CHAPTER 7

Deploying Stochastic Systems

If you've made it this far, you already have the skills to build a complete end to end data science system. Data science of course is more than machine learning and code which are really only tools, and to build end to end systems, we need to understand people, processes, and technology, so this chapter will take a step back and give you a bird's-eye view of the entire MLOps lifecycle, tying in what we've learned in previous chapters to formally define each stage. Once we have the lifecycle defined, we'll be able to analyze it to understand how we can reduce technical debt by considering the interactions between the various stages from data collection and data engineering through to model development and deployment. We'll cover some philosophical debates between model-centric vs. data-centric approaches to MLOps and look at how we can move toward continuous delivery, the ultimate litmus test for how much value your models are creating in production. We will also discuss how the rise of generative AI may impact data science development in general, build a CI/CD pipeline for our toolkit, and talk about how we can use pre-build cloud services to deploy your models. Without further ado, let's explore the stages of the ML lifecycle again and introduce the spiral ML lifecycle formally.

D. Sorvisto, *MLOps Lifecycle Toolkit*, https://doi.org/10.1007/978-1-4842-9642-4_7

Introducing the Spiral MLOps Lifecycle

Although we hinted at the ML lifecycle throughout this book and even talked about the “spiral” MLOps lifecycle in Chapter 1 (shown in Figure 7-1), we lacked the context to really define the lifecycle completely and to understand the big picture from a holistic point of view. Although you might see the machine learning lifecycle or MLOps lifecycle (to me the difference between the two is that MLOps takes into account the business and IT environment the models live in), the reality is a lot messier. It’s always been a pet peeve of mine that there’s infographics used in data science that summarize complex ideas very concisely but don’t map very well to real-world problems. Essentially these infographics are communication tools but not structures that can be mathematically defined or reasoned upon without a lot of imagination. Therefore, it’s my goal to take the MLOps lifecycle infographic we saw in Chapter 1 to the level where you can actually recognize it in a real project or even adapt it to your own custom project since not all data science problems are the same across industries (maybe this is a kind of meta transfer learning problem in itself).

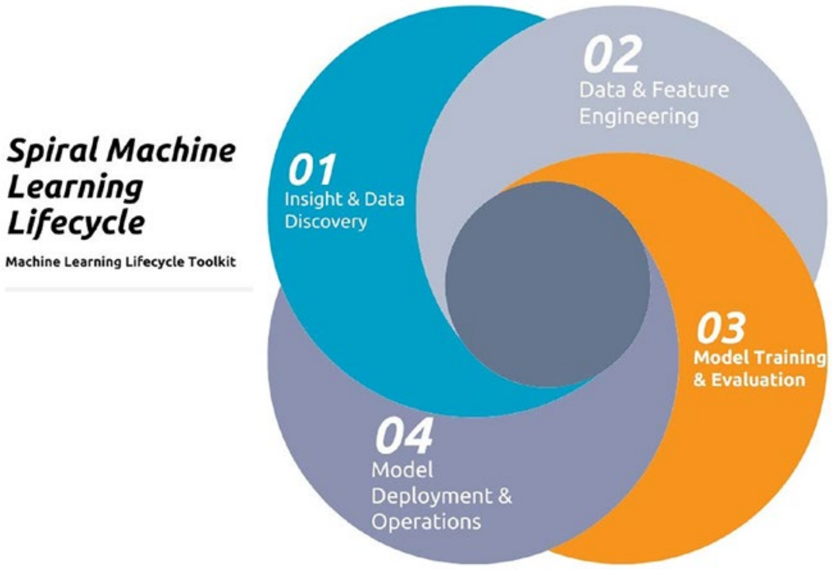
Spiral Machine Learning Lifecycle
Machine Learning Lifecycle Toolkit
01
Insight & Data Discovery
02
Data & Feature Engineering
03
Model Training & Evaluation
04
Model Deployment & Operations

Figure 7-1. *The spiral MLOps lifecycle*

So what is a lifecycle? In the context of biology, a lifecycle is a series of changes in the life of an organism. It's in itself a model for understanding change and a way of identifying the phases that come to define the organism over time.

Although MLOps is not a living organism, your IT environment is in many ways like a living, breathing organism. When we throw models and code into the mix, we legitimately have a kind of chaotic system, that although may not be technically living changes over time and is best understood by breaking it down into distinct phases.

The phases which include developing, deploying, and maintaining models can further be segmented into more granular stages that go into creating a useful machine learning model that solves a business problem.

