CHAPTER 6

Deploying to Production

Joachim Zentici

Business leaders view the rapid deployment of new systems into production as key to maximizing business value. But this is only true if deployment can be done smoothly and at low risk (software deployment processes have become more automated and rigorous in recent years to address this inherent conflict). This chapter dives into the concepts and considerations when deploying machine learning models to production that impact—and indeed, drive—the way MLOps deployment processes are built (Figure 6-1 presents this phase in the context of the larger life cycle).

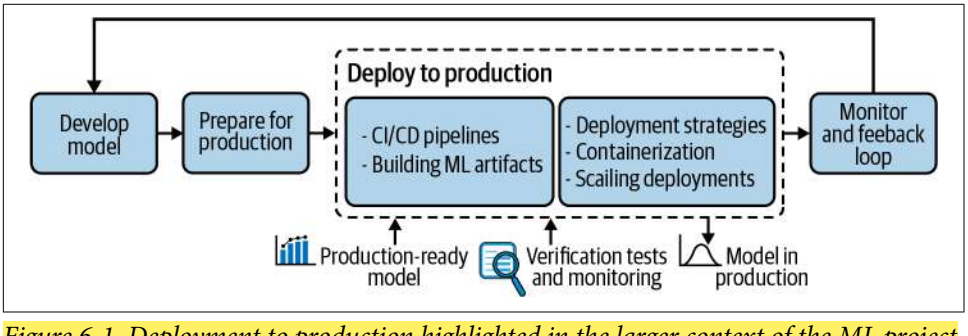

Figure 6-1. Deployment to production highlighted in the larger context of the ML project life cycle

CI/CD Pipelines

CI/CD is a common acronym for continuous integration and continuous delivery (or put more simply, deployment). The two form a modern philosophy of agile software development and a set of practices and tools to release applications more often and faster, while also better controlling quality and risk.

While these ideas are decades old and already used to various extents by software engineers, different people and organizations use certain terms in very different ways. Before digging into how CI/CD applies to machine learning workflows, it is essential to keep in mind that these concepts should be tools to serve the purpose of delivering quality fast, and the first step is always to identify the specific risks present at the organization. In other words, as always, CI/CD methodology should be adapted based on the needs of the team and the nature of the business.

CI/CD concepts apply to traditional software engineering, but they apply just as well to machine learning systems and are a critical part of MLOps strategy. After successfully developing a model, a data scientist should push the code, metadata, and documentation to a central repository and trigger a CI/CD pipeline. An example of such pipeline could be:

1. Build the model
   a. Build the model artifacts
   b. Send the artifacts to long-term storage
   c. Run basic checks (smoke tests/sanity checks)
   d. Generate fairness and explainability reports
2. Deploy to a test environment
   a. Run tests to validate ML performance, computational performance
   b. Validate manually
3. Deploy to production environment
   a. Deploy the model as canary
   b. Fully deploy the model

Many scenarios are possible and depend on the application, the risks from which the system should be protected, and the way the organization chooses to operate. Generally speaking, an incremental approach to building a CI/CD pipeline is preferred: a simple or even naïve workflow on which a team can iterate is often much better than starting with complex infrastructure from scratch.

A starting project does not have the infrastructure requirements of a tech giant, and it can be hard to know up front which challenges deployments will present. There are common tools and best practices, but there is no one-size-fits-all CI/CD methodology. This means the best path forward is starting from a simple (but fully functional) CI/CD workflow and introducing additional or more sophisticated steps along the way as quality or scaling challenges appear.

Building ML Artifacts

The goal of a continuous integration pipeline is to avoid unnecessary effort in merging the work from several contributors as well as to detect bugs or development conflicts as soon as possible. The very first step is using centralized version control systems (unfortunately, working for weeks on code stored only on a laptop is still quite common).

The most common version control system is Git, an open source software initially developed to manage the source code for the Linux kernel. The majority of software engineers across the world already use Git, and it is increasingly being adopted in scientific computing and data science. It allows for maintaining a clear history of changes, safe rollback to a previous version of the code, multiple contributors to work on their own branches of the project before merging to the main branch, etc.

While Git is appropriate for code, it was not designed to store other types of assets common in data science workflows, such as large binary files (for example, trained model weights), or to version the data itself. Data versioning is a more complex topic with numerous solutions, including Git extensions, file formats, databases, etc.

What's in an ML Artifact?

Once the code and data is in a centralized repository, a testable and deployable bundle of the project must be built. These bundles are usually called *artifacts* in the context of CI/CD. Each of the following elements needs to be bundled into an artifact that goes through a testing pipeline and is made available for deployment to production:

- Code for the model and its preprocessing
- Hyperparameters and configuration
- Training and validation data
- Trained model in its runnable form
- An environment including libraries with specific versions, environment variables, etc.
- Documentation
- Code and data for testing scenarios

The Testing Pipeline

As touched on in Chapter 5, the testing pipeline can validate a wide variety of properties of the model contained in the artifact. One of the important operational aspects

of testing is that, in addition to verifying compliance with requirements, good tests should make it as easy as possible to diagnose the source issue when they fail.

For that purpose, naming the tests is extremely important, and carefully choosing a number of datasets to validate the model against can be valuable. For example:

- A test on a fixed (not automatically updated) dataset with simple data and not-too-restrictive performance thresholds can be executed first and called "base case." If the test reports show that this test failed, there is a strong possibility that the model is way off, and the cause may be a programming error or a misuse of the model, for example.
- Then, a number of datasets that each have one specific oddity (missing values, extreme values, etc.) could be used with tests appropriately named so that the test report immediately shows the kind of data that is likely to make the model fail. These datasets can represent realistic yet remarkable cases, but it may also be useful to generate synthetic data that is not expected in production. This could possibly protect the model from new situations not yet encountered, but most importantly, this could protect the model from malfunctions in the system querying or from adversarial examples (as discussed in "Machine Learning Security" on page 67).
- Then, an essential part of model validation is testing on recent production data. One or several datasets should be used, extracted from several time windows and named appropriately. This category of tests should be performed and automatically analyzed when the model is already deployed to production. Chapter 7 provides more specific details on how to do that.

Automating these tests as much as possible is essential and, indeed, is a key component of efficient MLOps. A lack of automation or speed wastes time, but, more importantly, it discourages the development team from testing and deploying often, which can delay the discovery of bugs or design choices that make it impossible to deploy to production.

In extreme cases, a development team can hand over a monthslong project to a deployment team that will simply reject it because it does not satisfy requirements for the production infrastructure. Also, less frequent deployments imply larger increments that are harder to manage; when many changes are deployed at once and the system is not behaving in the desired way, isolating the origin of an issue is more time consuming.

