CHAPTER 1

Introducing MLOps

As data scientists we enjoy getting to see the impact of our models in the real world, but if we can't get that model into production, then the data value chain ends there and the rewards that come with having high-impact research deployed to production will not be achieved. The model will effectively be dead in the model graveyard, the place where data science models go to die.

So how do we keep our models out of this model graveyard and achieve greater impact? Can we move beyond simply measuring key performance indicators (KPIs) to moving them so that our models become the driver of innovation in our organization? It's the hypothesis of this book that the answer is yes but involves learning to become better data-driven, technical decision makers. In this chapter, I will define MLOps, but first we need to understand the reasons we need a new discipline within data science at all and how it can help you as a data scientist own the entire lifecycle from model training to model deployment.

What Is MLOps?

Imagine you are the director of data science at a large healthcare company. You have a team of five people including a junior data analyst, a senior software (data) engineer, an expert statistician, and two experienced data scientists. You have millions of data sets, billions of data points from thousands of clinical trials, and your small team has spent the last several sprints developing a model that can change real people's lives.

D. Sorvisto, *MLOps Lifecycle Toolkit*, https://doi.org/10.1007/978-1-4842-9642-4_1

You accurately predict the likelihood that certain combinations of risk factors will lead to negative patient outcomes, predict the posttreatment complication rate, and you use an inflated Poisson regression model to predict the number of hospital visits based on several data sources. Your models are sure to have an impact, and there's even discussion about bringing your research into a convolutional neural network used to aid doctors in diagnosing conditions. The model you created is finally at the cutting edge of preventative medicine. You couldn't be more excited, but there's a problem.

After several sprints of research, exploratory data analysis (EDA), data cleaning, feature engineering, and model selection, you have stakeholders asking some tough questions like, When is your model going to be in production? All of your code is in a Jupyter notebook, your cleaning scripts scattered in various folders on your laptop, and you've done so many exploratory analyses you're starting to have trouble organizing them.

Then, the chief health officer asks if you can scale the model to include data points from Canada and add 100 more patient features to expand into new services all while continuing your ad hoc analysis for the clinical trial. By the way, can you also ensure you've removed all PII from your thousands of features and ensure your model is compliant with HIPAA (Health Insurance Portability and Accountability Act)? At this point, you may be feeling overwhelmed.

As data scientists we care about the impact our models have on business, but when creating models in the real world, the process of getting your model into production so it's having an impact and creating value is a hard problem. There are regulatory constraints; industry-specific constraints on model interpretability, fairness, data science ethics; hard technical constraints in terms of hardware and infrastructure; and scalability, efficiency, and performance constraints when we need to scale our models to meet demand and growing volumes of data (in fact, each order of magnitude increase in the data volume leads to new architectures entirely).

MLOps can help you as a data scientist take control of the entire machine learning lifecycle end to end. This book is intended to be a rigorous approach to the emerging field of ML engineering, designed for the domain expert or experienced statistician who wants to become a more end-to-end data scientist and better technical decision maker.

The plan then is to use the language of data science, examples from industries and teach you the tools to build ML infrastructure; deploy models; set up feature groups, training pipelines, and data drift detection systems to accelerate your own projects; comply with regulatory standards; and reduce technical debt. Okay, so with this goal in mind, let's teach you the MLOps lifecycle toolkit, but first let us take the first step in a million-mile journey and define exactly what we mean by MLOps.

Defining MLOps

We need to define MLOps, but this is a problem because at the present time, MLOps is an emerging field. A definition you'll often hear is that MLOps is the intersection of people, processes, and technology, but this definition lacks specificity. What kind of people? Domain experts? Are the processes in this definition referring to DevOps processes or something else, and where in this definition is data science? Machine learning is but one tool in the data scientist's toolkit, so in some sense, MLOps is a bit of a misnomer as the techniques for deploying and building systems extend beyond machine learning (you might think of it as "Data Science Ops").

In fact, we are also not talking about one specific industry where all of the MLOps techniques exist. There is no one industry where all of the data science is invented first, but in fact each industry may solve data science problems differently and have its own unique challenges for building and deploying models. We revised the definition and came up with the following definition that broadly applies to multiple industries and also takes into account the business environment.

MLOps definition: MLOps (also written as ML Ops) is the intersection of industry domain experts, DevOps processes, and technology for building, deploying, and maintaining reliable, accurate, and efficient data science systems within a business environment. Figure 1-1 illustrates this.

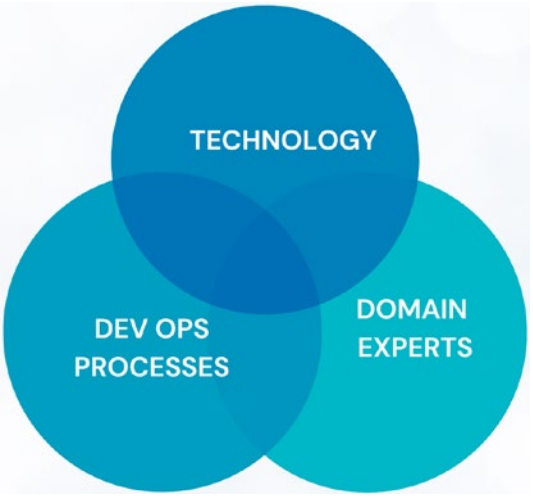

Figure 1-1. *Stakeholders will never trust a black box*

In machine learning, we solve optimization problems. We have data, models, and code. Moreover, the output of models can be non-deterministic with stochastic algorithms that are difficult to understand or communicate due to their mathematical nature and can lead stakeholders to viewing the system as an opaque "black box." This view of a machine learning system as a black box is a real barrier to trusting the system and ultimately accepting its output. If stakeholders don't trust your model, they won't use your model. MLOps can also help you track key metrics and create ***model transparency***.