Problem Definition

We want to start with the problem definition and requirements for the problem. This is the initial step of the lifecycle, where we define the initial conditions and involve gathering of information in the form of requirements. This is vitally important because if you can't define the problem or the requirements (a step often skipped in data science projects and justified in hand-wavy ways), then this ambiguity can trickle down into the user stories and individual developer tasks, creating chaos.

Problem Validation

The next phase of the lifecycle is validating the model. Some people confuse this with exploratory data analysis which is sometimes used as justification for finding a problem, but the goal should be to understand the problem better, what we're looking at, and the types of data sources available and validate whether or not we can solve the problem in the first phase. Problem validation is different than exploratory analysis though. Although this phase may be tedious, it saves a lot of time because it's relatively cheap to validate a problem but costly to implement a full solution that ends up missing the mark in the end.

Data Collection or Data Discovery

Once we have validated a problem, we can collect data. Collecting data is expensive. Even if you don't collect the data, there is still a lot of data discovery that has to take place. You may leverage metadata if you have a catalog built already, but if not, you may have to build the metadata catalog yourself and discover the name, variable names, data types, and statistical properties of the data.

Data Validation

At this stage, a decision needs to be made whether there is enough value in the discovered data sources or if you need to go back to the first stage to refine your problem. This is another example of a feedback loop or what we're calling a "spiral" since the process may be continuous and hopefully converge to a performance model at the end of the process.

Data Engineering

After data collection and data validation, the next step is data engineering. MLOps requires a solid data engineering foundation to support modeling activities, and this is not trivial. If you're the data engineering and MLOps practitioner on the team, then you might struggle to build this foundation before you're able to make use of MLOps.

You will set up feature stores, build feature pipelines, decide on a schedule for your pipeline (refresh rate), and ensure you're using production data sources. You have to decide which data sources are most valuable to operationalize. If the data sources are normalized (3NF or 2NF), you may have to join them together into a centralized repository.

At this stage, you may have a data architecture in mind such as a Data Warehouse or a Data Lake or Data Vault and build robust ELT pipelines. The goal of this phase is to have feature stores that are accessible, secure, and centralized and to ensure that there's enough data to support model training.

You may start feature engineering, building a library of features for model training, to support a variety of problem types.

If you are truly dealing with a prediction problem, you may only keep features with predictive value, but you may also have to deal with issues of multicollinearity and interpretability.

Model Training

Model development starts. You may start with a model baseline like simple linear regression or logistic regression; the simpler the model, the better the baseline. You might gradually increase the variance of the model. The boundary between the previous phase, data engineering, and model training will be blurred as you refine your model and require new features be added to the feature store and deal with schema drift and the curse of dimensionality. Eventually you'll have to create a way to reduce the number of features and may struggle to keep track of the entire library of features as business demand grows.

Since the lifecycle is a continuous process, after several iterations, the architecture of the model itself may change, and if enough data is available, you might consider deep learning at some point. This stage may start simple but also grow horizontally in terms of the number of models you need to support and the number of problem types and performance metrics that need to be tracked. As your MLOps process gets more advanced, the model training phase will eventually require MLFlow or similar experiment tracking software, hyper-parameter tuning frameworks like HyperOpt, hardware accelerated or distributed training, and eventually full training automation, registering your models in a model registry and having some kind of versioning system.

Diagnostic Plots and Model Retraining

Depending on the problem type, there are specific visual tools for evaluating models called diagnostic plots. For example, if you have a classification problem, you might consider plotting a decision surface for your model to evaluate its strengths and weaknesses. For a linear regression problem, you may be interested in plotting residuals vs. fitted values or some other variation to decide if it's a good or bad model.

Some of these plots may be used as diagnostic tools but not traditional monitoring tools which may not be able to accommodate images, so you could, for example, have a Jupyter notebook that's source controlled a part of the project and can generate these images on a schedule, for example, once a week, or another option is to build a separate monitoring dashboard using tools like Dash or PowerBI; the choice of reporting software really depends on your project and how you're comfortable creating the visuals, but it probably needs to support Python and libraries used like Pandas.

For model retraining, you can have more complex triggers such as if the distribution of features changes or if model performance over time is trending downward, but a simple solution to start is to retrain the model monthly. Note that these are two different types of triggers and both are needed to determine when to retrain the model since model performance can degrade over time but also the distribution of features themselves can change.