The most widespread tool for software engineering continuous integration is Jenkins, a very flexible build system that allows for the building of CI/CD pipelines regardless of the programming language, testing framework, etc. Jenkins can be used in data science to orchestrate CI/CD pipelines, although there are many other options.

Deployment Strategies

To understand the details of a deployment pipeline, it is important to distinguish among concepts often used inconsistently or interchangeably.

Integration
: The process of merging a contribution to a central repository (typically merging a Git feature branch to the main branch) and performing more or less complex tests.

Delivery
: As used in the continuous delivery (CD) part of CI/CD, the process of building a fully packaged and validated version of the model ready to be deployed to production.

Deployment
: The process of running a new model version on a target infrastructure. Fully automated deployment is not always practical or desirable and is a business decision as much as a technical decision, whereas continuous delivery is a tool for the development team to improve productivity and quality as well as measure progress more reliably. Continuous delivery is required for continuous deployment, but it also provides enormous value without.

Release
: In principle, release is yet another step, as deploying a model version (even to the production infrastructure) does not necessarily mean that the production workload is directed to the new version. As we will see, multiple versions of a model can run at the same time on the production infrastructure.

Getting everyone in the MLOps process on the same page about what these concepts mean and how they apply will allow for smoother processes on both the technical and business sides.

Categories of Model Deployment

In addition to different deployment strategies, there are two ways to approach model deployment:

- Batch scoring, where whole datasets are processed using a model, such as in daily scheduled jobs.
- Real-time scoring, where one or a small number of records are scored, such as when an ad is displayed on a website and a user session is scored by models to decide what to display.

There is a continuum between these two approaches, and in fact, in some systems, scoring on one record is technically identical to requesting a batch of one. In both cases, multiple instances of the model can be deployed to increase throughput and potentially lower latency.

Deploying many real-time scoring systems is conceptually simpler since the records to be scored can be dispatched between several machines (e.g., using a load balancer). Batch scoring can also be parallelized, for example by using a parallel processing runtime like Apache Spark, but also by splitting datasets (which is usually called *partitioning* or *sharding*) and scoring the partitions independently. Note that these two concepts of splitting the data and computation can be combined, as they can address different problems.

Considerations When Sending Models to Production

When sending a new model version to production, the first consideration is often to avoid downtime, in particular for real-time scoring. The basic idea is that rather than shutting down the system, upgrading it, and then putting it back online, a new system can be set up next to the stable one, and when it's functional, the workload can be directed to the newly deployed version (and if it remains healthy, the old one is shut down). This deployment strategy is called *blue-green*—or sometimes *red-black*—deployment. There are many variations and frameworks (like Kubernetes) to handle this natively.

Another more advanced solution to mitigate the risk is to have canary releases (also called *canary deployments*). The idea is that the stable version of the model is kept in production, but a certain percentage of the workload is redirected to the new model, and results are monitored. This strategy is usually implemented for real-time scoring, but a version of it could also be considered for batch.

A number of computational performance and statistical tests can be performed to decide whether to fully switch to the new model, potentially in several workload percentage increments. This way, a malfunction would likely impact only a small portion of the workload.

Canary releases apply to production systems, so any malfunction is an incident, but the idea here is to limit the blast radius. Note that scoring queries that are handled by the canary model should be carefully picked, because some issues may go unnoticed otherwise. For example, if the canary model is serving a small percentage of a region or country before the model is fully released globally, it could be the case that (for machine learning or infrastructure reasons) the model does not perform as expected in other regions.

A more robust approach is to pick the portion of users served by the new model at random, but then it is often desirable for user experience to implement an affinity mechanism so that the same user always uses the same version of the model.

Canary testing can be used to carry out A/B testing, which is a process to compare two versions of an application in terms of a business performance metric. The two concepts are related but not the same, as they don't operate at the same level of abstraction. A/B testing can be made possible through a canary release, but it could also be implemented as logic directly coded into a single version of an application. Chapter 7 provides more details on the statistical aspects of setting up A/B testing.

Overall, canary releases are a powerful tool, but they require somewhat advanced tooling to manage the deployment, gather the metrics, specify and run computations on them, display the results, and dispatch and process alerts.

Maintenance in Production

Once a model is released, it must be maintained. At a high level, there are three maintenance measures:

Resource monitoring
: Just as for any application running on a server, collecting IT metrics such as CPU, memory, disk, or network usage can be useful to detect and troubleshoot issues.

Health check
: To check if the model is indeed online and to analyze its latency, it is common to implement a health check mechanism that simply queries the model at a fixed interval (on the order of one minute) and logs the results.

ML metrics monitoring
: This is about analyzing the accuracy of the model and comparing it to another version or detecting when it is going stale. Since it may require heavy computation, this is typically lower frequency, but as always, will depend on the application; it is typically done once a week. Chapter 7 details how to implement this feedback loop.

Finally, when a malfunction is detected, a rollback to a previous version may be necessary. It is critical to have the rollback procedure ready and as automated as possible; testing it regularly can make sure it is indeed functional.

Containerization

As described earlier, managing the versions of a model is much more than just saving its code into a version control system. In particular, it is necessary to provide an exact description of the environment (including, for example, all the Python libraries used as well as their versions, the system dependencies that need to be installed, etc.).

But storing this metadata is not enough. Deploying to production should automatically and reliably rebuild this environment on the target machine. In addition, the target machine will typically run multiple models simultaneously, and two models may have incompatible dependency versions. Finally, several models running on the same machine could compete for resources, and one misbehaving model could hurt the performance of multiple cohosted models.

Containerization technology is increasingly used to tackle these challenges. These tools bundle an application together with all of its related configuration files, libraries, and dependencies that are required for it to run across different operating environments. Unlike virtual machines (VMs), containers do not duplicate the complete operating system; multiple containers share a common operating system and are therefore far more resource efficient.

The most well-known containerization technology is the open source platform Docker. Released in 2014, it has become the de facto standard. It allows an application to be packaged, sent to a server (the Docker host), and run with all its dependencies in isolation from other applications.

Building the basis of a model-serving environment that can accommodate many models, each of which may run multiple copies, may require multiple Docker hosts. When deploying a model, the framework should solve a number of issues:

- Which Docker host(s) should receive the container?
- When a model is deployed in several copies, how can the workload be balanced?
- What happens if the model becomes unresponsive, for example, if the machine hosting it fails? How can that be detected and a container reprovisioned?
- How can a model running on multiple machines be upgraded, with assurances that old and new versions are switched on and off, and that the load balancer is updated with a correct sequence?

