Preface

It's 9.25pm and the soft glow of Dana's computer screen glared into her bleary eyes as she logged on to continue fixing an error – the source of a splash of red across her screen. She had made and had her dinner, done her everyday chores, but her mind wasn't really there – it was in a few places in fact.

It's been a full-on day – she was in the depths of debugging why the model's performance just wouldn't improve, despite various tweaks to the data and model architecture. Her day was peppered with switches between long training runs and back-and-forth messages with the support team on customer queries on why their loan applications got denied. The occasional stack traces didn't help either.

She was tired, and the tangled heap of uncommitted code changes sitting on her local machine added to the latent cognitive load that was bubbling over in her mind. But she felt she had to keep going – her team had already missed the initial release date by four months, and the executives' impatience was showing. What made things worse was a fear that her job might be on the line. 1 in 10 of people – several of whom she knew – were laid off in the latest round of cost-cutting measures.

Everyone on her team was well-meaning and capable, but getting bogged down every day in a quagmire of tedious testing, anxiety-laden production deployments, stepping through illegible and brittle code – that wore them down within a few months. She and her team were doing their level best, but sometimes it feels like they're building a house without a foundation – things keep falling apart. She doesn't know it yet, but they weren't set up for success.

Many individuals begin their machine learning (ML) journey with great momentum and gain confidence quickly, thanks to the growing ecosystem of tools, techniques, tutorials, and community of ML practitioners. However, when we graduate beyond the controlled environment of tutorial notebooks and Kaggle competitions into the space of real-world problems, systems, and people, many inevitably struggle to realize the potential of AI in the real world due to various unforeseen traps and unanticipated detours.

When we peel back the glamorous claims of being the sexiest job of the 21st century, we often see ML practitioners mired in burdensome manual work, complex and brit-

tle codebases, and ultimately frustration resulting from Sisyphean ML experiments that never see the light of day in production.

In 2019, it was reported that 87% of data science projects never make it to production. According to Algorithmia's Enterprise AI/ML Trends, even among companies who have successfully deployed ML models in production, 64% of survey respondents take more than a month to deploy a new model in 2021, an increase from 56% in 2020. They also found that 38% of organizations surveyed are spending more than 50% of their data scientists' time on model deployment.

These barriers impede – or, in some cases, even prevent – ML practitioners from applying their expertise in ML to deliver on the value and promise of AI for customers and businesses. Even amidst the growing interest in data and AI, there are, more so than before, existential questions such as: is data science a dying profession?

But the good news is – it doesn't have to be this way. In the past few years, we've had the privilege to work on various data and ML projects and collaborate with ML practitioners from various industries. While there are barriers and pains as we have outlined above, there are also better paths, practices, and systems of work that allow ML practitioners to reliably deliver ML-enabled products into the hands of customers.

That's what this book is all about. We'll draw from our experience to distill a set of enduring principles and practices that consistently helps us to effectively deliver ML solutions in the real world. These practices work because they bring a holistic approach and go beyond just ML to create essential feedback loops in various subsystems (e.g. product experience, engineering, data, delivery processes) and enable teams to fail quickly and safely, experiment rapidly, and deliver reliably.

Who Is This Book For

> There [are] two meanings to ability, not one: a fixed ability that needs to be proven, and a changeable ability that can be developed through learning.
>
> —Carol S. Dweck, Mindset

> Whether you think you can, or you think you can't – you're right.
>
> —Henry Ford

Whether you're a ML practitioner in academia, enterprise organizations, start-ups, scale-ups, or consulting, the principles and practices in this book can help you and your team identify opportunities for improvement and be more effective at what you do.

Data scientists and ML engineers
: The job scope of a data scientist has evolved over the past few years. Instead of purely focusing on modeling techniques and data analysis, we're seeing expect-

ations (implicit or explicit) that one needs the capabilities of a full-stack data scientist: data wrangling, ML engineering, MLOps, business case formulation, among others. This book will elaborate on the essential capabilities (beyond ML) which are required in designing and delivering ML solutions in the real world.

We have presented the principles, practices and hands-on exercises in this book to various groups of ML practitioners (data scientists, ML engineers, PhD students, software engineers) over several years, and we've consistently received positive feedback. The ML practitioners that we have worked with in the industry said that they benefited from improvement in feedback cycles, flow and reliability that comes from practices such as automated testing and refactoring. Our takeaway from that is that there is a desire from the ML community to learn these skills and practices, and this is our attempt to scale the sharing of this knowledge.

Software engineers and infrastructure engineers

When we run workshops on this topic, we have noticed another persona or class of people who are working in the ML space – and that is software engineers, infrastructure engineers, or platform engineers. While capabilities from the software world (e.g. infrastructure as code, deployment automation, automated testing, etc.) are necessary in designing and delivering ML solutions in the real-world, they are not sufficient.

To create reliable ML solutions, we need to be able to properly see the entire problem space. To do so, we will supplement the software lens of looking at the world with other principles and practices, such as ML model tests, dual-track delivery, shifting ethics left, among others, to handle challenges that are unique to ML.