You should also consider how you write features to a table. For example, you might want to add a timestamp column and append features to a table so you have a complete history available for model retraining. These types of technical decisions around how frequently data needs to be updated, whether historical data needs to be maintained for model retraining, and how to operationalize diagnostic plots and other visuals are complex decisions that may require several team discussions.

Model Inference

In this phase, you'll select the best model, pulling the model from the model registry for use in an inference pipeline. You may decide to go through another round of cross-validation to evaluate the best model. You will need to have an inference pipeline that compiles your features and makes them available at prediction time. The runtime, model, and features will all need to be available at the same time for the model to make a prediction. Your inference pipeline may be as complicated as a full application or microservice or as simple as an API endpoint or batch inference pipeline depending on the requirements gathered during requirements gathering.

One commonality between model training and model inference is schema drift. Schema drift is also a factor in choosing a data architecture that can adapt to the demands of data science workloads since features that are used in both training and inference can change. The implication is that we either need to have complex code flows, updates, and frequent release cycles to accommodate changes in feature definition, or we need to create our tables dynamically using metadata. Since data types of each feature determine how the data is physically stored, changes in data type can impact our ability to store historical data required for model training. In the next section, I will talk about the various levels of schema drift.

The Various Levels of Schema Drift in Data Science

Schema drift is a different issue than data drift and is common to virtually all data science projects of sufficient complexity since features fed into the model may change. We have talked about schema drift before but post-deployment features can still change. It's interesting to note that there is no one size fits all solution to the problem of schema drift and actually there are various levels of schema drift. For example, you may have additive changes where you are only adding features and you may be able to simply set an option to allow the dataframe's schema to merge with the target table schema. You can implement this solution manually as well with a DDL SQL command like ALTER TABLE to avoid loss of historical data that may be required for model training.

However, what do you do when the order of columns changes, the data types are incompatible, for example, string to a float (casting a string to a float would lead to a loss of information), or there are other destructive operations on the table schema that are fundamentally not compatible with the target table? In this case, you may have to drop the table entirely. Many databases have a CREATE OR REPLACE TABLE statement, but you can implement this yourself by checking if the table exists and if it does dropping it and then recreating it. You should be careful to use atomic operations though if you're deploying this code to a distributed environment since race conditions and a source of strange errors in production are possible.

Traditional ways of handling schema drift like slowly changing dimensions don't work well for data science since features can change rapidly and the entire table structure may change this actually has consequences for model training since you could risk wiping out historical data required at a future point in time for training the model so running an ALTER TABLE statement on specific columns may be the safest bet along with regular backups of the data if possible. The schema itself, the metadata that describes the data types and structure of the table, needs to be stored as well with each version since of course this will change as well.

The Need for a More Flexible Table in Data Science

We talked about schema drift, and if you have actually worked as a data scientist, you might have encountered the problem of features being added or subtracted, names changing, data types changing regularly, and having to constantly update your tables. Traditional wisdom in database management assumes that the table structure is fixed which doesn't work well for data science.

While No-SQL databases and columnar storage address the problems of having a more flexible API and how to store data for analytical queries, you still need to handle schema drift. It's interesting to note that traditional SQL is based on relational algebra, and the equivalence with relational calculus under certain conditions such as domain independence is known as Codd's theorem.

While relational algebra and relational calculus are equivalent, relational calculus focuses on what to retrieve rather than how to retrieve it and so is more flexible. In relational calculus, there is no description of how to evaluate a query but instead a description (very similar to a prompt) of what information needs to be retrieved.

Whether a new kind of database is needed for data science that can better handle schema drift on a foundational level while maintaining performance for analytical queries is an open question that remains to be answered, but perhaps this technology could come from a fusion of large language models and the decades of wisdom built into relational database engineers ("optimizers"). In the next section, we will take all of the pieces of the lifecycle we have learned so far and discuss model deployment in general and ways to integrate all of the pieces of the puzzle into an existing business ecosystem.

Model Deployment

The model needs to be integrated into an existing business process. This seems like a technical problem but largely depends on your organization and industry. In the next section, we will look at how you can integrate your model into your business process as part of a larger system involving people, processes, and technology.

Deploying Model as Public or Private API

In the previous chapter, we talked about inference pipelines and microservices but for simple use cases where you only want to deploy a model, so it can be consumed as a private or public API endpoint, and there are many cloud services for doing this type of task; these types of cloud offerings are often called model as a service or inference as a service.