This is the reason machine learning in the real world is a hard problem. Oftentimes the models themselves are a solved problem. For example, in the transportation sector, we know predicting the lifetime of a component in a fleet of trucks is a regression problem. For other problems, we may have many different types of approaches and even cutting-edge research

that has not been battle-tested in the real world, and as a machine learning engineer, you may be the first to operationalize an algorithm that is effectively a black box, which not only is not interpretable but may not be reliable. We need to be transparent about our model output and what our model is doing, and the first step is to begin measuring quality and defining what success means in your project.

If stakeholders don't trust your model, they won't use your model.

So how do we even begin to measure the quality of an MLOps solution when there is so much variability in what an MLOps solution looks like? The answer to this conundrum is the MLOps maturity model, which applies across industries.

MLOps Maturity Model

What does it mean to have "good" MLOps practices, and what is the origin of this term? You probably have several questions around MLOps and might be wondering what the differences are between MLOps and software development, so let us first discuss, in the ideal case, what MLOps looks like by presenting the maturity model in diagram form (Figure 1-2).

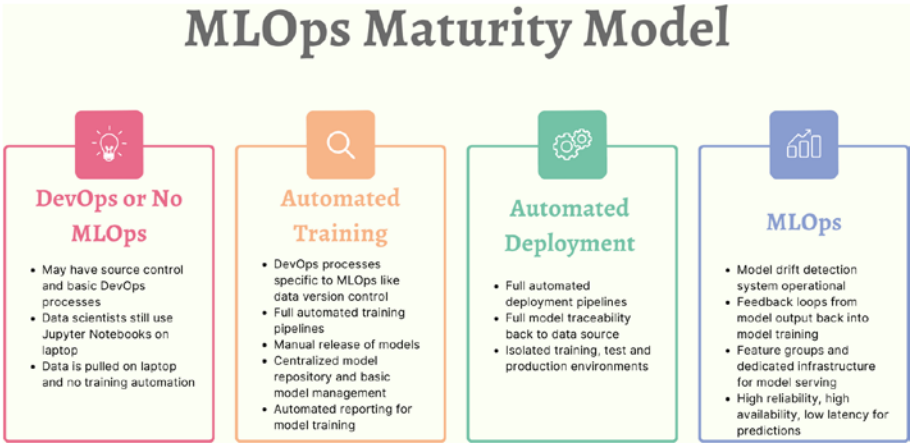

Figure 1-2. *MLOps maturity model*

Notice that the key differentiator from DevOps to stage 2 MLOps is the existence of an automated training pipeline with automated reporting requiring data management and infrastructure for reporting. The key differentiator from stage 2 to stage 3 is the existence of automated release pipelines (model management), and the differentiator between stage 3 and the final stage (full-blown MLOps) is we approach high reliability including specific subsystems to detect data and model drift and some way of creating feedback required to "move the needle" on key measurements we'll cover.

Brief History of MLOps

MLOps has its history in DevOps processes. DevOps, the merging of the words "Dev" and "Ops," changed the way software developers built, deployed, and maintained large-scale software systems. DevOps includes best practices for automating, continuous delivery, software testing and emphasizes developing software for the end user. However, if you notice the diagram of the MLOps maturity model, DevOps is not the same as MLOps, and it is not sufficient to apply DevOps principles to data science without some adaptation.

MLOps in contrast is about continuous delivery of machine learning systems but extends this idea to continuous training and reproducibility (an important part of the scientific process).

You may be thinking, well, software development is software development, which naturally encompasses parts of the software development lifecycle including infrastructure as code (creating reusable environments), containerization, CI/CD pipelines, and version and model control systems. However, this stuff does not automatically give you model management and data management best practices. Hence, MLOps takes DevOps a step further and can be thought of as a multi-dimensional version of DevOps that fuses together best practices from software engineering, ModelOps, and DataOps to move key metrics such as interpretability, accuracy, reliability, and correlation with key performance indicators specific to your industry.

Defining the Relationship Between Data Science and Engineering

Okay, we have defined MLOps, but it's important we have a clear idea on what we mean by data science and software engineering and the differences between them.

For the purpose of this book, we use data science as an umbrella term for a multidisciplinary approach to extracting knowledge from data that uses the scientific method. Data science uses machine learning as a tool. This is why we talk about stochastic systems instead of just machine learning systems since MLOps practices can apply to more than just machine learning, for example, operationalizing something more abstract like a causal model or Bayesian network or custom statistical analysis. In the next section, we will look at some general patterns for data science projects.

What Are the Types of Data Science Projects?

It is vital to understand the types of data science projects you might encounter in the real world before we dive into the *MLOps lifecycle*, which will be the focus of the rest of the book. We will look at supervised machine learning, semi-supervised machine learning, reinforcement learning, probabilistic programming paradigms, and statistical analysis.