Product managers, delivery managers, engineering managers

We set ourselves up for failure if we think that we only need data scientists and ML engineers to build an ML product. In contrast, our experience tells us that teams are most effective when they are cross-functional and equipped with the necessary ML, data, engineering, product and delivery capabilities. A data scientist can pair with a software engineer to write readable, modular, well-factored code with automated tests. An ML engineer can pair with a data scientist to set up MLOps tooling to run large-scale training on cloud infrastructure, with automated tests and deployments on CI/CD pipelines. An experience designer can facilitate product discovery and user testing sessions to bridge the gap between the customers' needs and our ML and software know-how.

In this book, we will elaborate on how you can apply Lean delivery practices and systems thinking to create structures that help teams to focus on the voice of the customer, shorten feedback loops, experiment rapidly and reliably, and iterate towards building the right thing. As Edwards Deming once said, "A bad system will beat a good person every time". So we will share principles and practices that

will help teams create a structure that optimizes information flow, reduce waste (e.g. handoffs, dependencies), and improve the flow of value.

If we've done our job right, this book will invite you to look closely at how things have "always been done" in ML and in your teams, reflect on how well they are working for you, and consider better alternatives. Read this book with an open mind, and – for the engineering-focused chapters – with an open code editor. As Peter M. Senge said, "taking in information is only distantly related to real learning. It would be nonsensical to say, 'I just read a great book about bicycle riding—I've now learned that.'" Try out the practices, and we hope you will experience first-hand the value that they bring us in our real-world projects.

Approach this book with a continuous improvement mindset, not a perfectionist mindset. There is no perfect project, with perfect scenarios and where everything works perfectly without challenges. There will always be complexity and challenges (a healthy amount of challenges is essential for growth), but the practices in this book will help you to minimize accidental complexity so that you can focus on the essential complexity of your ML solutions and on delivering value responsibly.

How This Book Is Organized

Chapter 1: Challenges and Better Paths in Delivering Machine Learning Solutions is a mini distillation of the entire book. We explore high-level and low-level reasons for why and how ML projects fail, and we lay out a more reliable path for delivering value in ML solutions by adopting Lean delivery practices across five key disciplines: product, delivery, machine learning, engineering, and data.

In the remaining chapters, we will describe practices of effective ML teams and ML practitioners. In Part 1: Engineering Practices, we cover practices that help ML practitioners in their day-to-day work (e.g. automated testing, refactoring, using the code editor effectively). In Part 2: Product and Delivery Practices, we elaborate on practices in other subsystems which are necessary for delivering ML solutions, such as product thinking, delivery practices, team topologies, continuous delivery, and MLOps.

Part 1: Engineering Practices

Chapters 2 and 3: Effective Dependency Management describes principles and practices – along with a hands-on example that you can code along with – for creating consistent, reproducible, secure, production-like runtime environments for running your code. Instead of getting trapped in dependency hell, the practices in this chapter will enable you and your teammates to "check out and go" and create consistent environments effortlessly.

Chapters 4 and 5: Automated Testing – Move Fast Without Breaking Things provides you with a rubric for testing components of your ML solution – be they software tests, model tests, or data tests. We will demonstrate how automated tests help us shorten our feedback cycles and reduce the tedious effort of manual testing, or worse, fixing production defects that slipped through the cracks of manual testing. We will also describe the limits of the software testing paradigm on ML models, and how ML fitness functions and behavioral tests can help us scale the automated testing of ML models.

Chapter 6: Supercharging Your Code Editor with Simple Techniques will show you how to configure your code editor (PyCharm or VS Code) to help you code more effectively. After we've configured our IDE in a few steps, we'll go through a series of keyboard shortcuts that can help you to automate refactoring, automatically detect and fix issues, navigate your codebase without getting lost in the weeds, among others.

Chapter 7: Refactoring – Getting Out Of Our Own Way. In this chapter, we draw from the wealth of software design to help us design readable, testable, maintainable, and evolvable code. In the spirit of "learning by doing", you'll see how we can take a problematic, messy and brittle codebase, and apply refactoring techniques to iteratively improve our codebase to a modular, tested, and readable state.

Part 2: Product and Delivery Practices

Chapter 8: MLOps and Continuous Delivery for Machine Learning (CD4ML) will articulate an expansive view of what MLOps and CI/CD (continuous integration and continuous delivery) really entails (spoiler alert: it's more than automating model deployments and defining CI pipelines). We lay out a blueprint for the unique shape of CI/CD for ML projects and walk through how you can set up each component in this blueprint to create reliable ML solutions and free up your teammates from repetitive and undifferentiated labor so that they can focus on other higher-value problems.

Chapter 9: A Systems Thinking Approach to Improving ML Delivery covers other subsystems (namely product and delivery) which are essential for delivering ML solutions rapidly and reliably. We will discuss product discovery techniques that help us ideate, validate and eventually converge so that we set ourselves up for success by starting with the most valuable problems. We will also go through delivery practices that help us measure delivery progress and learn how to identify and manage project risks. We will address the unique challenges resulting from the experimental and high-uncertainty nature of certain ML problems, and discuss techniques such as the dual-track delivery model that help us iterate more quickly in shorter cycles.