Hugging Face, for example, provides inference endpoints to easily deploy transformers, diffusers, or custom models to dedicated fully managed infrastructure in the cloud. This offering is a platform as a service where Hugging Face handles the security, load balancing, and other low level details, and you can choose your cloud provider and region if you have data compliance requirements. You can also choose public or private endpoints (intra-region secured AWS or Azure PrivateLink to VPC) that are not accessible over the Internet.

Integrating Your Model into a Business System

For stakeholders, one strategy for hedging against the vicissitudes of business is improving operational efficiency, reducing costs, and identifying opportunities to improve decision-making processes through models and innovate on insights found through data science. However, integrating a model into an established business system is a challenging problem and one that might be glossed over by data scientists and other technical leaders. The challenge is magnified when the machine learning system requires input from multiple departments and teams within those departments that have conflicting goals.

One way you can start to bring a model into an established business environment is by thinking incrementally and identifying opportunities where machine learning could bring the most value. Start with the pain points; are there repetitive tasks that could be automated? Are there tasks that nobody wants to do and may be a quick win? A good example would be cleaning data. Nobody enjoys data cleaning, but it's a business necessity and if that process is still done in spreadsheets, it may be a good candidate for automation.

Once you've identified a task that could be automated, you can investigate the data sources and look for ways the process could be improved. At this stage, it's critical to create a strategy and secure stakeholder buy-in for the first few phases of your project. Once you've proven you can bring value, adding multiple data sources, testing algorithms, training models, adding monitoring, and alerting can naturally add value and provide a segway into the next phases of the project.

Developing a Deployment Strategy

There are several established frameworks for data science and data mining that you might want to consider when building a business strategy and executing against that strategy. Although this book is meant to cover the technical aspects of the MLOps lifecycle, model deployment involves people and processes, and having a set of tools for execution can serve as a kind of checklist and mitigate risk of forgetting a step. Here are a couple frameworks you might incorporate into your own model deployment strategy.

CRISP-DM: The CRISP-DM (Cross-Industry Standard Process for Data Mining) is a standard framework originally developed for data mining but applies equally well to machine learning and data science. One of its advantages is it applies across multiple industries (we will look at specific case studies in a later chapter but it's worthwhile to have a framework that applies to multiple industries in mind). It has six phases which broadly correspond to phases in the MLOps lifecycle including model deployment:

1. Business understanding
2. Data understanding

3. Data preparation
4. Modeling
5. Evaluation
6. Deployment

Each of these phases consists of tasks, and phases follow sequentially with arrows between data preparation and modeling (unlike the spiral we talked about earlier), but this gives a very structured approach to data science and you can use it as a deployment checklist to make sure you're not missing steps. For nontechnical stakeholders, this may be a good way to communicate the various phases in a linear way. In the next section, we will look at ways in which these frameworks can be used to reduce technical debt.

Reducing Technical Debt in your Lifecycle

Technical debt can appear in many forms and can come about in different ways from working too fast to using suboptimal algorithms to forgetting how code works and making changes without updating old code and documentation. At the model deployment phase, it's critical to have standards in place to reduce technical debt across all stages of the lifecycle. Here are some deployment checklists you can use to ensure you're paying down technical debt in a timely manner:

1. Implement quality checks and linters before deployment. Friction between teams can often be reduced by doing something as simple as installing a code linter to ensure code is formatted in a standard way, eliminating arguments over what style is the best (since most data scientists have their own style). This can be done, for instance, on the main branch of the shared repository your team uses.

2. Hold regular code reviews and designate someone in charge of merging PRs (or you could implement this in a round-robin style).
3. Periodically reassess model performance post-deployment, and keep up to date with alerts and errors that are generated once the model goes to production.
4. Automate testing and monitoring as much as possible.

By following some of the preceding strategies, you can minimize technical debt post-deployment. At this point, you might be asking, I've already deployed my model, I've set up monitoring and automated testing, is it all hands-off from here? The answer is unfortunately, no. Data changes, environments change, and this does not stop after you deploy your model. Remember, the lifecycle is a continuous process. In the following section, we will look at what this process and how you can apply Agile principles in data science to make the process more efficient for you and your team. One way that you can reduce technical debt is with generative AI. In the next section, we will briefly look at how you can use generative AI to reduce technical debt by automating code reviews.

Generative AI for Code Reviews and Development

Generative AI leverages large language models which use reinforcement learning and the quadratic complexity of the transformer architecture at scale to billions of parameters. It can automate common tasks in coding and provide feedback through prompt-based development. With the rise of tools like AutoGPT, even prompt engineering is slowly being replaced.