Kubernetes, an open source platform that has gained a lot of traction in the past few years and is becoming the standard for container orchestration, greatly simplifies these issues and many others. It provides a powerful declarative API to run applications in a group of Docker hosts, called a Kubernetes *cluster*. The word *declarative* means that rather than trying to express in code the steps to set up, monitor, upgrade, stop, and connect the container (which can be complex and error prone), users specify in a configuration file the desired state, and Kubernetes makes it happen and then maintains it.

For example, users need only specify to Kubernetes "make sure four instances of this container run at all times," and Kubernetes will allocate the hosts, start the containers, monitor them, and start a new instance if one of them fails. Finally, the major cloud providers all provide managed Kubernetes services; users do not even have to install

and maintain Kubernetes itself. If an application or a model is packaged as a Docker container, users can directly submit it, and the service will provision the required machines to run one or several instances of the container inside Kubernetes.

Docker with Kubernetes can provide a powerful infrastructure to host applications, including ML models. Leveraging these products greatly simplifies the implementation of the deployment strategies—like blue-green deployments or canary releases—although they are not aware of the nature of the deployed applications and thus can't natively manage the ML performance analysis. Another major advantage of this type of infrastructure is the ability to easily scale the model's deployment.

Scaling Deployments

As ML adoption grows, organizations face two types of growth challenges:

- The ability to use a model in production with high-scale data
- The ability to train larger and larger numbers of models

Handling more data for real-time scoring is made much easier by frameworks such as Kubernetes. Since most of the time trained models are essentially formulas, they can be replicated in the cluster in as many copies as necessary. With the auto-scaling features in Kubernetes, both provisioning new machines and load balancing are fully handled by the framework, and setting up a system with huge scaling capabilities is now relatively simple. The major difficulty can then be to process the large amount of monitoring data; Chapter 7 provides some details on this challenge.

Scalable and Elastic Systems

A computational system is said to be horizontally scalable (or just scalable) if it is possible to incrementally add more computers to expand its processing power. For example, a Kubernetes cluster can be expanded to hundreds of machines. However, if a system includes only one machine, it may be challenging to incrementally upgrade it significantly, and at some point, a migration to a bigger machine or a horizontally scalable system will be required (and may be very expensive and require interruption of service).

An elastic system allows, in addition to being scalable, easy addition and removal of resources to match the compute requirements. For example, a Kubernetes cluster in the cloud can have an auto-scaling capability that automatically adds machines when the cluster usage metrics are high and removes them when they are low. In principle, elastic systems can optimize the usage of resources; they automatically adapt to an increase in usage without the need to permanently provision resources that are rarely required.

For batch scoring, the situation can be more complex. When the volume of data becomes too large, there are essentially two types of strategies to distribute the computation:

- Using a framework that handles distributed computation natively, in particular Spark. Spark is an open source distributed computation framework. It is useful to understand that Spark and Kubernetes do not play similar roles and can be combined. Kubernetes orchestrates containers, but Kubernetes is not aware of what the containers are actually doing; as far as Kubernetes is concerned, they are just containers that run an application on one specific host. (In particular, Kubernetes has no concept of data processing, as it can be used to run any kind of application.) Spark is a computation framework that can split the data and the computation among its nodes. A modern way to use Spark is through Kubernetes. To run a Spark job, the desired number of Spark containers are started by Kubernetes; once they are started, they can communicate to complete the computation, after which the containers are destroyed and the resources are available for other applications, including other Spark jobs that may have different Spark versions or dependencies.
- Another way to distribute batch processing is to partition the data. There are many ways to achieve this, but the general idea is that scoring is typically a row-by-row operation (each row is scored one by one), and the data can be split in some way so that several machines can each read a subset of the data and score a subset of the rows.

In terms of computation, scaling the number of models is somewhat simpler. The key is to add more computing power and to make sure the monitoring infrastructure can handle the workload. But in terms of governance and processes, this is the most challenging situation.

In particular, scaling the number of models means that the CI/CD pipeline must be able to handle large numbers of deployments. As the number of models grows, the need for automation and governance grows, as human verification cannot necessarily be systematic or consistent.

In some applications, it is possible to rely on fully automated continuous deployment if the risks are well controlled by automated validation, canary releases, and automated canary analysis. There can be numerous infrastructure challenges since training, building models, validating on test data, etc., all need to be performed on clusters rather than on a single machine. Also, with a higher number of models, the CI/CD pipeline of each model can vary widely, and if nothing is done, each team will have to develop its own CI/CD pipeline for each model.

This is suboptimal from efficiency and governance perspectives. While some models may need highly specific validation pipelines, most projects can probably use a small

number of common patterns. In addition, maintenance is made much more complex as it may become impractical to implement a new systematic validation step, for example, since the pipelines would not necessarily share a common structure and would then be impossible to update safely, even programmatically. Sharing practices and standardized pipelines can help limit complexity. A dedicated tool to manage large numbers of pipelines can also be used; for example, Netflix released Spinnaker, an open source continuous deployment and infrastructure management platform.

Requirements and Challenges

When deploying a model, there are several possible scenarios:

- One model deployed on one server
- One model deployed on multiple servers
- Multiple versions of a model deployed on one server
- Multiple versions of a model deployed on multiple servers
- Multiple versions of multiple models deployed on multiple servers

An effective logging system should be able to generate centralized datasets that can be exploited by the model designer or the ML engineer, usually outside of the production environment. More specifically, it should cover all of the following situations:

- The system can access and retrieve scoring logs from multiple servers, either in a real-time scoring use case or in a batch scoring use case.
- When a model is deployed on multiple servers, the system can handle the mapping and aggregation of all information per model across servers.
- When different versions of a model are deployed, the system can handle the mapping and aggregation of all information per version of the model across servers.

In terms of challenges, for large-scale machine learning applications, the number of raw event logs generated can be an issue if there are no preprocessing steps in place to filter and aggregate data. For real-time scoring use cases, logging streaming data requires setting up a whole new set of tooling that entails a significant engineering effort to maintain. However, in both cases, because the goal of monitoring is usually to estimate aggregate metrics, saving only a subset of the predictions may be acceptable.

CHAPTER 7

Monitoring and Feedback Loop

Du Phan

When a machine learning model is deployed in production, it can start degrading in quality fast—and without warning—until it's too late (i.e., it's had a potentially negative impact on the business). That's why model monitoring is a crucial step in the ML model life cycle and a critical piece of MLOps (illustrated in Figure 7-1 as a part of the overall life cycle).