Chapter 10: Building Blocks of Effective ML Teams. In this chapter, we switch gears to focus on team topologies and the people aspect of effective teams. We will describe

principles and practices that help create a safe, human-centric, and growth-oriented team. We will also discuss organizational dynamics, cultural modes, team topologies, and how teams can work towards the culture that they want. We will lay out concrete principles and practices that you can use to nurture a culture of collaboration, effectiveness, and learning.

Chapter 11: Responsible AI – Theory and Practice will describe mental models, techniques, and practices for putting the Responsible AI framework into practice. Beyond philosophical discussions (which are important), we will also provide a framework for shifting ethics left and identifying any potential risk of harm or ethical issues in the ML solutions that we create. We will demonstrate how automated tests and automated quality checks from earlier chapters can help us test and prevent harm.

Some Parting Thoughts

There are three things we wanted to muse on before wrapping up the preface.

First, we want to acknowledge that machine learning is more than just supervised learning. We can also solve data-intensive (and even even data-poor) problems using other optimization techniques (e.g. reinforcement learning, operations research, simulation), data engineering, and data analysis approaches. In addition, machine learning is not a silver bullet and some problems can be solved without machine learning.

Even though we've chosen a supervised learning problem (loan default prediction) as an anchoring example, the principles and practices in this book are useful even beyond supervised learning. For example, the chapters on automated testing, dependency management, code editor productivity are useful even in reinforcement learning. The chapters on product thinking are useful for exploratory phases of a product or problem space.

Second, on the role of culture: ML effectiveness and the practices in this book are not – and cannot be – a solo effort. That's why we've titled the book as Effective Machine Learning *Teams*. You can't be the only person writing tests, for instance. In organizations that we've worked with, individuals become most effective when there is a cultural alignment (within the team, department and even organization) on these Lean and agile practices. This doesn't mean that you need to boil the ocean with the entire organization, and it's also saying that it's not enough to go it alone. As Steve Jobs once said, "Great things in business are never done by one person. They're done by a team of people."

Finally, this book is not about productivity (how to ship as many features, stories, or code as possible), nor is it about efficiency (how to ship features, stories, or code at the fastest possible rate). Rather, it's about effectiveness – how to ship the right features and stories reliably and responsibly. This book is about finding balance through movement – and moving in effective ways. The principles and practices in

this book have consistently helped us to successfully deliver ML solutions, and we are confident that they will do the same for you as well.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You

do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Effective Machine Learning Teams* by David Tan and Ada Leung (O'Reilly). Copyright 2024 David Tan Rui Guan and Ada Leung Wing Man, 978-1-098-14463-0."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/effective-machine-learning-teams*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*

Follow us on Twitter: *https://twitter.com/oreillymedia*

Watch us on YouTube: *https://youtube.com/oreillymedia*

CHAPTER 1

Challenges and Better Paths in Delivering Machine Learning Solutions

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the first chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@oreilly.com*.

> The most dangerous kind of waste is the waste we do not recognize.
>
> —Shigeo Shingo, leading expert on the Toyota Production System

> Not everything that is faced can be changed, but nothing can be changed until it is faced.
>
> —James Baldwin, writer and playwright

We kick off this chapter with the dual reality of promise and disappointment in ML in the real world. We then examine both high-level and day-to-day challenges that cause ML projects to fail. Finally, we outline a better path based on the principles and practices of Lean delivery, product thinking, and agile engineering. This chapter is somewhat of a miniature representation of the remainder of this book.

Regardless of your background – be it academia, data science, ML engineering, product management, software engineering, etc. – if you are building products or systems that involve ML, you will likely face some of the challenges that we describe in this chapter. This chapter is our attempt to distill our experience – and the experience of others – in building and delivering ML-enabled products in the real world. We hope that these principles and practices will help you avoid unnecessary pitfalls and find a more reliable path for your journey.

Machine Learning: Promises and Disappointments

While ML promises to solve complex challenges for us and automatically optimize everything from customer experiences to internal processes, the lived experience of many ML practitioners suggest that the majority of ML projects either fail to make it into production or become tediously challenging to evolve once they are in production.[1] Even amidst the excitement around the generative language models arms race of 2023, as interest groups and professionals collectively stress tested tools such as ChatGPT, its limitations quickly emerged as we observed patterns of behavior in producing generic middle-of-the-read positions, making stuff up, and even writing code that has "a high rate of being incorrect".

As hype cycles go, we travel past the peak of inflated expectations and crash-land into the trough of disillusionment. We might see some high-performing AI teams move onto the plateau of productivity and wonder if we might ever get there ourselves.

In this section, we will look at the evidence suggesting that investments and interest in ML are continuing to grow in the near future, and then we'll take a deep dive into the engineering, product, and delivery bottlenecks that impede the returns on our investments.

Continued Optimism in Machine Learning

Putting aside the hype and our individual coordinates on the hype cycle for a moment, data shows that ML continues to be a fast-advancing field that provides many techniques for solving real-world problems. Stanford's AI Index Report 2022 found that in 2021, global private investment in AI totaled around $93.5 billion, which is *more than double* the total private investment even in 2019, before the COVID-19 pandemic. McKinsey's State of AI survey indicates that AI adoption is continuing its steady rise: 56 percent of all respondents report AI adoption in at least one function, up from 50 percent in 2020.