Will this be a good thing for data scientists? I think so, as it can automate the boring stuff. Even for software developers, if you're a creative builder, you will be able to be more productive.

One way generative AI could improve the data scientist development cycle is through automated code reviews, getting feedback on their code before it is deployed. Some other ways but this list is by no means exhaustive.

- Automated code reviews
- Optimizing code (focus on accuracy)
- Translating between SQL and Python or other languages (removes translation bottlenecks)
- Generating tests for test-driven development

However, focus needs to be on validation. Output of generative models cannot be trusted, and data scientists will play a vital role in ensuring the validity, accuracy, and quality of model output when generative AI is mis-used. We should also be mindful of the cost per token and the license requirements before using this in your data science development cycle. We'll talk a lot more about these ethical issues in the next chapter, but generative AI has potential to reduce technical debt and free up time for doing data science.

Adapting Agile for Data Scientists

You may have heard of Agile before especially if you have worked on software projects in the last 20 years. Agile is a project management methodology, and although some developers have a love-hate relationship with Agile, some principles can be adapted to data science, and others fail miserably.

One principle of Agile that can be adapted is the principle of regular communication. Data science and software development share a common thread in that it's really all about communication. There's even a term for this called Conway's law which states

Any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure

What does this mean? It means, if you fail to communicate effectively with other data scientists, the structure of your project will reflect this chaos and the system you end up creating will be a mess. Setting up a side channel for communication, for example, via Microsoft Teams, can help to dissipate this risk.

How about another conundrum often faced in real-world projects: conflicting requirements. In data science, stakeholders can be incredibly demanding, and requirements you gathered in the early phase of the lifecycle will change and you will face conflicts. We can borrow from another principle of Agile development to help in this scenario, namely, prioritizing tasks.

Another important principle of both agile and other methodologies like Twelve-Factor App is the notion of clean code that can be run in separate build, run and deploy stages and that can be easily adapted. While clean code and code management is an important tenet of Agile, in data science, it's all about the data. By emphasizing proper data management, we can ensure our models are accurate. For example, developing and prioritizing a process to improve the consistency of our labeled data set could have a massive impact on the accuracy of our models.

You may want to consider applying test-driven development to data science especially in early-stage development. While testing data heavy workflows is not easy, choosing a test framework, for example, Pytest or Hypothesis, and developing data fixtures (preferably ones that use realistic data from a database) can ensure models and code are performing as you expect even after you've deployed your model. These tests from the early development stage can easily be added to a CI/CD pipeline as well and become part of the model deployment process.

One area where Agile fails miserably in data science is regular sprints. Since the lifecycle has feedback into previous phases, for example, retraining models due to data drift may require reengineering some of the features or collecting more data. How do you anticipate these changes and fit them into a regular sprint? This is a difficult question as urgent and important tasks can get added to the board mid sprint and cause havoc on data science teams. Understanding the difference between model-centric and data-centric workflows may help to align teams and reduce some of the pain points in trying to pigeonhole data science into regular sprints.

Model-Centric vs. Data-Centric Workflows

When we talk about model deployment, there are two main approaches we could take to the overall process: a model-centric approach and a data-centric approach. What do I mean by model-centric and data-centric?

In order to illustrate this somewhat philosophical concept, let's suppose you are working on an NLP problem. You're trying to classify unstructured data collected from a free form response field on a survey into categories that can help the support team quickly prioritize issues. For example, the text "I have a problem with my Internet connection" may be classified as "connectivity issue". In fact you've done an exploratory analysis of the data, and you know 90% of the training data falls into the buckets of "connectivity issue," "hardware issue," and "authentication issue," corresponding to labels in your training set. However, there's a lot of training data, several gigabytes, and there's some ambiguity. You also had to label a lot of the data by hand, and you're not sure if it's completely consistent, and 10% of the data may be classified into new buckets.

Your model accuracy is only 70%, and you decide to improve this accuracy by changing the type of model and its architecture. Eventually you decide to try transfer learning, fine-tuning the last layers of a large language model on your specific data set, and this improves the accuracy to 85%. Since you primarily focused on the model and how you could improve the model, you've taken a model-centric approach to this problem, but is there another way?

In fact you could have taken a data-centric approach to improving model accuracy by focusing more on the data. You could have improved the consistency of the labeled data or developed a process to label the 10% of the data that was unlabeled. This would have been a data-centric approach.

