# 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).

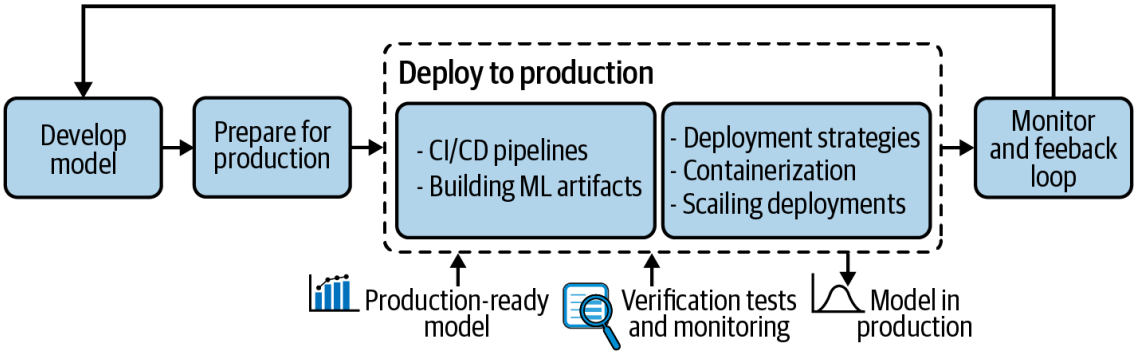


Figure 6-1. Deployment to production highlighted in the larger context of the ML project life cycle

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

plifies 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.

# Closing Thoughts

Deploying to production is a key component of MLOps, and as dissected in this chapter, having the right processes and tools in place can ensure that it happens quickly. The good news is that many of the elements of success, particularly CI/CD best practices, are not new. Once teams understand how they can be applied to machine learning models, the organization will have a good foundation on which to expand as MLOps scales with the business.