Supervised Machine Learning

Supervised machine learning is a machine learning problem that requires labeled input data. Examples of supervised learning include classification and regression. The key is labeled data. For example, in NLP (natural language processing), problems can be supervised. You might be building a classification model using a transformer architecture to recommend products for new customers based on free-form data. Although you might know how to build a transformer model in PyTorch or TensorFlow, the challenge comes from labeled data itself. How do you create labels? Furthermore, how do you ensure these labels are consistent? This is a kind of chicken and egg problem that machine learning teams often face, and although there are solutions like Mechanical Turk, with privacy and data regulations like GDPR, it may be impossible to share sensitive data, and so data needs to be labeled internally, creating a kind of bottleneck for some teams.

Semi-supervised Machine Learning

In semi-supervised problems, we have a rule for generating labels but don't explicitly have labeled data. The difference between semi-supervised algorithms is the percentage of training data that is unlabeled. Unlike supervised learning where consistency of labels may be an issue, with semi-supervised the data may consist of 80% unlabeled data and a small

percentage, say 20%, of labeled data. In some cases like fraud detection in banking, this is a very important class of machine learning problems since not all cases of fraud are even known and identifying new cases of fraud is a semi-supervised problem. Graph-based semi-supervised algorithms have been gaining a lot of traction lately.

Reinforcement Learning

Reinforcement learning is a type of machine learning where the goal is to choose an action that maximizes a future reward. There are several MLOps frameworks for deploying this type of machine learning system such as Ray, but some of the challenge is around building the environment itself, which may consist of thousands, millions, or billions of states depending on the complexity of the problem being modeled. We can also consider various trade-offs between exploration and exploitation.

Probabilistic Programming

Frameworks like PyMC3 allow data scientists to create Bayesian models. Unfortunately, these models tend not to be very scalable, and so this type of programming for now is most often seen during hyper-parameter tuning using frameworks like Hyperopt where we need to search over a large but finite number of hyper-parameters but want to perform the sweep in a more efficient way than brute force or grid search.

Ad Hoc Statistical Analysis

You may have been asked to perform an "EDA" or exploratory data analysis before, often the first step after finding suitable data sources when you're trying to discover a nugget of insight you can act upon. From an MLOps perspective, there are important differences between this kind of "ad hoc" analysis and other kinds of projects since operationalizing an ad hoc analysis has unique challenges.

One challenge is that data scientists work in silos of other statisticians creating ad hoc analysis. Ad hoc analysis is kind of an all-in category for data science code that is one-off or is not meant to be a part of an app or product since the goal may be to discover something new. Ad hoc analysis can range from complex programming tasks such as attribution modeling to specific statistical analysis like logistic regression, survival analysis, or some other one-off prediction.

Another noteworthy difference between ad hoc analysis and other types of data science projects is ad hoc analysis is likely entirely coded in a Jupyter notebook either in an IDE for data scientists such as Anaconda in the case the data scientist is running the notebook locally or Databricks notebook in the cloud environment collaborating with other data scientists.

An ad hoc analysis may be a means to an end, written by a lone data scientist to get an immediate result such as estimating a population parameter in a large data set to communicate to stakeholders.

Examples of ad hoc analysis might include the following:

- Computing correlation coefficient
- Estimating feature importance for a response variable in an observational data set
- Performing a causal analysis on some time series data to determine interdependencies among time series components
- Visualizing a pairwise correlation plot for variables in your data set to understand dependence structure

The Two Worlds: Mindset Shift from Data Science to Engineering

It is no secret that data science is a collaborative sport. The idea of a lone data scientist, some kind of mythical persona that is able to work in isolation to deliver some groundbreaking insight that saves the company

from a disaster using only sklearn, probably doesn't happen all that often in a real business environment. Communication is king in data science; the ability to present analysis and explain what your models are doing to an executive or a developer requires to shift mindsets and understand your problem from at least two different perspectives: the technical perspective and the business perspective. The business perspective is talked about quite a bit, how to communicate results to stakeholders, but what about the other half of this? Communicating with other technical but non–data science stakeholders like DevOps and IT?

The topic of cross-team communication in data science crops up when requesting resources, infrastructure, or more generally in any meetings with non–data scientists such as DevOps, IT, data engineering, or other engineering-focused roles as a data scientist.

Leo Breiman, the creator of random forests and bootstrap aggregation (bagging), wrote an essay entitled "Statistical Modeling: The Two Cultures." Although Breiman may not have been talking about type A and type B data scientists specifically, we should be aware that in a multidisciplinary field like data science, there's more than one way to solve a problem and sometimes one approach, although valid, is not a good culture fit for every technical team and needs to be reframed.

What Is a Type A Data Scientist?

Typically a type A data scientist is one with an advanced degree in mathematics, statistics, or business. They tend to be focused on the business problem. They may be domain experts in their field or statisticians (both frequentist or Bayesian), but they might also come from an applied math or business background and be non-engineering.

These teams may work in silos because there is something I'm going to define as the *great communication gap* between type A and type B data scientists. The "B" in type B stands for *building* (not really, but this is how you can remember the distinction).

As data science matures, the distinction may disappear, but more than likely data scientists will split into more specialized roles such as data analyst, machine learning engineer, data engineer, and statistician, and it will become even more important to understand this distinction, which we present in Table 1-1.