In reality, you could mix the two and use semi-supervised learning, developing a model or rule to label the remaining 10% of data and then working to increase the consistency of the data, taking both a model- and data-centric approach to improving model accuracy.

So which approach is better? There is some debate, and both approaches can have their advantages and disadvantages, but when the problem is well-understood, optimizing the model makes sense. When the problem is less well-understood, there's ambiguity or complexity in the data or we're working with a very large amount of data, and then a data-centric approach may be the best option since the focus would be on capturing the variability and complexity of the data to move performance metrics in the right direction. Like most things in engineering, there are guiding principles and rules of thumb but with no clear-cut answer that applies universally in all cases. Regardless of the approach you take for your specific problem, you will want to automate the process of deployment as much as possible, and in the next section, we'll borrow from a DevOps concept of continuous delivery and continuous deployment and see how it applies to stochastic systems.

Continuous Delivery for Stochastic Systems

In the previous three chapters, we discussed several types of pipelines. We talked about ETL and ELT pipelines for refreshing our feature stores, we talked about training pipelines and actually built an end to end training pipeline, and we talked about inference pipelines that automated the model prediction. How about model deployment? Is there a pipeline we can create for this process? The answer is yes and it even has a name; these types of pipelines are called CI/CD pipelines.

CI/CD (continuous integration and continuous deployment) are a type of pipeline with several steps to guarantee that each time there's a code, model, or data change, that change gets tested and deployed to the right environment.

You may have several types of environments including development, testing, staging, and production consisting of databases, configuration, code, and data that need to be deployed to these environments. The CI/CD pipeline will consist of several steps:

1. *Version control:* The pipeline can be "triggered" whenever there's a commit to the main branch. This could, for example, be a pull request after a code review and cause the pipeline to start.

2. *Automated tests:* After starting, several tests will be run. As a data scientist, you can define what tests get run; for example, you may want to check if your features have the statistical properties you expect before deploying. These tests can include security tests, data quality and code quality checks, as well as formatting like linking.

3. *Build step:* After the tests have passed, the next step of the pipeline will take the code, data, and models and package them up into an environment and runtime. This may be a docker container, for example, which can be deployed.

4. *Deployment:* Once the changes are packaged and containerized, the container is deployed to the target environment. This environment could be production, releasing your changes to a live environment with end users (do you see why we need tests first?)

Introducing to Kubeflow for Data Scientists

Kubeflow is an open source tool for data scientists that makes it relatively straightforward for both data scientists and MLOps engineers to build, deploy, and manage workflows at scale. Kubeflow provides several features for deploying models that are particularly useful for data scientists like Jupyter notebook server (similar to the one we build ourselves) for managing and deploying models and code.

Kubeflow is designed to work on top of Kubernetes, so it may be overkill for your project. In the lab, you'll be able to optionally remove the Kubeflow step of the CI/CD pipeline if you only want to say host your project on GitHub or push your code and models to a docker container and host it on a docker registry. However, knowledge of Kubeflow is worthwhile for data scientists because you may encounter it in the wild, and knowing that a tool exists that abstracts away some of the details required to work with Kubernetes is enough to get you started on the right path.

For data scientists, you can use Kubeflow in several ways to do machine learning at scale.

1. Kubeflow provides a Jupyter notebook server for developing and test models. In combination with MLFlow, this can be a powerful tool for setting up experiments and hyper-parameter tuning

2. *Scaling workflows:* Data scientists can leverage Kubeflow to do model training at scale. Kubeflow can be used to provision resources like clusters required for distributed training on GPUs or CPUs and takes care of scheduling, orchestration, and managing cluster resources.

3. *Model deployment and serving:* Data scientist can use Kubeflow to deploy models to production and serve production models to end users by deploying them as Kubernetes services (remember, this is for a full-blown application or inference API). You can manage or fine-tune the Kubernetes deployment as well add load balancers and other services so you can scale up or scale down to match demand.

I've also said this several times before; but, for some projects using Kubernetes, it is not necessary, and you may choose a batch oriented workflow for the deployment step in which case you only need to build a batch inference pipeline and use your model to make predictions in batch. This is a completely valid way to deploy models. In the lab, we'll look at creating a CI/CD pipeline that you can modify to match your particular deployment needs.