1 There are several surveys that support this claim, and we will refer to specific surveys in the following section when we explore the challenges of ML in detail.

The Stanford report also found companies continuing to invest in applying a diverse set of ML techniques (e.g. natural language understanding, computer vision, reinforcement learning, etc.) across a wide array of sectors (e.g. healthcare, retail, manufacturing, financial services, etc.). From a jobs and skills perspective, Stanford's analysis of millions of job postings since 2010 showed that the demand for ML capabilities has been growing steadily year-on-year in the past decade, even through and after the COVID-19 pandemic.

While these trends are reassuring from an opportunities perspective, they are also highly worrying if we journey ahead without confronting and learning from the challenges that have ensnared us – both the producers and consumers of ML-driven solutions – in the past. Let's take a look at these challenges in detail.

Why ML Projects Fail

The lived experiences of many ML practitioners suggest that the journey of delivering ML solutions is ridden with traps, quicksand, and even seemingly insurmountable barriers. Despite the plethora of chart-topping Kaggle notebooks, it's common for ML projects to fail in the real world.

Just to be clear – we're not trying to avoid failures. Drawing an analogy to education, failure is as valuable as it is inevitable. There's lots that we can learn from failure. The problem arises as the cost of failure increases – missed deadlines, unmet business outcomes, and sometimes even collateral damage: harm to humans and loss of jobs and livelihoods of many employees who aren't even directly related to the ML initiative.

What we want is to fail in a low-cost way, and often, so that we improve our odds of success for everyone who has a stake in the undertaking. In this section, we'll look at some common challenges – spanning product, delivery and engineering – that contribute to the likelihood of failure, and in the next section, we'll explore principles and practices that help us fail safely and deliver rapidly and reliably.

Let's start at the macro-level and then we'll zoom in to look at day-to-day barriers to the flow of value.

Macro-level view: barriers to success

At the macro-level – i.e. the level of an ML project or a program of work – we've heard and seen ML projects fail to achieve their desired outcomes due to the following challenges.

Failing to solve the right problem or deliver value for users

In this failure mode, even if we have all the right engineering practices and "build the thing right", we fail to move the needle on the intended business outcomes because we neglected to "build the right thing". ML practitioners often focus on

training, improving and deploying ML models, but it's common for ML teams to overlook user testing or user journey mapping exercises to identify the pains and needs of the users (and ergo why users would bother to use the product that we're building).[2]

Challenges in productionizing models

Many ML projects do not see the light of day in production. A 2021 Gartner poll of roughly 200 business and IT professionals found that only 53% of AI projects make it from pilot into production, and among those that succeed, it takes an average of nine months to do so."[3] The challenge isn't just around compute (e.g. model deployment), but also around data (e.g. the model's data dependencies – usable inference data, available at the right latency.)

Challenges after productionizing models

Once in production, iterative experimentation and deployments of improved models can be overly tedious. In the 2021 State of Enterprise ML report, Algorithmia reported that 64% of companies take *more than one month* to deploy a new model (an increase from 58%, as reported in the 2020 report). 38% of organizations spend more than 50% of their data scientists' time on deployment —and that only gets worse with scale.

Brittle and convoluted codebases

Feedback loops in the model development lifecycle are long and tedious, and divert valuable time from important ML product development work. The primary way of knowing if everything works might be to manually run a training notebook or script, wait for it to complete (sometimes waiting up to hours), and manually eyeballing some model metrics to know if the model is still as good as before. This doesn't scale well and more often than not, we encounter unexpected errors during development and even in production, and waste time troubleshooting issues.

In addition, the codebase is generally full of code smells (e.g. badly named variables, tightly coupled spaghetti code) and is difficult to understand (and therefore difficult to change).

Data quality issues in production

We'll illustrate this point with an example: A study in the British Medical Journal found that *none* of the hundreds of predictive tools that were developed to help

2 It's worth noting that identifying the wrong customer problem to solve is not unique to ML, and any product is susceptible to this.

3 As this Gartner survey is a small survey comprising only 200 people, there's likely to be high variance in the number of ML projects that never got delivered across regions, industries and companies. Take the specific number with a dash of salt, and try to relate it to your qualitative experience. Have you personally experienced or heard of ML projects that, even after months of investment, were never shipped to users?

hospitals detect COVID-19 actually worked. There were many reasons for the failure of these models, and one key theme was on data quality. There was data leakage (which caused the models to appear better than they really are), mislabelled data, distributional asymmetry between training data and actual data in production, among others.

To compound the problem, the aforementioned challenges in retraining, re-evaluating, re-testing, and redeploying models in an automated fashion further inhibits our ability to respond to changing data distributions in time.

Inadequate data security and privacy
: Data security and privacy is a cross-cutting concern that should be the responsibility of everyone in the organization, from product teams to data engineering teams and every team in between. From an ML perspective, there are several unique data security and privacy challenges in the context of ML that can cause a project to fail. One such challenge is data poisoning, which involves injecting malicious or biased data into the training set to corrupt the model. For example, the famous (or infamous) Microsoft Tay chatbot, which was taken down within a day of release because it learned inflammatory and offensive content from users who deliberately attempted to train it to produce such responses.