Table 1-1. *Comparing Type A and Type B Data Scientists*

Type A Data Scientist	Type B Data Scientist
Focuses on understanding the process that generated the data	Focuses on building and deploying models
Focuses on measuring and defining the problem	Focuses on building infrastructure and optimizing models
Values statistical validity, accuracy, and domain expertise	Values system performance, efficiency, and engineering expertise

Types of Data Science Roles

Over the past decade, data science roles have become more specialized, and we often see roles such as data analyst, data engineer, machine learning engineer, subject matter expert, and statistician doing data science work to address challenges. Here are the types of data science roles:

- *Business analysts*: Problems change with the market and model output (called data drift or model drift). The correlation of model output with key business KPIs needs to be measured, and this may require business analysts who understand what the business problem means.

- *Big data devs*: Data volume can have properties such as volume, veracity, and velocity that transactional systems are not designed to address and require specialized skills.
- *DevOps*: Data needs to be managed as schemas change over time (called schema drift) creating endless deployment cycles.
- *Non-traditional software engineers*: Data scientists are often formally trained in statistics or business and not software engineering.

Even within statistics, there is division between Bayesians and frequentists. In data science there are also some natural clusters of skills, and often practitioners have a dominant skill such as software engineering or statistics.

Okay, so there's a rift even within statistics itself, but what about across industries? Is there one unified "data scientist"?

For example, geospatial statistics is its own beast with spatial dependence of the data unlike most data science workflows, and in product companies, R&D data scientists are highly sought after as not all model problems are solved and they require iterating on research and developing reasoning about data from axioms. For example, a retail company may be interested in releasing a new product that has never been seen on the market and would like to forecast demand for the product. Given the lack of data, novel techniques are required to solve the "cold start" problem. Recommender systems, which use a collaborative filtering approach to solve this problem, are an example, but oftentimes out-of-the-box or standard algorithms fall short. For example, slope-one, a naive collaborative filtering algorithm, has many disadvantages.

Hackerlytics: Thinking Like an Engineer for Data Scientists

The ability to build, organize, and debug code is an invaluable skill even if you identify as a type "A" data scientist. Automation is a key ingredient in this mindset, and we will get there (we cover specific MLOps tools like PySpark, MLflow, and give an introduction to programming in Python and Julia in the coming chapters), but right now we want to focus on concepts. If you understand the concept of technical debt, which is particularly relevant in data science, and the need to future-proof your code, then you will appreciate the MLOps tools and understand how to use them without getting bogged down in technical details. In order to illustrate the concept of technical debt, let's take a look at a specific example that you might have encountered when building a machine learning pipeline with real data.

Anti-pattern: The Brittle Training Pipeline

Suppose you work for a financial institution where you're asked by your data science lead to perform some data engineering task like writing a query that pulls in the last 24 months of historical customer data from an analytical cloud data warehouse (the database doesn't matter for this example; it could be anything like a SQL Pool or Snowflake). The data will be used for modeling consumer transactions and identifying fraudulent transactions, which are only 0.1% of the data.

You need to repeat this process of extracting customer transaction data and refreshing the table weekly from production so you have the latest to build important features for each customer like number of recent chargebacks and refunds. You are now faced with a technical choice: do you build a single table, or do you build multiple tables, one for each week that may make it easier for you to remember?

You decide to opt for this latter option and build one table per week and adjust the name of the table, for example, calling it something such as historical_customer_transactions_20230101 for transaction dates ending on January 1, 2023, and the next week historical_customer_transactions_20230108 for transactions ending on January 8, 2023. Unfortunately, this is a very brittle solution and may not have been a good technical decision.

What is brittleness? Brittleness is a concept in software engineering that is hard to grasp without experiencing its *future consequences.* In this scenario, our solution is brittle because a single change can break our pipelines or cause undue load on IT teams. For example, within six months you will have around 26 tables to manage; each table schema will need to be source controlled, leading to 26 changes each time a new feature is added. This could quickly become a nightmare, and building training pipelines will be challenging since you'll need to store an array of dates and think about how to update this array each time a new date is added. So how do we fix this?

If we pick the first option, a single table, can we make this work and eliminate the array of dates from our training pipeline and reduce the effort it takes to manage all of these tables? Yes, easily in this case we can add metadata to our table, something like a snapshot date, and give our table a name that isn't tethered to a specific datetime, something like historical_customer_transaction (whether your table name is plural or singular is also a technical decision you should establish early in your project). Understanding, evaluating, and making technical decisions like this comes with experience, but you can learn to become a better technical decision maker by applying our first MLOps tool: *future-proofing your code*.

Future-Proofing Data Science Code

As we discussed, a better way to store historical transaction data is to add an additional column to the table rather than in the table name itself (which ends up increasing the number of tables we have to manage and thus technical debt, operational risk, and weird code necessary to deal with the decision such as handling an unnecessary dynamic array of dates in a training pipeline).

From a DevOps perspective, this is fantastic news because you will reduce the IT load from schema change and data change down to a simple insert statement.

As you develop an engineering sense should be asking two questions before any technical decision:

- Am I being consistent? (Example: Have I used this naming convention before?)
- If I make this technical decision, what is the future impact on models, code, people, and processes?

Going back to our original example, by establishing a consistent naming convention for tables and thinking about how our naming convention might impact IT that may have to deploy 26 scripts to refresh a table, if we choose a poor naming convention such as table sprawl, code spiral, or repo sprawl, we'll start to see cause and effect relationships and opportunities to improve our project and own workload as well. This leads us to the concept of ***technical debt***.

What Is Technical Debt?