Lab: Deploying Your Data Science Project

This is the final hands-on lab of the book, and you're going to build your own CI/CD pipeline. The goal of this lab is to have a CI/CD pipeline that is part of the toolkit and that you can modify to deploy your own projects to the cloud by adding steps as necessary. We'll be using GitHub actions in this lab. You can follow along with the following steps to understand how the pipeline is constructed or look at the finished CI/CD pipeline located in the .github folder of the final MLOps toolkit included with this chapter.

Before you proceed with the lab, you should know that YAML is another data format for configuration files consisting of key value pairs that can be arranged in a hierarchy. It's a human-readable format (actually it's a superset of JSON) and is a widely used standard for defining infrastructure as code, CI/CD pipelines, and a range of other configurations used in MLOps.

1. Create a new GitHub repository for your data science project (if you need help with this, refer to the lab from Chapter 3 on setting up source control).

2. Create a .github/workflows folder in the project root. In our case, this folder already exists.

3. Create a new YAML file in the .github/workflows folder and name it, for example, cicd_model_deployment.yml.

4. Edit the YAML file as needed for your specific data science project. For example, you may need to update the name of the Docker image and the name of the container registry or remove the step to deploy to Kubernetes if you are not using Kubernetes with your project.

5. Commit the changes and push these changes to the repository created in the first step.

6. Add two secrets to the repository settings: REGISTRY_USERNAME and REGISTRY_PASSWORD. These secrets should be kept confidential and correspond to the username and password for the container registry (e.g., Docker Hub or Azure Container Registry) that you are using.

7. Try to push changes to the main branch of the repository; the pipeline will automatically be triggered.

This CI/CD pipeline performs the following steps:

- Checkout the code from the repository.
- Set up a Python environment with the specified version of Python.
- Install pipenv and the project dependencies.
- Convert notebooks to Python scripts.
- Run Pytest to test the code.
- Build and push a Docker image with the latest changes.
- Deploy the Docker image to Kubeflow using kfctl.

You can modify this lab to fit your needs; you now have a full CI/CD pipeline with automated tests and a way to deploy your models to Kubeflow whenever a change is pushed to main. Remember, you should go through a PR process to ensure code quality before pushing to main. We've also included all of the notebooks from previous labs in the toolkit as a complete package.

Open Source vs. Closed Source in Data Science

Machine learning software can be open and closed, and if you've been following industry trends, there is a battle between the two philosophies as companies seek to establish a data moat; the open source community continues to develop open source versions of tools, models, and software packages.

Somewhere between the two are composed of both open and closed components (maybe we could refer to this as "clopen" software). This is further complicated by models that significantly transform the input like generative AI. When deploying models that use open source components you have to make a technical decision whether to open source or closed source your software at the end of the day and which components you choose and accompanying licensing impacts this decision. This adds even more complexity to the problem of MLOps placing a premium on MLOps practitioners to make ethical decisions when it comes to the decision systems they are deploying, regardless of the underlying technology behind the models. In the next chapter, we'll look at some of these ethical decisions and how they impact the MLOps role.

Monolithic vs. Distributed Architectures

Architecture is about trade-offs, and although we've covered many rules of thumb in this book like SOLID principles, distributed architectures for more event driven and real-time workstreams vs. batch oriented architectures that tend to be more monolithic, there is no one perfect architecture for each project, and you need to understand the trade-offs and the type of performance, security, data, and process requirements to decide what is the best architecture. Once you are committed to an architecture or platform, it can be difficult to change though, so you should do this ground work up front and commit to one type of architecture and platform.

Choosing a Deployment Model

In data science, there are several types of deployment models that can be used, and in some cases, you need to support multi-model deployment. Choosing a deployment model that best suits your needs is key to a smooth transition from development to deployment.

On-premises deployment: In this deployment mode, you utilize your own servers or IT environment (physical hardware, e.g., your own GPU enabled server running Jupyter labs). Although this gives you maximum control over the hardware, you are responsible for patching, updates, any upgrades, and regular maintenance as well as the inbound and outbound network connectivity and security.

Public cloud deployment: This may be a cloud service provider like Azure, for instance, with your own resource groups and cloud services such as Databricks. Cloud deployment may also include public services like releasing packages to PyPi or hosting packages on web servers or even GitHub.

Mobile deployment: Creating machine learning for smartphones or mobile devices is becoming more popular lately. Since these devices have limited memory compared to servers, you need to choose between hosting your models in the cloud and connecting to them from the device or reducing the size of the model. There is ongoing research to reduce the size of large language models and other models, for example, quantization[1] (representing the model weights as fewer bits) and knowledge distillation ("distilling a model[2]") to achieve a smaller size.

