above 100 milliseconds, or more than 5% of inferences take more than 200

milliseconds. These assumptions can also be called *expectations*, *checks*, or *assertions*, as in traditional software engineering.

Statistical tests on results can also be performed but are typically used for subpopulations. It is also important to be able to compare the model with its previous version. It can allow putting in place a champion/challenger approach (described in detail in "Champion/Challenger" on page 100) or checking that a metric does not suddenly drop.

Subpopulation Analysis and Model Fairness

It can be useful to design test scenarios by splitting data into subpopulations based on a "sensitive" variable (that may or may not be used as a feature of the model). This is how fairness (typically between genders) is evaluated.

Virtually all models that apply to people should be analyzed for fairness. Increasingly, failure to assess model fairness will have business, regulatory, and reputational implications for organizations. For details about biases and fairness, refer to "Impact of Responsible AI on Modeling" on page 53 and "Key Elements of Responsible AI" on page 113.

In addition to validating the ML and computational performance metrics, model stability is an important testing property to consider. When changing one feature slightly, one expects small changes in the outcome. While this cannot be always true, it is generally a desirable model property. A very unstable model introduces a lot of complexity and loopholes in addition to delivering a frustrating experience, as the model can feel unreliable even if it has decent performance. There is no single answer to model stability, but generally speaking, simpler models or more regularized ones show better stability.

Reproducibility and Auditability

Reproducibility in MLOps does not have the same meaning as in academia. In the academic world, reproducibility essentially means that the findings of an experiment are described well enough that another competent person can replicate the experiment using the explanations alone, and if the person doesn't make any mistakes, they will arrive at the same conclusion.

In general, reproducibility in MLOps also involves the ability to easily rerun the exact same experiment. It implies that the model comes with detailed documentation, the data used for training and testing, and with an artifact that bundles the implementation of the model plus the full specification of the environment it was run in (see

"Version Management and Reproducibility" on page 56). Reproducibility is essential to prove model findings, but also to debug or build on a previous experiment.

Auditability is related to reproducibility, but it adds some requirements. For a model to be auditable, it must be possible to access the full history of the ML pipeline from a central and reliable storage and to easily fetch metadata on all model versions including:

- The full documentation
- An artifact that allows running the model with its exact initial environment
- Test results, including model explanations and fairness reports
- Detailed model logs and monitoring metadata

Auditability can be an obligation in some highly regulated applications, but it has benefits for all organizations because it can facilitate model debugging, continuous improvement, and keeping track of actions and responsibilities (which is an essential part of governance for responsible applications of ML, as discussed at length in Chapter 8). A full QA toolchain for machine learning—and, thus, MLOps processes—should provide a clear view of model performance with regard to requirements while also facilitating auditability.

Even when MLOps frameworks allow data scientists (or others) to find a model with all its metadata, understanding the model itself can still be challenging (see "Impact of Responsible AI on Modeling" on page 53 for a detailed discussion).

To have a strong practical impact, auditability must allow for intuitive human understanding of all the parts of the system and their version histories. This doesn't change the fact that understanding a machine learning model (even a relatively simple one) requires appropriate training, but depending on the criticality of the application, a wider audience may need to be able to understand the details of the model. As a result, full auditability comes at a cost that should be balanced with the criticality of the model itself.

Machine Learning Security

As a piece of software, a deployed model running in its serving framework can present multiple security issues that range from low-level glitches to social engineering. Machine learning introduces a new range of potential threats where an attacker provides malicious data designed to cause the model to make a mistake.

There are numerous cases of potential attacks. For example, spam filters were an early application of machine learning essentially based on scoring words that were in a dictionary. One way for spam creators to avoid detection was to avoid writing these exact words while still making their message easily understandable by a human

reader (e.g., using exotic Unicode characters, voluntarily introducing typos, or using images).

Adversarial Attacks

A more modern but quite analogous example of a machine learning model security issue is an adversarial attack for deep neural networks in which an image modification that can seem minor or even impossible for a human eye to notice can cause the model to drastically change its prediction. The core idea is mathematically relatively simple: since deep learning inference is essentially matrix multiplication, carefully chosen small perturbations to coefficients can cause a large change in the output numbers.