Ethically problematic ML products
: One needn't look far to see how ML can go wrong in the wild. For example, Amazon developed an ML-driven tool to support its recruitment process, and it turned out that the model penalized resumes containing the word "women" (They decommissioned the tool within a year of its release). In another example, a benchmark analysis found that an ML software that was used to predict recidivism had *twice as high a false positive rate* for black defendants as for white defendants, and twice as high a false negative rate for white defendants.

Now that we've painted a macro-level picture of the reasons that can cause ML projects to fail, let's take a look at the day-to-day challenges that make it harder for ML projects to succeed.

Micro-level view: everyday impediments to success

At the micro-level – i.e. the level of delivering a feature in an ML project – there are several bottlenecks that impede our ability to execute on our ideas.

This story is best told by contrasting the lifecycle of a user story in the agile development lifecycle under two conditions: a *low effectiveness* environment and *a high effectiveness* environment. In our experience, these roadblocks present themselves not just in aspects of ML and engineering, but also in the areas of suboptimal ways of working and unplanned work.

Lifecycle of a story in a low effectiveness environment. Let's journey with Dana – an ML practitioner – in this scenario. The character is fictional but the pain is real.

- Dana starts their day having to deal immediately with alerts for problems in production, or customer support queries on why the model behaved in a certain way.
- Dana checks a number of logging and monitoring systems to triage the issue as there are no aggregated logs across systems. They manually prod the model to find an explanation for why the model produced that particular prediction for that customer. They vaguely remember that there was a similar customer query last month, but cannot find any internal documentation on how to resolve such customer queries.
- Dana also sends a reminder on the team chat to ask for a volunteer to review a pull request they created last week, so that it can be merged.
- Dana finally finds some time to code and picks up a task from the team's wallboard.
- The codebase doesn't have any automated tests, so after making some code changes, Dana needs to restart and re-run the entire training script or notebook, wait for the duration of model training (which could be minutes or hours), and hope that it runs without errors. They also manually eyeball some print statements at the end to check that the model metric hasn't declined. Sometimes, the code blows up midway because of an error that slipped in during development.
- While coding, Dana received comments and questions on the pull request. For example, one comment was that a particular function was too long and hard to read. Dana then switches contexts, types out a response (without necessarily updating the code) for coding design decisions they made last week, and mentions that they will create a story card to refactor this long function next time.
- After investing a few days of effort in a solution (without pair programming), they share it back with the team. A teammate candidly shares that the story wasn't actually high priority and the new code changes didn't provide clear business value. Or perhaps the solution introduced too much complexity to the codebase and needed to be re-written.

Dana ended the week feeling frustrated and demotivated. The long feedback cycles and context switching (between doing ML and other burdensome tasks) was a bottleneck to how much they could achieve. It also had a real cognitive cost that made them feel exhausted and unproductive. They sometimes log on again after office hours because they feel the pressure to finish the work and there just wasn't enough time in the day to complete the work – both the tedious tasks and the important tasks which are done in a tedious way.

Long feedback loops at each micro-level step leads to an overall increase in cycle time, which leads to fewer experimentation or iteration cycles in a day (see Figure 1-1). Work and effort often move backwards and laterally between multiple tasks, which lead to a disrupted state of flow.

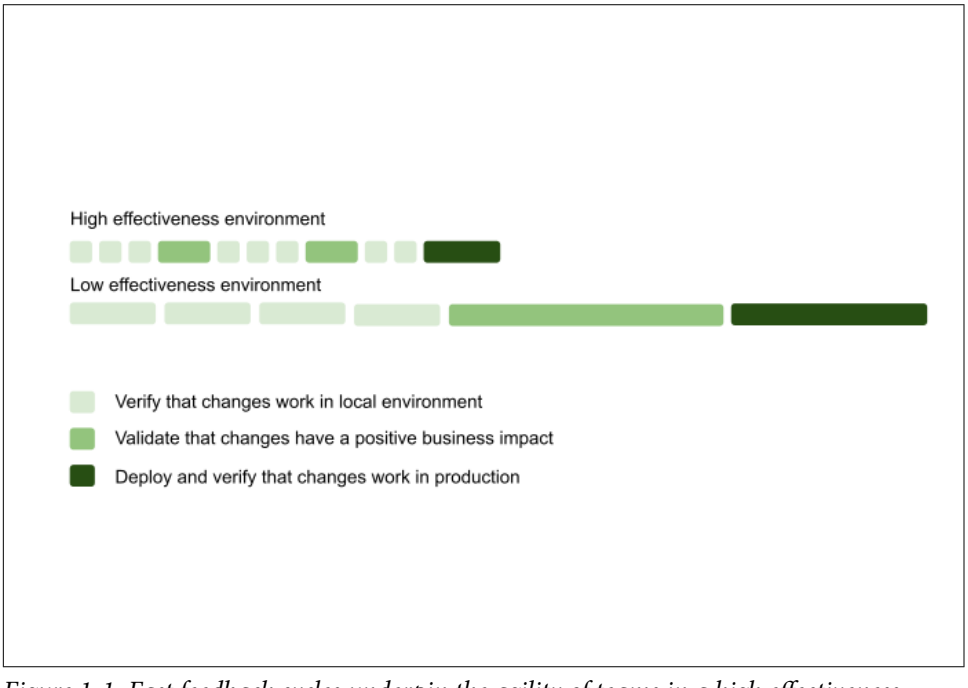

Figure 1-1. Fast feedback cycles underpin the agility of teams in a high effectiveness environment. (Adapted from Maximizing Developer Effectiveness, Tim Cochran)