[1] Kohonen, T. (1998). Learning Vector Quantization. In *Springer series in information sciences* (pp. 245–261). Springer Nature. https://doi.org/10.1007/978-3-642-56927-2_6

[2] Yuan, L., Tay, F. E. H., Li, G., Wang, T., & Feng, J. (2020). Revisiting Knowledge Distillation via Label Smoothing Regularization. https://doi.org/10.1109/cvpr42600.2020.00396

Post-deployment

The post-deployment, although not technically a phase since it's an ongoing continuous process so it is not formerly part of the lifecycle, refers to the stage where the trained model is deployed to production and being used. Some of the considerations during this phase are communicating with stakeholders, soliciting user feedback, regular maintenance, and monitoring (e.g., of an API you rely on is deprecated and it must be updated, or if you find a CVE or common vulnerability exposure that impacts a PyPi package you're using, you need to patch it).

Beyond security and stakeholder feedback, collecting user feedback and monitoring how the users are interacting with your model can be invaluable for future projects and be used to train the model and augment existing data sources. Post-deployment monitoring ensures that all of the models deployed to production continue to provide business value and are used in an ethical way.

Deploying More General Stochastic Systems

Can we use the principles in this book to deploy more general stochastic systems such as Bayesian machine learning models? The answer is yes, but we should discuss some of the caveats.

If you use a library like PyMC3 (we used this in the second chapter to create a Bayesian logistic regression model), you can still save your model, but you should choose a custom serialization framework to match the model architecture, for example, ONNX, an open standard for neural network architectures, but others include HDF5 and Python's pickle (e.g., this works well with Bayesian models from PyMC3).

You may also need to consider the types of performance metrics you want to track, for example, Bayesian information criterion for feature selection or Bayesian credibility interval along with a prediction.

The other problem is you'll need to carefully consider sampling methods you use and may have to have hardware to ensure you have sufficient entropy for random sampling. Some of the algorithms may not scale well to large data sets or be intractable, so you may have a need to use Monte Carlo methods as opposed to a "big data" solution that may be a necessary approach for some algorithms.

There may be other stochastic algorithms that you may encounter that need to productionize. For example, reinforcement learning could be applied as part of a training pipeline to do hyper-parameter search or in specific use cases in healthcare, finance, and energy to simulate physical systems and make recommendations, dynamic planning, and natural language processing.

If you use a reinforcement learning algorithm like Q-learning, you will have to think about how to represent your environment and agents and how to update a Q-table and choose a framework that can handle interacting with the environment between learning steps, so you may choose a framework like Ray RLlib framework that offers support for highly distributed workflows.

Understanding the problem type may help you to identify the frameworks available since you should not reinvent the wheel (e.g., reinforcement learning frameworks, deep learning frameworks, frameworks for Bayesian inference). Other times, you may be able to achieve similar results with another approach where a library of framework exists (e.g., many problems can be reframed to use a different methodology, like how you can solve the multiarmed bandit problem using reinforcement learning or Bayesian sampling, and this is a kind of equifinality prosperity of many stochastic systems).

Still, you may one day encounter a bespoke stochastic algorithm that has never been used before in the wild, where no Python wrapper exists, and in that case, you would have to build your own from scratch. In this scenario, you would require knowledge of a low level language like C++, compilers, hardware, distributed systems, and APIs like MPI, OpenMP, or CUDA.

Summary

In this chapter, we looked at the spiral MLOps lifecycle and its different phases. We took another look at reducing technical debt from a holistic point of view after understanding each phase of the lifecycle. We discussed the philosophy behind taking a model-centric vs. a data-centric view of MLOps and why when working with big data, a data-centric view that encapsulates variability and complexity in the data may be preferable. We took a look at continuous delivery for stochastic systems and how we could adapt principles in this chapter to deploying Bayesian systems or more general types of stochastic systems along with some of the technical challenges. Finally, you did a hands-on lab, designing a CI/CD pipeline for the final toolkit that is a part of this book. Here is a summary of some of the topics we covered.

- Introducing the Spiral MLOps Lifecycle
- Reducing Technical Debt in Your Lifecycle
- The Various Levels of Schema Drift in Data Science
- Model Deployment
- Continuous Delivery for Stochastic Systems

In the final two chapters, we will diverge from the technical and hands-on components and instead take a deep dive into the ethical considerations around using AI and machine learning responsibly. We'll focus on model fairness, bias reduction, and policy that can minimize technical risk.