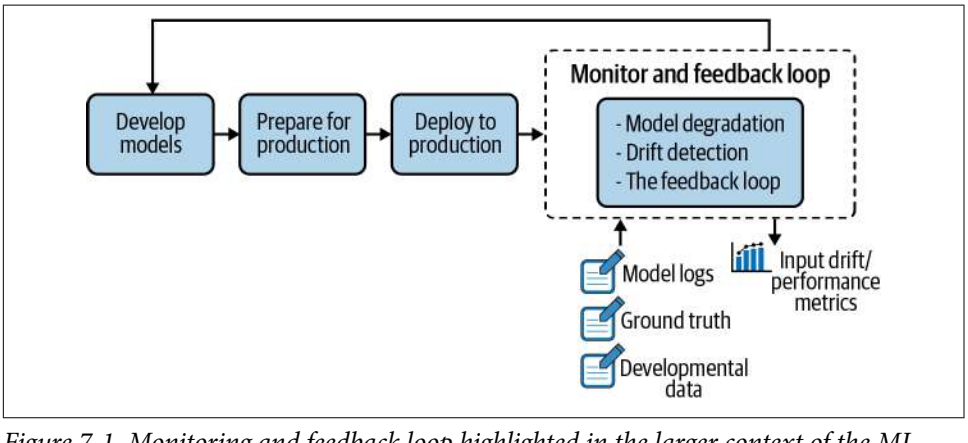

Figure 7-1. Monitoring and feedback loop highlighted in the larger context of the ML project life cycle

Machine learning models need to be monitored at two levels:

- At the resource level, including ensuring the model is running correctly in the production environment. Key questions include: Is the system alive? Are the CPU, RAM, network usage, and disk space as expected? Are requests being processed at the expected rate?

- At the performance level, meaning monitoring the pertinence of the model over time. Key questions include: Is the model still an accurate representation of the pattern of new incoming data? Is it performing as well as it did during the design phase?

The first level is a traditional DevOps topic that has been extensively addressed in the literature (and has been covered in Chapter 6). However, the latter is more complicated. Why? Because how well a model performs is a reflection of the data used to train it; in particular, how representative that training data is of the live request data. As the world is constantly changing, a static model cannot catch up with new patterns that are emerging and evolving without a constant source of new data. While it is possible to detect large deviations on single predictions (see Chapter 5), smaller but still significant deviations have to be detected statistically on datasets of scored rows, with or without ground truth.

Model performance monitoring attempts to track this degradation, and, at an appropriate time, it will also trigger the retraining of the model with more representative data. This chapter delves into detail on how data teams should handle both monitoring and subsequent retraining.

How Often Should Models Be Retrained?

One of the key questions teams have regarding monitoring and retraining is: how often should models be retrained? Unfortunately, there is no easy answer, as this question depends on many factors, including:

The domain
: Models in areas like cybersecurity or real-time trading need to be updated regularly to keep up with the constant changes inherent in these fields. Physical models, like voice recognition, are generally more stable, because the patterns don't often abruptly change. However, even more stable physical models need to adapt to change: what happens to a voice recognition model if the person has a cough and the tone of their voice changes?

The cost
: Organizations need to consider whether the cost of retraining is worth the improvement in performance. For example, if it takes one week to run the whole data pipeline and retrain the model, is it worth a 1% improvement?

The model performance
: In some situations, the model performance is restrained by the limited number of training examples, and thus the decision to retrain hinges on collecting enough new data.

Whatever the domain, the delay to obtain the ground truth is key to defining a lower bound to the retraining period. It is very risky to use a prediction model when there is a possibility that it drifts faster than the lag between prediction time and ground truth obtention time. In this scenario, the model can start giving bad results without any recourse other than to withdraw the model if the drift is too significant. What this means in practice is that it is unlikely a model with a lag of one year is retrained more than a few times a year.

For the same reason, it is unlikely that a model is trained on data collected during a period smaller than this lag. Retraining will not be performed in a shorter period, either. In other words, if the model retraining occurs way more often than the lag, there will be almost no impact of the retraining on the performance of the model.

There are also two organizational bounds to consider when it comes to retraining frequency:

An upper bound
: It is better to perform retraining once every year to ensure that the team in charge has the skills to do it (despite potential turnover—i.e., the possibility that the people retraining the model were not the ones who built it) and that the computing toolchain is still up.

A lower bound
: Take, for example, a model with near-instantaneous feedback, such as a recommendation engine where the user clicks on the product offerings within seconds after the prediction. Advanced deployment schemes will involve shadow testing or A/B testing to make sure that the model performs as anticipated. Because it is a statistical validation, it takes some time to gather the required information. This necessarily sets a lower bound to the retraining period. Even with a simple deployment, the process will probably allow for some human validation or for the possibility of manual rollback, which means it's unlikely that the retraining will occur more than once a day.

Therefore, it is very likely that retraining will be done between once a day and once a year. The simplest solution that consists of retraining the model in the same way and in the same environment it was trained in originally is acceptable. Some critical cases may require retraining in a production environment, even though the initial training was done in a design environment, but the retraining method is usually identical to the training method so that the overall complexity is limited. As always, there is an exception to this rule: online learning.

Online Learning

Sometimes, the use case requires teams to go further than the automation of the existing manual ML pipeline by using dedicated algorithms that can train themselves iteratively. (Standard algorithms, by contrast, are retrained from scratch most of the time, with the exception of deep learning algorithms.)

While conceptually attractive, these algorithms are more costly to set up. The designer has to not only test the performance of the model on a test dataset, but also qualify its behavior when data changes. (The latter is required because it's difficult to mitigate bad learning once the algorithm is deployed, and it's hard to reproduce the behavior when each training recursively relies on the previous one because one needs to replay all the steps to understand the bad behavior). In addition, these algorithms are not stateless: running them twice on the same data will not give the same result because they have learned from the first run.

There is no standard way—similar to cross-validation—to do this process, so the design costs will be higher. Online machine learning is a vivid branch of research with some mature technologies like state-space models, though they require significant skills to be used effectively. Online learning is typically appealing in streaming use cases, though mini batches may be more than enough to handle it.

In any case, some level of model retraining is definitely necessary—it's not a question of if, but of when. Deploying ML models without considering retraining would be like launching an unmanned aircraft from Paris in the exact right direction and hoping it will land safely in New York City without further control.