Lifecycle of a story in a high effectiveness environment. Now, let's take a look at how different things can be for Dana in a high effectiveness environment:

- Dana starts the day by checking the team project management tool and then attends standup where they can pick up a story card. Each story card articulates its business value (which has been validated from a product perspective) and provides clarity about what they have to work on with a clear definition of done.
- Dana pairs with a teammate to write code to solve the problem specified in the story card. As they are coding, they provide each other with real-time feedback (e.g. to write readable and maintainable code), help catch each other's blindspots and share knowledge along the way.
- As they code, each incremental code change is quickly validated (within seconds or minutes) by running automated tests – both existing tests and new tests that they write. They run end-to-end ML model training pipeline locally on a small

dataset and get feedback on whether everything is still working fine within a minute.

- If they need to do a full ML training, they can trigger training on elastic compute infrastructure from their local machine with their local code changes, without the need to "push to know if something works". Model training will then commence in an environment with the necessary access to production data and elastic compute resources.
- They commit the code change, which then passes through a number of automated checks on the continuous integration and continuous delivery (CI/CD) pipeline before triggering full ML model training, which can take between minutes to hours depending on the ML model architecture and the volume of data.
- They focus on their task for a few hours, peppered with regular breaks, coffee, and even walks (separately).
- When the model training completes, a model deployment pipeline is automatically triggered. The deployment pipeline runs model quality tests and checks if the model is above the quality threshold for a set of specified metrics (e.g. accuracy, precision, etc.). If the model is of a satisfactory quality, the newly trained model artifact is automatically packaged and deployed to a pre-production environment, and the CI/CD pipeline also runs post-deployment tests on the freshly deployed artifact.
- When the story card's definition of done is satisfied, they inform the team, call for a brief (e.g. 20-minute) team huddle to share context with the team and demonstrate how the solution meets the definition of done. If they had missed anything, any teammate could provide feedback there and then.
- Otherwise, if no further development work is needed, another teammate then puts on the "testing hat" and brings a fresh perspective when testing if the solution satisfies the definition of done. The teammate can do exploratory and high-level testing within a reasonable timeframe because most, if not all, of the acceptance criteria in the new feature have been tested via automated tests.
- Whenever business wants to, they can deploy the change gradually to users in production, while monitoring business and operational metrics. Because the team has maintained a good test coverage, when the pipeline is all green, they can deploy the new model to production without any feelings of anxiousness.

The team makes incremental progress on the delivery plan every day, and team velocity is higher and stabler than in the low-effectiveness environment. Work and effort generally flows forward, and Dana leaves work feeling satisfied and with wind in their hair. Huzzah!

To wrap-up the tale of two velocities, let's look at the feedback mechanisms and compare the cycle time of each feedback mechanism (see Table 1-1).

Table 1-1. Comparison of feedback mechanisms and time to feedback in low- and high-effectiveness environments

Action	Feedback loops and time to feedback (in approximate orders of magnitude)	
	High-effectiveness environment	Low-effectiveness environment
Testing if code changes worked as expected	Automated testing *(~ seconds to minutes)* ●●	Manual testing (~ minutes to 10s of minutes) ●●●●
Testing if ML training pipeline works end to end	Training smoke test *(~ 1 minute)* ●●	Full model training (~ minutes to hours, depending on the model architecture) ●●●●●
Getting feedback on code changes	Pair programming *(~ seconds to minutes)* ●●	Pull request reviews (~ hours to days) ●●●●●●●
Understanding if application is working as expected in production	Monitoring in production *(~ seconds - as it happens)* ●	Customer complaints (~ days, or longer if not directly reported) ●●●●●●●

Now that we've painted a picture of common pitfalls in delivering ML solutions and a more effective alternative, let's explore if it's possible for teams to mature from a low-effectiveness environment to a high-effectiveness environment.

Is There a Better Way? How Lean and Systems Thinking Can Help

> A bad system will beat a good person every time.
>
> — Edwards Deming

The toil, frustration, and burnout that ML practitioners often face are a sign that our system of work can be improved. In this section, we'll put on a systems thinking lens to identify subsystems (we also call them "disciplines" in this book) that are required for effective ML delivery. Then we'll look to Lean for principles and practices that can help us operate these subsystems in an interconnected way that reduces waste and maximizes the flow of value.

But First, You Can't "MLOps" Your Problems Away

One reflexive but misguided approach to improving the effectiveness of ML delivery is for organizations to turn to MLOps practices and ML platforms. While they may be necessary, they are definitely not sufficient. In the world of software delivery, you can't "DevOps" or "platform" your problems away. DevOps helps to optimize and manage one subsystem (relating to infrastructure, deployment, and operations), but

other subsystems (e.g. software design, user experience, software delivery lifecycle) are just as important in delivering great products.