One example of this is that small stickers glued to road signs can confuse an autonomous car's computer vision system, rendering signs invisible or incorrectly classified by the system, while remaining fully visible and understandable to a human being. The more the attacker knows about the system, the more likely they are to find examples that will confuse it.

A human can use reason to find these examples (in particular for simple models). However, for more complex models like deep learning, the attacker will probably need to perform many queries and either use brute force to test as many combinations as possible or use a model to search for problematic examples. The difficulty of countermeasures is increasing with the complexity of models and their availability. Simple models such as logistic regressions are essentially immune, while an open source pretrained deep neural network will basically always be vulnerable, even with advanced, built-in attack detectors.

Adversarial attacks don't necessarily happen at inference time. If an attacker can get access to the training data, even partially, then they get control over the system. This kind of attack is traditionally known as a *poisoning attack* in computer security.

One famous example is the Twitter chatbot released by Microsoft in 2016. Just a few hours after launch, the bot started to generate very offensive tweets. This was caused by the bot adapting to its input; when realizing that some users submitted a large amount of offensive content, the bot started to replicate. In theory, a poisoning attack can occur as a result of an intrusion or even, in a more sophisticated way, through pretrained models. But in practice, one should mostly care about data collected from easily manipulated data sources. Tweets sent to a specific account are a particularly clear example.

Other Vulnerabilities

Some patterns do not exploit machine learning vulnerabilities per se, but they do use the machine learning model in ways that lead to undesirable situations. One example

is in credit scoring: for a given amount of money, borrowers with less flexibility tend to choose a longer period to lower the payments, while borrowers who are not concerned about their ability to pay may choose a shorter period to lower the total cost of credit. Salespeople may advise those who do not have a good enough score to shorten their payments. This increases the risk for the borrower *and* the bank and is not a meaningful course of action. Correlation is not causality!

Models can also leak data in many ways. Since the machine learning models can fundamentally be considered a summary of the data they have been trained on, they can leak more or less precise information on the training data, up to the full training set in some cases. Imagine, for example, that a model predicts how much someone is paid using the nearest neighbor algorithm. If one knows the zip code, age, and profession of a certain person registered on the service, it's pretty easy to obtain that person's exact income. There are a wide range of attacks that can extract information from models in this way.

In addition to technical hardening and audit, governance plays a critical role in security. Responsibilities must be assigned clearly and in a way that ensures an appropriate balance between security and capacity of execution. It is also important to put in place feedback mechanisms, and employees and users should have an easy channel to communicate breaches (including, potentially, "bug bounty programs" that reward reporting vulnerabilities). It is also possible, and necessary, to build safety nets around the system to mitigate the risks.

Machine learning security shares many common traits with general computer system security, one of the main ideas being that security is not an additional independent feature of the system; that is, generally you cannot secure a system that is not designed to be secure, and the organization processes must take into account the nature of the threat from the beginning. Strong MLOps processes, including all of the steps in preparing for production described in this chapter, can help make this approach a reality.

Model Risk Mitigation

Generally speaking, as discussed in detail in Chapter 1, the broader the model deployment, the greater the risk. When risk impact is high enough, it is essential to control the deployment of new versions, which is where tightly controlled MLOps processes come into play in particular. Progressive or canary rollouts should be a common practice, with new versions of models being served to a small proportion of the organization or customer base first and slowly increasing that proportion, while monitoring behavior and getting human feedback if appropriate.

Changing Environments

Rapidly changing environments also multiply risk, as mentioned earlier in this chapter. Changes in inputs is a related and also well-identified risk, and Chapter 7 dives into these challenges and how to address them in more detail. But what's important to note is that the speed of change can amplify the risk depending on the application. Changes may be so fast that they have consequences even before the monitoring system sends alerts. That is to say, even with an efficient monitoring system and a procedure to retrain models, the time necessary to remediate may be a critical threat, especially if simply retraining the model on new data is not sufficient and a new model must be developed. During this time, the production systems misbehaving can cause large losses for the organization.

To control this risk, monitoring via MLOps should be reactive enough (typically, alerting on distributions computed every week might not be enough), and the procedure should consider the period necessary for remediation. For example, in addition to retraining or rollout strategies, the procedure may define thresholds that would trigger a degraded mode for the system. A degraded mode may simply consist of a warning message displayed for end users, but could be as drastic as shutting down the dysfunctional system to avoid harm until a stable solution can be deployed.