"Machine learning is the high interest credit card of technical debt."[1] Simply put, technical debt occurs when we write code that doesn't anticipate change. Each suboptimal technical decision you make now doesn't just disappear; it remains in your code base and will at some point, usually at the worst time (Murphy's Law), come back to bite you in the form of crashed pipelines, models that choke on production data, or failed projects.

Technical debt may occur for a variety of reasons such as prioritizing speed of delivery over all else or a lack of experience with basic software engineering principles such as in our brittle table example. To illustrate the concept of technical debt and why it behaves like real debt, let's consider another industry-specific scenario.

Imagine you are told by the CEO of a brick-and-mortar retail company that you need to build a model to forecast customer demand for a new product. The product is similar to one the company has released before, so there is data available, but the goal is to use the model to reduce costs of storing unnecessary product inventory. You know black box libraries won't be sufficient and you need to build a custom model and feature engineering library.

Your engineering sense is telling you that building a custom solution will require understanding various trade-offs. Should you build the perfect model and aim for 99% accuracy and take a hit on performance? Does the business need 99% accuracy, or will forecasting demand with 80% accuracy be sufficient to predict product inventory levels two weeks in advance?

[1] Machine Learning: The High Interest Credit Card of Technical Debt

Hidden Technical Trade-Offs in MLOps

In the previous example, we identified a ***performance-accuracy trade-off*** (Figure 1-3) that is one of many trade-offs you'll face in technical decision making when wearing an MLOps hat. MLOps is full of these hidden technical trade-offs that underlie each technical decision you make. By understanding the kinds of trade-offs, you can reduce technical debt instead of accumulating it. We've summarized some common trade-offs in data science:

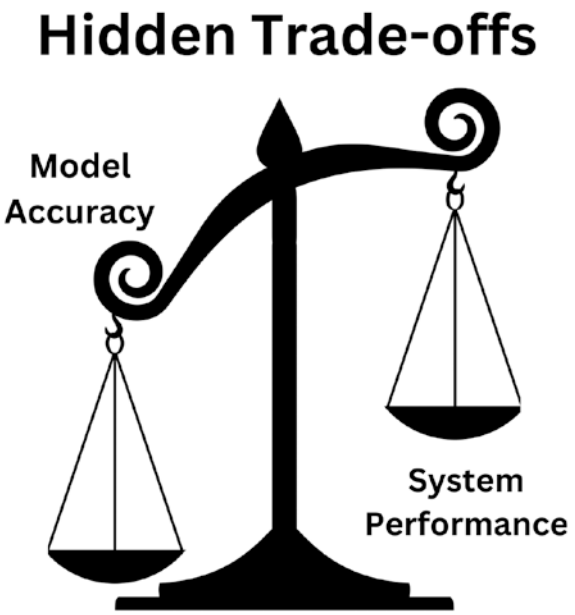

Figure 1-3. *Data science projects have many hidden technical trade-offs*

- Data volume vs. model accuracy (more data is better, but each 10× scale-up requires new infrastructure)
- Technical correctness vs. cognitive complexity (data science code has high cognitive complexity especially when handling every possible edge case, which can cause performance bottlenecks)

- Model accuracy vs. model complexity (do you really need to use deep learning, or is a decision tree–based model that is 90% accurate sufficient for the first iteration?)

How to Protect Projects from Change

Change is everywhere! Change is also inevitable and can be the seed of innovation in data science, so it has its upsides and downsides. The downsides are change can increase technical debt and, if not properly managed, can cause failed data science projects.

So where does it come from? What are the main drivers of change in data science? Change can come from stakeholders or market conditions, problem definition, changes in how a KPI is measured, or schema changes. You need to learn to protect your data science projects from change.

Software engineering principles are about learning to protect against change and be future thinking and so can be applied to data science. Writing clean code has little to do with being a gatekeeper or annoyance but is a necessary part of building a reliable system that isn't going to crash on Sunday's nightly data load and cause your team to be called in to troubleshoot an emergency.

Maybe the most basic example of shielding from change in data science is the concept of a view. A view is an abstraction, and as software engineers we like abstractions since they allow us to build things and depend on something that is stable and unchanging such as the name of a view, even if what is being abstracted, the query itself, the schema, and the data underneath, is constantly changing.

Managing views, source control, and understanding when to apply abstractions are something that can come with practice, but understanding the value of a good abstraction will take you a long way in shielding your own code from change and understanding some of the reasons technical decisions are made without becoming frustrated in the process.

There are abstractions for data like views we can use to manage changes in data along with other tools like data versioning, but there are also abstractions for models and code like source control and model registries like MLflow. We'll talk about all of these MLOps tools for managing change in subsequent chapters, but keep in mind the concept of abstractions and how these tools help us protect our project from change.

Drivers of Change in Data Science Projects

We know the types of approaches needed to build an attribution model, but there is no one way to build one without historical data, and the types of approaches may involve more than something you're familiar with like semi-supervised learning; it may involve, instead, stochastic algorithms or a custom solution. For attribute modeling we could think about various techniques from Markov chains to logistic regression of Shap values.

From a coding perspective, for each type of approach, we are faced with a choice as a designer of a stochastic learning system on programming language, framework, and tech stack to use. These technologies exist in the real world and not in isolation, and in the context of a business environment, change is constantly happening.

These combinatorial and business changes, called change management, can cause disruptions and delays in project timelines, and for data science projects, the line is often a gap between what the business wants and the problem that needs well-defined requirements or at worst an impossible problem or one that would require heroic efforts to solve within the time and scope.