Likewise, in machine learning, *you can't "MLOps" your problems away*. No amount of MLOps practices and platform capabilities can save us from the complexity and toil that result from the lack of software engineering practices (e.g. automated testing, well factored design) and product delivery practices (e.g. user testing, measuring delivery metrics). MLOps or ML platforms aren't going to write comprehensive tests for you, talk to users for you, or reduce the negative impacts of team silos for you.

In a study on 150 successful ML-driven customer-facing applications at Booking.com, done through rigorous randomized controlled trials, the authors concluded that the key factor for success is *an iterative, hypothesis-driven process, integrated with other disciplines, such as product development, user experience, computer science, software engineering, causal inference, among others*. This finding is aligned with our approach as well, based on our experience delivering multiple ML and data products. We have seen time and again that delivering successful ML projects requires a multidisciplinary approach across *product, engineering, data, machine learning,* and *delivery* (see Figure 1-2).

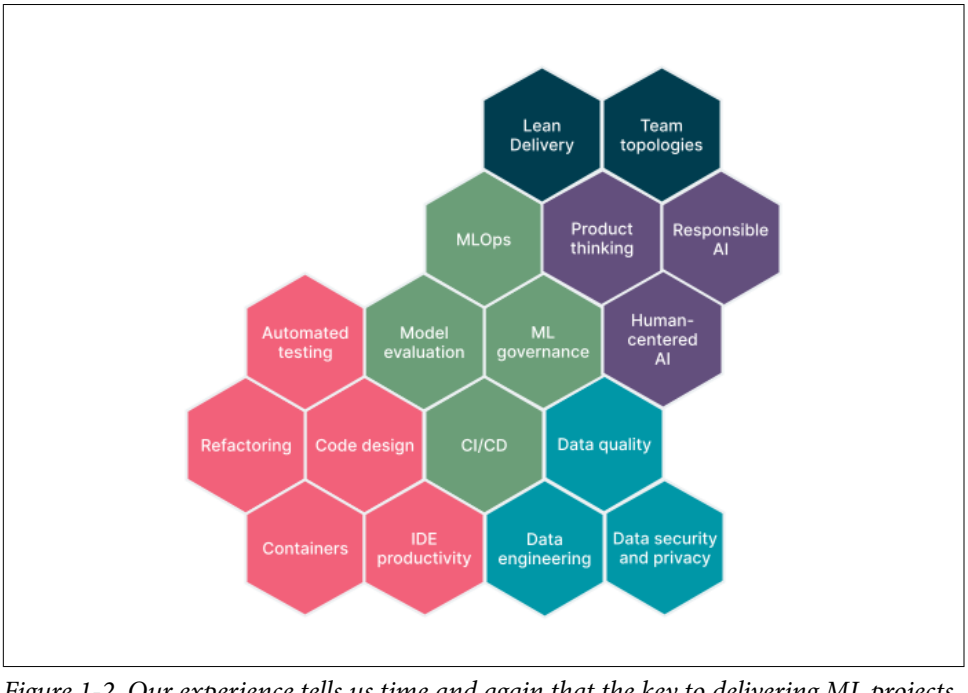

Figure 1-2. Our experience tells us time and again that the key to delivering ML projects successfully is a multidisciplinary approach across product, delivery, engineering, data, and machine learning.

To help us see the value of putting these five disciplines together – or the costs of focusing only on some disciplines while ignoring others – we can put on the lens of systems thinking. In the next section, we'll uncover the interconnected disciplines required to effectively deliver an ML product.

See the Whole: A Systems Thinking Lens for Effective ML Delivery

Systems thinking helps us shift our focus from individual part(s) of a system to relationships and interactions between all the components that constitute a system. As the author states in the linked article, systems thinking gives us mental models and tools for understanding – and eventually changing – structures that are not serving us well, including our mental models and perceptions.

You may be asking, why should we frame ML product delivery as a system? And well, what even *is* a system? Donella Meadows, a pioneer in systems thinking, defines a system as an interconnected set of elements that is coherently organized in a way that achieves something. A system must consist of three kinds of things: *elements*, *interconnections* and *a function or purpose*.

Let's read that again in the context of delivering ML products. A system must consist of three kinds of things: elements (e.g. ML experimentation, software engineering, infrastructure and deployment, data pipelines, delivery iterations, user experience), interconnections (e.g. a cross functional team) and a function or purpose (*deliver value, solve problems*) (see Figure 1-3). Our ability to see and optimize information flow in these interconnections determines our probability of success in delivering ML products. In contrast, teams that frame ML product delivery solely as a data and ML problem are more likely to fail because the true nature of the system (for example, user experience is key in determining product success) will eventually catch up and reveal itself to us.