Less dramatic issues that are frequent enough can also do harm that quickly becomes difficult to control. If the environment changes often, even if remediation never seems urgent, a model can always be slightly off, never operating within its nominal case, and the operating cost can be challenging to evaluate. This can only be detected through dedicated MLOps, including relatively long-term monitoring and reevaluating the cost of operating the model.

In many cases, retraining the model on more data will increasingly improve the model, and this problem will eventually disappear, but this can take time. Before this convergence, a solution might be to use a less complex model that may have a lower evaluated performance and may be more consistent in a frequently changing environment.

Interactions Between Models

Complex interactions between models is probably the most challenging source of risk. This class of issue will be a growing concern as ML models become pervasive, and it's an important potential area of focus for MLOps systems. Obviously, adding models will often add complexity to an organization, but the complexity does not necessarily grow linearly in proportion to the number of models; having two models is more complicated to understand than the sum since there are potential interactions between them.

Moreover, the total complexity is heavily determined by how the interactions with models are designed at a local scale and governed at an organizational scale. Using models in chains (where a model uses inputs from another model) can create significant additional complexity as well as totally unexpected results, whereas using models in independent parallel processing chains, which are each as short and explainable as possible, is a much more sustainable way to design large-scale deployment of machine learning.

First, the absence of obvious interactions between models makes the complexity grow closer to linearly (though note that, in practice, it is rarely the case, as there can always be interactions in the real world even if models are not connected). Also, models used in redundant chains of processing can avoid errors—that is, if a decision is based on several independent chains of processing with methods as different as possible, it can be more robust.

Finally, generally speaking, the more complex the model, the more complex its interactions with other systems may be, as it may have many edge cases, be less stability in some domains, overreact to the changes of an upstream model, or confuse a sensitive downstream model, etc. Here again, we see that model complexity has a cost, and a potentially highly unpredictable one at that.

Model Misbehavior

A number of measures can be implemented to avoid model misbehavior, including examining its inputs and outputs in real time. While training a model, it is possible to characterize its domain of applicability by examining the intervals on which the model was trained and validated. If the value of a feature at inference time is out of bounds, the system can trigger appropriate measures (e.g., rejecting the sample or dispatching a warning message).

Controlling feature-value intervals is a useful and simple technique, but it might be insufficient. For example, when training an algorithm to evaluate car prices, the data may have provided examples of recent light cars and old heavy cars, but no recent heavy cars. The performance of a complex model for these is unpredictable. When the number of features is large, this issue becomes unavoidable due to the curse of dimensionality—i.e., the number of combinations is exponential relative to the number of features.

In these situations, more sophisticated methods can be used, including anomaly detection to identify records where the model is used outside of its application domain. After scoring, the outputs of the model can be examined before confirming the inference. In the case of classification, many algorithms provide certainty scores in addition to their prediction, and a threshold can be fixed to accept an inference output. Note that these certainty scores do not typically translate into probabilities, even if they are named this way in the model.

Conformal prediction is a set of techniques that helps calibrate these scores to obtain an accurate estimation of the probability of correctness. For regression, the value can be checked against a predetermined interval. For example, if the model predicts a car costs $50 or $500,000, you may not want to commit any business on this prediction. The complexity of the implemented techniques should be relevant for the level of risk: a highly complex, highly critical model will require more thorough safeguards.

Closing Thoughts

In practice, preparing models for production starts from the beginning at the development phase; that is to say, the requirements of production deployments, security implications, and risk mitigation aspects should be considered when developing the models. MLOps includes having a clear validation step before sending models to production, and the key ideas to successfully prepare models for productions are:

- Clearly identifying the nature of the risks and their magnitudes
- Understanding model complexity and its impact at multiple levels, including increased latency, increased memory and power consumption, lower ability to interpret inference in production, and a harder-to-control risk
- Providing a simple but clear standard of quality, making sure the team is appropriately trained and the organization structure allows for fast and reliable validation processes
- Automating all the validation that can be automated to ensure it is properly and consistently performed while maintaining the ability to deploy quickly