So model, code, well-defined requirements… What about the data? We mentioned the business is constantly changing, and this is reflected in the data and the code. It is often said that a company ships its own org chart, and the same is true for data projects where changes to business entities cause changes in business rules or agreement upon ways to measure

KPIs for data science projects, which leads to intense downstream change to feature stores, feature definitions, schema changes, and downstream pipelines for model training and inference.

Externalities or macro-economic conditions may also cause changes in customer assumptions and behavior that get reflected in the model, a problem often called concept drift. These changes need to be monitored and acted upon (e.g., can we retrain the model when we detect concept drift), and these actions need to be automated and maintained as packages of configuration, code, infrastructure as code, data, and models. Artifacts like data and models require versioning and source control systems, and these take knowledge of software engineering to set up.

Choosing a Programming Language for Data Science

Arguments over programming languages can be annoying, especially when this leads to a holy war instead of which programming language is the right tool for the job. For example do you want performance, type safety, or the ability to rapidly prototype with extensive community libraries?

This is not a programming book, and it's more important that you learn how to think conceptually, which we will cover in the next chapter. Python is a very good language to learn if you are starting out, but there are other languages like Julia and R and SQL that each have their uses. You should consider the technical requirements, skill set of your team before committing to a language. For example, Python has distinct advantages over R for building scalable code, but when it comes to speed, you might consider an implementation in Julia or C++. This also comes with a cost: the data science community for Python is prolific, and packages like Pandas and sklearn are widely supported. This isn't to say using a language like R or Julia is wrong, but you should make a decision based on available data.

More advanced data scientists interested in specializing in MLOps may learn C++, Julia, or JAX (for accelerating tensor operations) in addition to Python and strong SQL skills.

We'll cover programming basics including data structures and algorithms in the following chapter. It's worth noting that no one language is best and new languages are developed all the time. In the future, functional programming languages oriented around probabilistic programming concepts will likely play a bigger role in MLOps.

MapReduce and Big Data

Big data is a relative term. What was considered "big data" 20 years ago is different from what is considered "big data" today. The term became popular with the advent of MapReduce algorithms and Google's Bigtable technology, which allowed algorithms that could be easily parallelized to be run over extremely large gigabyte-scale data sets. Today, we have frameworks like Spark, and knowledge of MapReduce and Java or Scale isn't necessary since Spark has a Python API called PySpark and abstracts the concept of a mapper and reducer from the user. However, as data scientists we should understand when we're dealing with "big data" as there are specific challenges. Not all "big data" is high volume. In fact there are three Vs in big data you may need to handle.

Big Data a.k.a. "High Volume"

This is most commonly what is meant by "big data." Data that is over a gigabyte may be considered big data, but it's not uncommon to work with billions of rows or terabytes of data sourced from cold storage. High-volume data may pose operational challenges in data science since we need to think about how to access it and transferring large data sets can

be a problem. For example, Excel has a famous 1 million row limit, and similarly with R, the amount of memory is restricted to 1 GB, so we need tools like Spark to read and process this kind of data.

High-Velocity Data

By "high-velocity" data sources, we usually mean streaming data. Streaming data is data that is unbounded and may come from an API or IoT edge device in the case of sensor data. It's not the size of the data that is an operational challenge but the speed at which data needs to be stored and processed. There are several technologies for processing high-velocity data including "real-time" or near-real-time ones (often called micro-batch architecture) like Apache Flink or Spark Streaming.

High-Veracity Data

If you are a statistician, you know the concept of variability. Data can have variability in how it's structured as well. When data is unstructured like text or semi-structured like JSON or XML in the case of scraping data from the Web, we refer to it as high veracity. Identifying data sources as semi-structured, structured, or unstructured is important because it dictates which tools we use and how we represent the data on a computer, for example, if dealing with structured data, it might be fine to use a data frame and Pandas, but if our data is text, we will need to build a pipeline to store and process this data and you may even need to consider NoSQL databases like MongoDB for this type of problem.

Types of Data Architectures

We might choose a data architecture to minimize change like a structured data warehouse, but when it comes to data science projects, the inherent inflexibility of a structured schema creates a type of impedance mismatch. This is why there are a number of data architectures such as the data lake,

data vault, or medallion architecture that may be better fit for data science. These architectures allow for anticipated changes in schema, definition, and business rules.

The Spiral MLOps Lifecycle

We will discuss the titular MLOps lifecycle in detail in Chapter 7, but we can broadly distinguish the different phases of a data science project into

1. Insight and data discovery
2. Data and feature engineering
3. Model training and evaluation
4. Model deployment and orchestration

The reason we call it a spiral lifecycle is because each of these stages may feedback into previous stages; however, as the technical architecture matures (according to the MLOps maturity model), the solution should converge to one where you are delivering continuous value to stakeholders. Figure 1-4 shows the spiral MLOps lifecycle.

