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

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).

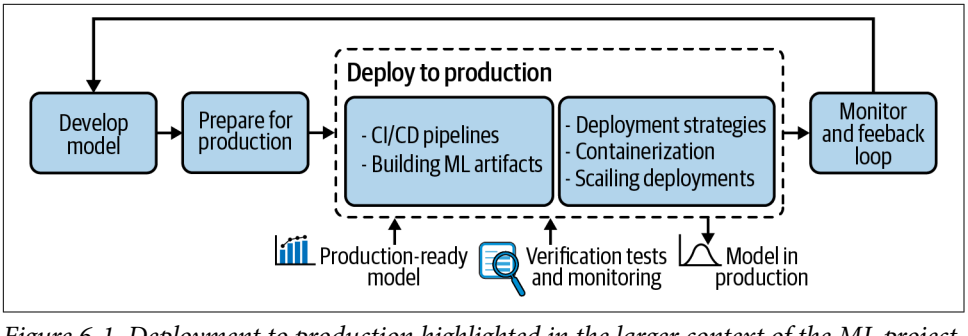

Figure 6-1. Deployment to production highlighted in the larger context of the ML project life cycle

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.