The good news is that if it was possible to gather enough data to train the model the first time, then most of the solutions for retraining are already available (with the possible exception of cross-trained models that are used in a different context—for example, trained with data from one country but used in another). It is therefore critical for organizations to have a clear idea of deployed models' drift and accuracy by setting up a process that allows for easy monitoring and notifications. An ideal scenario would be a pipeline that automatically triggers checks for degradation of model performance.

It's important to note that the goal of notifications is not necessarily to kick off an automated process of retraining, validation, and deployment. Model performance can change for a variety of reasons, and retraining may not always be the answer. The point is to alert the data scientist of the change; that person can then diagnose the issue and evaluate the next course of action.

It is therefore critical that as part of MLOps and the ML model life cycle, data scientists and their managers and the organization as a whole (which is ultimately the

entity that has to deal with the business consequences of degrading model performances and any subsequent changes) understand model degradation. Practically, every deployed model should come with monitoring metrics and corresponding warning thresholds to detect meaningful business performance drops as quickly as possible. The following sections focus on understanding these metrics to be able to define them for a particular model.

Understanding Model Degradation

Once a machine learning model is trained and deployed in production, there are two approaches to monitor its performance degradation: ground truth evaluation and input drift detection. Understanding the theory behind and limitations of these approaches is critical to determining the best strategy.

Ground Truth Evaluation

Ground truth retraining requires waiting for the label event. For example, in a fraud detection model, the ground truth would be whether or not a specific transaction was actually fraudulent. For a recommendation engine, it would be whether or not the customer clicked on—or ultimately bought—one of the recommended products.

With the new ground truth collected, the next step is to compute the performance of the model based on ground truth and compare it with registered metrics in the training phase. When the difference surpasses a threshold, the model can be deemed as outdated, and it should be retrained.

The metrics to be monitored can be of two varieties:

- Statistical metrics like accuracy, ROC AUC, log loss, etc. As the model designer has probably already chosen one of these metrics to pick the best model, it is a first-choice candidate for monitoring. For more complex models, where the average performance is not enough, it may be necessary to look at metrics computed by subpopulations.
- Business metrics, like cost-benefit assessment. For example, the credit scoring business has developed its own specific metrics.

The main advantage of the first kind of metric is that it is domain agnostic, so the data scientist likely feels comfortable setting thresholds. So as to have the earliest meaningful warning, it is even possible to compute p-values to assess the probability that the observed drop is not due to random fluctuations.

A Stats Primer: From Null Hypothesis to *p*-Values

The *null hypothesis* says that there is no relationship between the variables being compared; any results are due to sheer chance.

The *alternative hypothesis* says that the variables being compared are related, and the results are *significant* in supporting the theory being considered, and not due to chance.

The level of statistical significance is often expressed as a *p-value* between 0 and 1. The smaller the *p*-value, the stronger the evidence that one should reject the null hypothesis.

The drawback is that the drop may be statistically significant without having any noticeable impact. Or worse, the cost of retraining and the risk associated with a redeployment may be higher than the expected benefits. Business metrics are far more interesting because they ordinarily have a monetary value, enabling subject matter experts to better handle the cost-benefit trade-off of the retraining decision.

When available, ground truth monitoring is the best solution. However, it may be problematic. There are three main challenges:

- Ground truth is not always immediately, or even imminently, available. For some types of models, teams need to wait months (or longer) for ground truth labels to be available, which can mean significant economic loss if the model is degrading quickly. As said before, deploying a model for which the drift is faster than the lag is risky. However, by definition, drifts are not forecastable, so models with long lags need mitigation measures.
- Ground truth and prediction are decoupled. To compute the performance of the deployed model on new data, it's necessary to be able to match ground truth with the corresponding observation. In many production environments, this is a challenging task because these two pieces of information are generated and stored in different systems and at different timestamps. For low-cost or short-lived models, it might not be worth automated ground truth collection. Note that this is rather short-sighted, because sooner or later, the model will need to be retrained.
- Ground truth is only partially available. In some situations, it is extremely expensive to retrieve the ground truth for all the observations, which means choosing which samples to label and thus inadvertently introducing bias into the system.

For the last challenge, fraud detection presents a clear use case. Given that each transaction needs to be examined manually and the process takes a long time, does it make sense to establish ground truth for only suspect cases (i.e., cases where the model

gives a high probability of fraud)? At first glance, the approach seems reasonable; however, a critical mind understands that this creates a feedback loop that will amplify the flaws of the model. Fraud patterns that were never captured by the model (i.e., those that have a low fraud probability according to the model) will never be taken into account in the retraining process.

One solution to this challenge might be to randomly label, establishing a ground truth for just a subsample of transactions in addition to those that were flagged as suspicious. Another solution might be to reweight the biased sample so that its characteristics match the general population more closely. For example, if the system awarded little credit to people with low income, the model should reweight them according to their importance in the applicant, or even in the general, population.

The bottom line is that whatever the mitigation measure, the labeled sample subset must cover all possible future predictions so that the trained model makes good predictions whatever the sample; this will sometimes mean making suboptimal decisions for the sake of checking that the model continues to generalize well.

Once this problem is solved for retraining, the solution (reweighting, random sampling) can be used for monitoring. Input drift detection complements this approach, as it is needed to make sure that ground truth covering new, unexplored domains is made available to retrain the model.

Input Drift Detection

Given the challenges and limitations of ground truth retraining presented in the previous section, a more practical approach might be input drift detection. This section takes a brief but deep dive into the underlying logic behind drift and presents different scenarios that can cause models and data to drift.

Say the goal is to predict the quality of Bordeaux wines using as training data the UCI Wine Quality dataset, which contains information about red and white variants of the Portuguese wine vinho verde along with a quality score varying between 0 and 10.

The following features are provided for each wine: type, fixed acidity, volatile acidity, citric acid, residual sugar, chlorides, free sulfur dioxide, total sulfur dioxide, density, pH, sulphates, and alcohol rate.

To simplify the modeling problem, say that a good wine is one with a quality score equal to or greater than 7. The goal is thus to build a binary model that predicts this label from the wine's attributes.

To demonstrate data drift, we explicitly split the original dataset into two:

- wine_alcohol_above_11, which contains all wines with an alcohol rate of 11% and above
- wine_alcohol_below_11, which contains all wines with an alcohol rate below 11%

We split wine_alcohol_above_11 to train and score our model, and the second dataset, wine_alcohol_below_11, will be considered as new incoming data that needs to be scored once the model has been deployed.

We have artificially created a big problem: it is very unlikely that the quality of wine is independent from the alcohol level. Worse, the alcohol level is likely to be correlated differently with the other features in the two datasets. As a result, what is learned on one dataset ("if the residual sugar is low and the pH is high, then the probability that the wine is good is high") may be wrong on the other one because, for example, the residual sugar is not important anymore when the alcohol level is high.