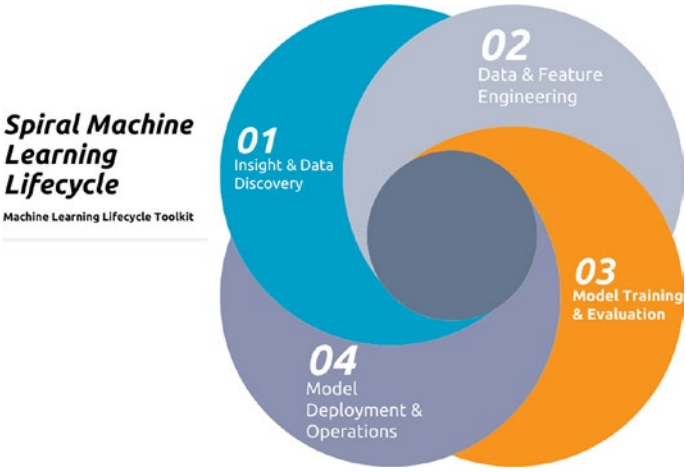

Figure 1-4. *The spiral MLOps lifecycle*

Data Discovery

Ideally, you would approach the problem like a statistician. You have a problem, design an experiment, choose your sampling method, and collect the exact data you need to ensure there are no biases in your data, repeating the process as many times as you can if need be. Unfortunately, business does not work like this. You will be forced to approach the problem backward; you might have a vague idea of what problem you want to solve and the data you need, but you may only have the resources or access to the data that the business has already collected.

How can MLOps help with this process? Well, the data is often stored as flat CSV files, which can total several gigabytes or more. A typical scenario is you have to figure out how to load this data and you quickly run into memory errors if using tools like Pandas. We'll show you how to leverage distributed computing tools like Databricks to get around these issues without too much headache and if possible without even rewriting your code.

Data Discovery and Insight Generation

This phase is all about exploring your data sets, forming and testing hypotheses, and developing intuition for your data. There are often several challenges at this stage. If you are using a tool like seaborn or matplotlib for generating visualizations, you might face the challenge of how to deploy your work or share it with other data scientists or stakeholders.

If you're working on an NLP problem, you might have lots of different experiments and want to quickly test them. How do you organize all of your experiments, generate metrics and parameters, and compare them at different points in time? Understanding standard tools like MLflow and how to set up an experimentation framework can help.

Let us suppose as a data scientist you are an expert at understanding and identifying biases in your data. You work for a financial institution and are tasked with creating a customer churn model. You know net promoter score is a key metric, but you notice survey responses for your customers are missing on two key dates.

You decide to write a script in Pandas to filter out these two key dates. Suddenly, the next week your data has doubled in size, and your script no longer scales. You manually have to change the data in a spreadsheet and upload it to a secure file server. Now, you spend most of your time on this manual data cleaning step, validating key dates and updating them in a spreadsheet. You wish you knew how to automate some of these steps with a pipeline.

Few people enjoy doing ETL or ELT; however, understanding the basics of building pipelines and when to apply automation at scale especially during the data cleaning process can save you time and effort.

Data and Feature Engineering

Feature selection may be applied to both supervised and unsupervised machine learning problems. In the case that labeled data exists, we might use some measure like correlation to measure the degree of independence between our features and a response or target variable. It may make sense to remove features that are not correlated with the target variable (a kind of low-pass filter), but in other cases, we may use a model itself like Lasso or Ridge regression or random forest to decide which features are important.

Model Training

How do you choose the best model for your problem? A training pipeline in the wild can take hours or even sometimes days to finish especially if the pipeline consists of multiple steps.

As an MLOps engineer, you should be familiar with frameworks and methods for speeding up model training. We'll look at Hyperopt, a framework for using Bayesian hyperparameter search, and Horovod for distributed model training that takes advantage of parallelism. By speeding up model training time, by using distributed computing or GPU, we can immediately add value to a project and have more time spent doing data science.

Model Evaluation

Model selection is the process of choosing the "best" model for a business problem. Best may not necessarily be as simple as the model with the best training accuracy in case the data overfits and does not generalize to new samples (see the bias-variance trade-off). It may be more nuanced than this, as there might be regulatory constraints such as in the healthcare and banking industries where model interpretability and fairness are a concern. In this case, we must make a technical decision, balancing the attributes of the model we desire like precision, recall, and accuracy over how interpretable or fair the model is and what kind of data sources we are legally allowed to use to train the model. MLOps can help with this step of the machine learning lifecycle by automating the process of model selection and hyper-parameter tuning.

Deployment and Ops

You've trained your model and even automated some of the data cleaning and feature engineering processes, but now what? Your model is not creating business value unless it's deployed in production, creating insights that decision makers can take action on and incorporate into their tactical or business strategy.

But what does it mean to deploy a model to production? It depends. Are you doing an online inference or batch inference? Is there a requirement on latency or how fast your model has to make predictions? Will all the features be available at prediction time, or will we have to do some preprocessing to generate features on the fly?

Typically infrastructure is involved; either some kind of container registry or orchestration tool is used, but we might also have caches to support low-latency workflows, GPUs to speed up tensor operations during prediction, or have to use APIs if we deploy an online solution. We'll cover the basics of infrastructure and even show how you can deliver continuous value from your models through continuous integration and delivery pipelines.

Monitoring Models in Production

Okay, you've deployed your model to production. Now what? You have to monitor it. You need a way to peer underneath the covers and see what is happening in case something goes wrong. As data scientists we are trained to think of model accuracy and maybe have an awareness of how efficient one model is compared with another, but when using cloud services, we must have logging and exception handling for when things go wrong. Again, Murphy's Law is a guiding principle here.

Understanding the value of setting up logging and explicit exception handling will be a lifesaver when your model chokes on data in production it has never seen before. In subsequent chapters you'll learn to think like an engineer to add logging to your models and recover gracefully in production.