Mathematically speaking, the samples of each dataset cannot be assumed to be drawn from the same distribution (i.e., they are not "identically distributed"). Another mathematical property is necessary to ensure that ML algorithms perform as expected: independence. This property is broken if samples are duplicated in the dataset or if it is possible to forecast the "next" sample given the previous one, for example.

Let's assume that despite the obvious problems, we train the algorithm on the first dataset and then deploy it on the second one. The resulting distribution shift is called a drift. It will be called a feature drift if the alcohol level is one of the features used by the ML model (or if the alcohol level is correlated with other features used by the model) and a concept drift if it is not.

Drift Detection in Practice

As explained previously, to be able to react in a timely manner, model behavior should be monitored solely based on the feature values of the incoming data, without waiting for the ground truth to be available.

The logic is that if the data distribution (e.g., mean, standard deviation, correlations between features) diverges between the training and testing phases[1] on one side and the development phase on the other, it is a strong signal that the model's performance won't be the same. It is not the perfect mitigation measure, as retraining on the

1 It is also advisable to assess the drift between the training and the test dataset, especially when the test dataset is posterior to the training dataset. See "Choosing Evaluation Metrics" on page 51 for details.

drifted dataset will not be an option, but it can be part of mitigation measures (e.g., reverting to a simpler model, reweighting).

Example Causes of Data Drift

There are two frequent root causes of data drift:

- Sample selection bias, where the training sample is not representative of the population. For instance, building a model to assess the effectiveness of a discount program will be biased if the best discounts are proposed for the best clients. Selection bias often stems from the data collection pipeline itself. In the wine example, the original dataset sample with alcohol levels above 11% surely does not represent the whole population of wines—this is sample selection at its best. It could have been mitigated if a few samples of wine with an alcohol level above 11% had been kept and reweighted according to the expected proportion in the population of wines to be seen by the deployed model. Note that this task is easier said than done in real life, as the problematic features are often unknown or maybe even not available.
- Non-stationary environment, where training data collected from the source population does not represent the target population. This often happens for time-dependent tasks—such as forecasting use cases—with strong seasonality effects, where learning a model over a given month won't generalize to another month. Back to the wine example: one can imagine a case where the original dataset sample only includes wines from a specific year, which might represent a particularly good (or bad) vintage. A model trained on this data may not generalize to other years.

Input Drift Detection Techniques

After understanding the possible situations that can cause different types of drift, the next logical question is: how can drift be detected? This section presents two common approaches. The choice between them depends on the expected level of interpretability.

Organizations that need proven and explainable methods should prefer univariate statistical tests. If complex drift involving several features simultaneously is expected, or if the data scientists want to reuse what they already know and assuming the organization doesn't dread the black box effect, the domain classifier approach may be a good option, too.

Univariate statistical tests

This method requires applying a statistical test on data from the source distribution and the target distribution for each feature. A warning will be raised when the results of those tests are significant.

The choice of hypothesis tests have been extensively studied in the literature, but the basic approaches rely on these two tests:

- For continuous features, the Kolmogorov-Smirnov test is a nonparametric hypothesis test that is used to check whether two samples come from the same distribution. It measures a distance between the empirical distribution functions.
- For categorical features, the Chi-squared test is a practical choice that checks whether the observed frequencies for a categorical feature in the target data match the expected frequencies seen from the source data.

The main advantage of *p*-values is that they help detect drift as quickly as possible. The main drawback is that they detect an effect, but they do not quantify the level of the effect (i.e., on large datasets, they detect very small changes, which may be completely without impact). As a result, if development datasets are very large, it is necessary to complement *p*-values with business-significant metrics. For example, on a sufficiently large dataset, the average age may have significantly drifted from a statistical perspective, but if the drift is only a few months, this is probably an insignificant value for many business use cases.

Domain classifier

In this approach, data scientists train a model that tries to discriminate between the original dataset (input features and, optionally, predicted target) and the development dataset. In other words, they stack the two datasets and train a classifier that aims at predicting the data's origin. The performance of the model (its accuracy, for example) can then be considered as a metric for the drift level.

If this model is successful in its task, and thus has a high drift score, it implies that the data used at training time and the new data can be distinguished, so it's fair to say that the new data has drifted. To gain more insights, in particular to identify the features that are responsible for the drift, one can use the feature importance of the trained model.

Interpretation of results

Both domain classifier and univariate statistical tests point to the importance of features or of the target to explain drift. Drift attributed to the target is important to identify because it often directly impacts the bottom line of the business. (Think, for example, of credit scores: if the scores are lower overall, the number of awarded loans

is likely to be lower, and therefore revenue will be lower.) Drift attributed to features is useful to mitigate the impact of drift, as it may hint at the need for:

- Reweighting according to this feature (e.g., if customers above 60 now represent 60% of users but were only 30% in the training set, then double their weight and retrain the model)
- Removing the feature and training a new model without it

In all cases, it is very unlikely that automatic actions exist if drift is detected. It could happen if it is costly to deploy retrained models: the model would be retrained on new data only if performance based on ground truth had dropped or significant drift was detected. In this peculiar case, new data is indeed available to mitigate the drift.

The Feedback Loop

All effective machine learning projects implement a form of data feedback loop; that is, information from the production environment flows back to the model prototyping environment for further improvement.

One can see in Figure 7-2 that data collected in the monitoring and feedback loop is sent to the model development phase (details about this data are covered in Chapter 6). From there, the system analyzes whether the model is working as expected. If it is, no action is required. If the model's performance is degrading, an update will be triggered, either automatically or manually by the data scientist. In practice, as seen at the beginning of this chapter, this usually means either retraining the model with new labeled data or developing a new model with additional features.