Example Components of a Production Machine Learning System

A production machine learning system has many supporting components that go into its design or *technical architecture*. You can think of the technical architecture as a blueprint for creating the entire machine learning system and may include cloud storage accounts, relational and nonrelational (NoSQL) databases, pipelines for training and prediction, infrastructure for reporting to support automated training, and many other components. A list of some of the components that go into creating a technical architecture include

- **Cloud storage accounts**
- **Relational or NoSQL databases**
- **Prediction pipeline**
- **Training pipeline**
- **Orchestration pipelines**
- **Containers and container registries**
- **Python packages**
- **Dedicated servers for training or model serving**
- **Monitoring and alerting services**
- **Key Vault for secure storage of credentials**
- **Reporting infrastructure**

Measuring the Quality of Data Science Projects

The goal of this section is to give you the ability to quantitatively define and measure and evaluate the success of your own data science projects. You should begin by asking what success means for your project.

This quantitative toolbox, akin to a kind of multi-dimensional measuring tape, can be applied to many types of projects from traditional supervised, unsupervised, or semi-supervised machine learning projects to more custom projects that involve productionizing ad hoc data science workflows.

Measuring Quality in Data Science Projects

With the rapid evolution of data science, a need has arisen for MLOps, which we discussed, in order to make the process effective and reproducible in a way that mirrors scientific rigor.

Measuring software project velocity and other KPIs common to project management, an analogous measurement is needed for data science. Table 1-2 lists some measurements that you might be interested in tracking for your own project. In later chapters we'll show you how to track these or similar measures in production using tools like MLflow so that you can learn to move the needle forward.

Table 1-2. *Common KPIs*

Measurement	Stakeholder Question	Examples
Model accuracy	Can we evaluate model performance?	Precision, recall, accuracy, F1 score; depends on the problem and data
Model interpretability	How did each feature in the model contribute to our prediction?	Shap values
Fairness	Are there sensitive features being used as input into the model?	Model output distribution
Model availability	Does the model need to make predictions at any time of day or on demand? What happens if there is downtime?	Uptime
Model reliability	Do the training and inference pipelines fail gracefully? How do they handle data that has never been seen before?	Test coverage percentage
Data drift	What happens when features change over time?	KL divergence
Model drift	Has the business problem changed?	Distribution of output of model
Correlation with key KPIs	How do the features and prediction relate to key KPIs? Does the prediction drive key KPIs?	Correlation with increased patient hospital visits for a healthcare model
Data volume	What is the size of our data set?	Number of rows in feature sets

(*continued*)

Table 1-2. (*continued*)

Measurement	Stakeholder Question	Examples
Feature profile	What kinds of features and how many?	Number of features by category
Prediction latency	How long does the user have to wait for a prediction?	Average number of milliseconds required to create features at prediction time

Importance of Measurement in MLOps

How can we define the success of our projects? We know intuitively that the code we build should be reliable, maintainable, and fail gracefully when something goes wrong, but what is reliability and maintainability? We need to take a look at each of these concepts and understand what each means in the context of data science.

What Is Reliability?

Reliability means there are checks, balances, and processes in place to recover when disaster strikes. While availability is more about uptime (is the system always available to make a prediction when the user needs it?), reliability is more about acknowledging that the system will not operate under ideal conditions all the time and there will be situations that are unanticipated. Since we cannot anticipate how the system will react when faced with data it's never seen before, for example, we need to program in error handling, logging, and ensure we have proper tests in place to cover all code paths that could lead to failure. A cloud logging framework and explicit exception handling are two ways to make the system more reliable, but true reliability comes from having redundancy in the system, for example, if you're building an API for your model, you should consider a load balancer and multiple workers.

What Is Maintainability?

Maintainability is related to code quality, modularity, and readability of the code. It requires a future-oriented mindset since you should be thinking about how you will maintain the code in the future. You may have an exceptional memory, but will you be able to remember every detail a year from now when you have ten other projects? It's best to instead focus on maintainability early on so running the project in the future is easier and less stressful.

Moving the Needle: From Measurement to Actionable Business Insights

Ultimately the goal of MLOps is to move the needle forward. If the goal of the data scientist is to create accurate models, it's the job of the machine learning engineer to figure out how to increase accuracy, increase performance, and move business KPIs forward by developing tools, processes, and technologies that support actionable decision making.

Hackerlytics: The Mindset of an MLOps Role

Finally, I would like to close out this chapter by discussing the mindset shift required for an MLOps role from data scientist. If you are a data scientist focused on data and analytics, you are probably used to thinking outside the box already to solve problems. The shift to using this out-of-the-box thinking to solve data problems with technology is exactly the mindset required. In the next chapter, we'll look at the fundamental skills from mathematical statistics to computer science required so you can begin to apply your new hackerlytics skills on real data problems.

Summary

In this chapter we gave an introduction to MLOps from the data scientist's point of view. You should now have an understanding of what MLOps is and how you can leverage MLOps in your own projects to bring continuous value to stakeholders through your models. You should understand some of the technical challenges that motivate the need for an MLOps body of knowledge and be able to measure the quality of data science projects to evaluate technical gaps. Some of the majors topics covered were as follows:

- What is MLOps?
- The need for MLOps
- Measuring quality of data science projects

In the next few chapters, we will cover some core fundamentals needed for data scientists to fully take ownership of the end-to-end lifecycle of their projects. We'll present these fundamentals like algorithmic and abstract thinking in a unique way that can help in the transition from data science to MLOps.