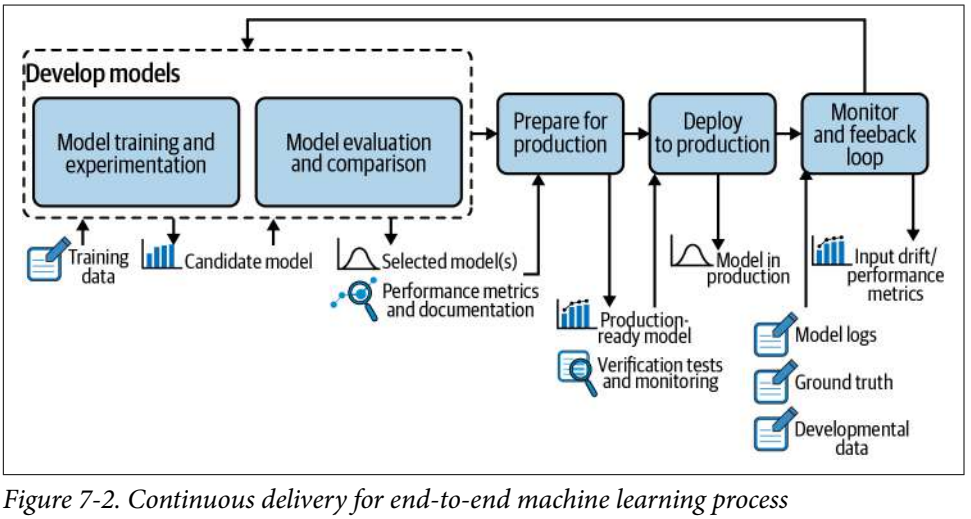

Figure 7-2. Continuous delivery for end-to-end machine learning process

In either case, the goal is to be able to capture the emerging patterns and make sure that the business is not negatively impacted. This infrastructure is comprised of three main components, which in addition to the concepts discussed in the first part of this chapter, are critical to robust MLOps capabilities:

- A logging system that collects data from several production servers
- A model evaluation store that does versioning and evaluation between different model versions
- An online system that does model comparison on production environments, either with the shadow scoring (champion/challenger) setup or with A/B testing

The following sections address each of these components individually, including their purpose, key features, and challenges.

Logging

Monitoring a live system, with or without machine learning components, means collecting and aggregating data about its states. Nowadays, as production infrastructures are getting more and more complex, with several models deployed simultaneously across several servers, an effective logging system is more important than ever.

Data from these environments needs to be centralized to be analyzed and monitored, either automatically or manually. This will enable continuous improvement of the ML system. An event log of a machine learning system is a record with a timestamp and the following information.

Model metadata
: Identification of the model and the version.

Model inputs
: Feature values of new observations, which allow for verification of whether the new incoming data is what the model was expecting and thus allowing for detection of data drift (as explained in the previous section).

Model outputs
: Predictions made by the model that, along with the ground truth collected later on, give a concrete idea about the model performance in a production environment.

System action
: It's rare that the model prediction is the end product of a machine learning application; the more common situation is that the system will take an action based on this prediction. For example, in a fraud detection use case, when the model gives high probability, the system can either block the transaction or send a warning to

the bank. This type of information is important because it affects the user reaction and thus indirectly affects the feedback data.

Model explanation
: In some highly regulated domains such as finance or healthcare, predictions must come with an explanation (i.e., which features have the most influence on the prediction). This kind of information is usually computed with techniques such as Shapley value computation and should be logged to identify potential issues with the model (e.g., bias, overfitting).

Model Evaluation

Once the logging system is in place, it periodically fetches data from the production environment for monitoring. Everything goes well until one day the data drift alert is triggered: the incoming data distribution is drifting away from the training data distribution. It's possible that the model performance is degrading.

After review, data scientists decide to improve the model by retraining it, using the techniques described earlier in this chapter. With several trained candidate models, the next step is to compare them with the deployed model. In practice, this means evaluating all the models (the candidates as well as the deployed model) on the same dataset. If one of the candidate models outperforms the deployed model, there are two ways to proceed: either update the model on the production environment or move to an online evaluation via a champion/challenger or A/B testing setup.

In a nutshell, this is the notion of model store. It is a structure that allows data scientists to:

- Compare multiple, newly trained model versions against existing deployed versions
- Compare completely new models against versions of other models on labeled data
- Track model performance over time

Formally, the model evaluation store serves as a structure that centralizes the data related to model life cycle to allow comparisons (though note that comparing models makes sense only if they address the same problem). By definition, all these comparisons are grouped under the umbrella of a logical model.

Logical model

Building a machine learning application is an iterative process, from deploying to production, monitoring performance, retrieving data, and looking for ways to improve how the system addresses the target problem. There are many ways to iterate, some of which have already been discussed in this chapter, including:

- Retraining the same model on new data
- Adding new features to the model
- Developing new algorithms

For those reasons, the machine learning model itself is not a static object; it constantly changes with time. It is therefore helpful to have a higher abstraction level to reason about machine learning applications, which is referred to as a *logical model*.

A logical model is a collection of model templates and their versions that aims to solve a business problem. A model version is obtained by training a model template on a given dataset. All versions of model templates of the same logical model can usually be evaluated on the same kinds of datasets (i.e., on datasets with the same feature definition and/or schema); however, this may not be the case if the problem did not change but the features available to solve it did. Model versions could be implemented using completely different technologies, and there could even be several implementations of the same model version (Python, SQL, Java, etc.); regardless, they are supposed to give the same prediction if given the same input.

Let's get back to the wine example introduced earlier in this chapter. Three months after deployment, there is new data about less alcoholic wine. We can retrain our model on the new data, thus obtaining a new model version using the same model template. While investigating the result, we discover new patterns are emerging. We may decide to create new features that capture this information and add it to the model, or we may decide to use another ML algorithm (like deep learning) instead of XGBoost. This would result in a new model template.

As a result, our model has two model templates and three versions:

- The first version is live in production, based on the original model template.
- The second version is based on the original template, but trained on new data.
- The third version uses the deep learning–based template with additional features and is trained on the same data as the second version.

The information about the evaluation of these versions on various datasets (both the test datasets used at training time and the development datasets that may be scored after training) is then stored in the model evaluation store.

Model evaluation store

As a reminder, model evaluation stores are structures that centralize the data related to model life cycles to allow comparisons. The two main tasks of a model evaluation store are:

- Versioning the evolution of a logical model through time. Each logged version of the logical model must come with all the essential information concerning its training phase, including:
  - The list of features used
  - The preprocessing techniques that are applied to each feature
  - The algorithm used, along with the chosen hyperparameters
  - The training dataset
  - The test dataset used to evaluate the trained model (this is necessary for the version comparison phase)
  - Evaluation metrics
- Comparing the performance between different versions of a logical model. To decide which version of a logical model to deploy, all of them (the candidates and the deployed one) must be evaluated on the same dataset.

The choice of dataset to evaluate is crucial. If there is enough new labeled data to give a reliable estimation of the model performance, this is the preferred choice because it is closest to what we are expecting to receive in the production environment. Otherwise, we can use the original test dataset of the deployed model. Assuming that the data has not drifted, this gives us a concrete idea about the performance of the candidate models compared to the original model.

After identifying the best candidate model, the job is not yet done. In practice, there is often a substantial discrepancy between the offline and online performance of the models. Therefore, it's critical to take the testing to the production environment. This online evaluation gives the most truthful feedback about the behavior of the candidate model when facing real data.

Online Evaluation

Online evaluation of models in production is critical from a business perspective, but can be challenging from a technical perspective. There two main modes of online evaluation:

- Champion/challenger (otherwise known as shadow testing), where the candidate model shadows the deployed model and scores the same live requests
- A/B testing, where the candidate model scores a portion of the live requests and the deployed model scores the others

Both cases require ground truth, so the evaluation will necessarily take longer than the lag between prediction and ground truth obtention. In addition, whenever

shadow testing is possible, it should be used over A/B testing because it is far simpler to understand and set up, and it detects differences more quickly.

Champion/Challenger

Champion/challenger involves deploying one or several additional models (the challengers) to the production environment. These models receive and score the same incoming requests as the active model (the champion). However, they do not return any response or prediction to the system: that's still the job of the old model. The predictions are simply logged for further analysis. That's why this method is also called "shadow testing" or "dark launch."

This setup allows for two things:

- Verification that the performance of the new models is better than, or at least as good as, the old model. Because the two models are scoring on the same data, there is a direct comparison of their accuracy in the production environment. Note that this could also be done offline by using the new models on the dataset made of new requests scored by the champion model.
- Measurement of how the new models handle realistic load. Because the new models can have new features, new preprocessing techniques, or even a new algorithm, the prediction time for a request won't be the same as that of the original model, and it is important to have a concrete idea of this change. Of course, this is the main advantage of doing it online.

The other advantage of this deployment scheme is that the data scientist or the ML engineer is giving visibility to other stakeholders on the future champion model: instead of being locked in the data science environment, the challenger model results are exposed to the business leaders, which decreases the perceived risk to switch to a new model.

To be able to compare the champion and the challenger models, the same information must be logged for both, including input data, output data, processing time, etc. This means updating the logging system so that it can differentiate between the two sources of data.

How long should both models be deployed before it's clear that one is better than the other? Long enough that the metric fluctuations due to randomness are dampened because enough predictions have been made. This can be assessed graphically by checking that the metric estimations are not fluctuating anymore or by doing a proper statistical test (as most metrics are averages of row-wise scores, the most usual test is a paired sample T-test) that yields the probability that the observation that one metric is higher than the other is due to these random fluctuations. The wider the metric difference, the fewer predictions necessary to ensure that the difference is significant.

Depending on the use case and the implementation of the champion/challenger system, server performance can be a concern. If two memory-intensive models are called synchronously, they can slow the system down. This will not only have a negative impact on the user experience but also corrupt the data collected about the functioning of the models.

Another concern is communication with the external system. If the two models use an external API to enrich their features, that doubles the number of requests to these services, thus doubling costs. If that API has a caching system in place, then the second request will be processed much faster than the first, which can bias the result when comparing the total prediction time of the two models. Note that the challenger may be used only for a random subset of the incoming requests, which will alleviate the load at the expense of increased time before a conclusion can be drawn.

Finally, when implementing a challenger model, it's important to ensure it doesn't have any influence on the system's actions. This implies two scenarios:

- When the challenger model encounters an unexpected issue and fails, the production environment will not experience any discontinuation or degradation in terms of response time.
- Actions taken by the system depend only on the prediction of the champion model, and they happen only once. For example, in a fraud detection use case, imagine that by mistake the challenger model is plugged directly into the system, charging each transaction twice—a catastrophic scenario.

In general, some effort needs to be spent on the logging, monitoring, and serving system to ensure the production environment functions as usual and is not impacted by any issues coming from the challenger model.

A/B testing

A/B testing (a randomized experiment testing two variants, A and B) is a widely used technique in website optimization. For ML models, it should be used only when champion/challenger is not possible. This might happen when:

- The ground truth cannot be evaluated for both models. For example, for a recommendation engine, the prediction gives a list of items on which a given customer is likely to click if they are presented. Therefore, it is impossible to know if the customer would have clicked if an item was not presented. In this case, some kind of A/B testing will have to be done, in which some customers will be shown the recommendations of model A, and some the recommendations of model B. Similarly, for a fraud detection model, because heavy work is needed to obtain the ground truth, it may not be possible to do so for the positive predictions of two models; it would increase the workload too much, because some frauds are

detected by only one model. As a result, randomly applying only the B model to a small fraction of the requests will allow the workload to remain constant.

- The objective to optimize is only indirectly related to the performance of the prediction. Imagine an ad engine based on an ML model that predicts if a user will click on the ad. Now imagine that it is evaluated on the buy rate, i.e., whether the user bought the product or service. Once again, it is not possible to record the reaction of the user for two different models, so in this case, A/B testing is the only way.

Entire books are dedicated to A/B testing, so this section presents only its main idea and a simple walkthrough. Unlike the champion/challenger framework, with A/B testing, the candidate model returns predictions for certain requests, and the original model handles the other requests. Once the test period is over, statistical tests compare the performance of the two models, and teams can make a decision based on the statistical significance of those tests.

In an MLOps context, some considerations need to be made. A walkthrough of these considerations is presented in Table 7-1.

Table 7-1. Considerations for A/B testing in MLOps

Stage	MLOps consideration
Before the A/B test	Define a clear goal: A quantitative business metric that needs to be optimized, such as click-through rate. Define a precise population: Carefully choose a segment for the test along with a splitting strategy that assures no bias between groups. (This is the so-called experimental design or randomized control trial that's been popularized by drug studies.) This may be a random split, or it may be more complex. For example, the situation might dictate that all the requests of a particular customer are handled by the same model. Define the statistical protocol: The resulting metrics are compared using statistical tests, and the null hypothesis is either rejected or retained. To make the conclusion robust, teams need to define beforehand the sample size for the desired minimum effect size, which is the minimum difference between the two models' performance metrics. Teams must also fix a test duration (or alternatively have a method to handle multiple tests). Note that with similar sample sizes, the power to detect meaningful differences will be lower than with champion/challenger because unpaired sample tests have to be used. (It is usually impossible to match each request scored with model B to a request scored with model A, whereas with champion/challenger, this is trivial.)
During the A/B test	It is important not to stop the experiment before the test duration is over, even if the statistical test starts to return a significant metric difference. This practice (also called p-hacking) produces unreliable and biased results due to cherry-picking the desired outcome.
After the A/B test	Once the test duration is over, check the collected data to make sure the quality is good. From there, run the statistical tests; if the metric difference is statistically significant in favor of the candidate model, the original model can be replaced with the new version.