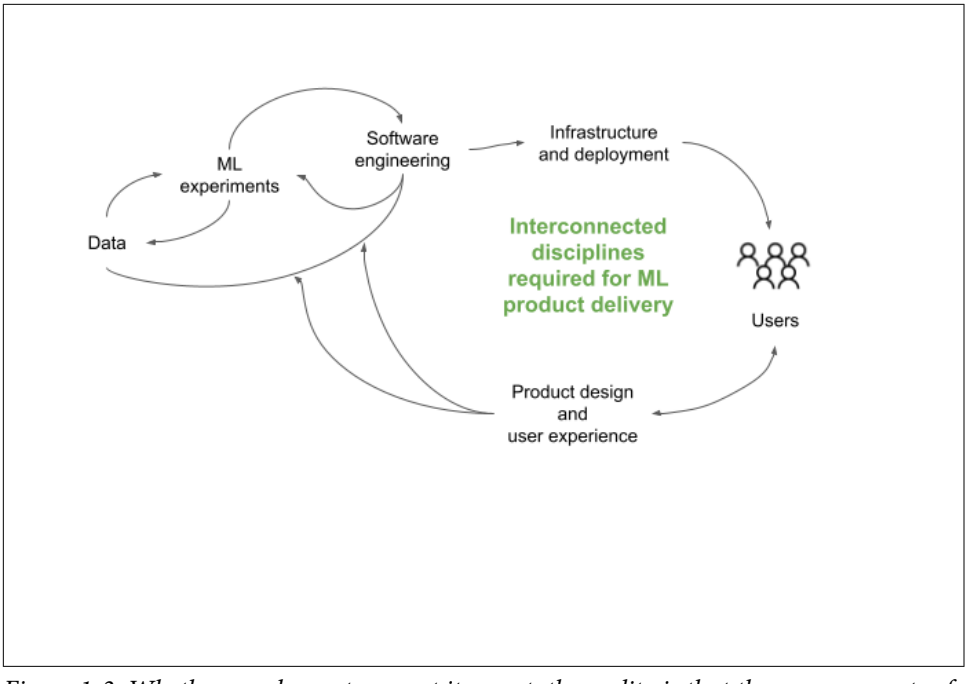

Figure 1-3. Whether we choose to accept it or not, the reality is that these components of ML product delivery are inherently interconnected.

Let's look at a concrete example of how a systems thinking tool (a causal loop diagram) can help us make sense of a common challenge in ML: turnover and attrition among ML practitioners. Using this example, we will illustrate systems thinking concepts such as *feedback loops*, *interconnectedness, and circularity* (as opposed to linearity).

In Figure 1-4, we see a reinforcing feedback loop starting with tedious tasks (e.g. data munging, manual testing), which increases an ML practitioner's workload significantly and contributes to delays in tasks and in the project. The manual nature of tedious tasks (e.g. manual testing) also means that errors will slip through the crack sometimes, which leads to production defects and creates a reinforcing feedback loop (a.k.a. vicious cycle) that creates even more delays (e.g. due to time needed to resolve production issues). This contributes to the feeling of dissatisfaction, frustration, and finally turnover.

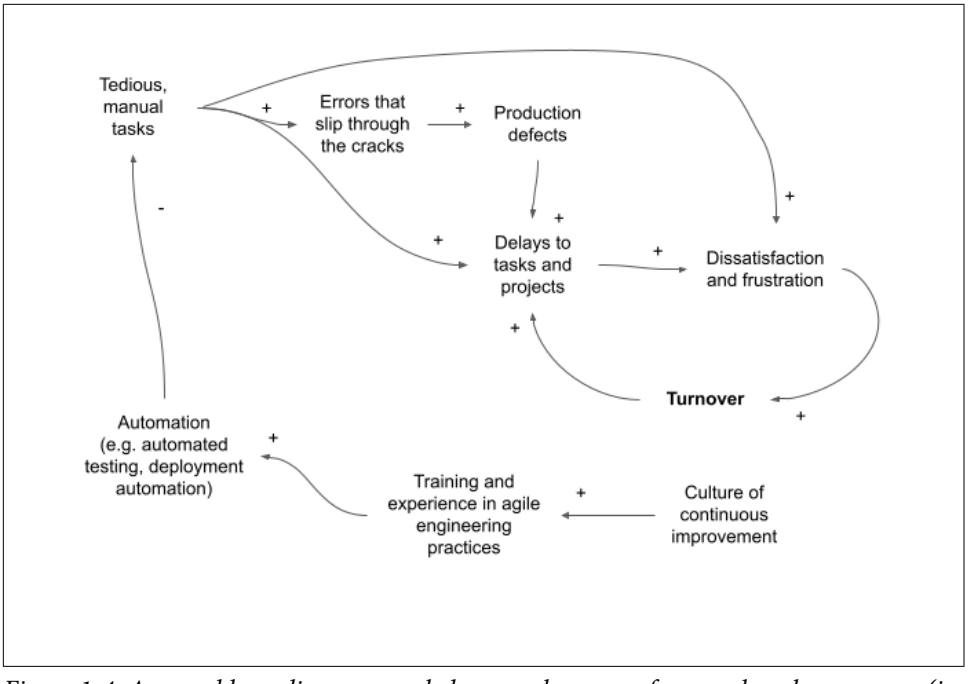

Figure 1-4. A causal loop diagram can help us make sense of a complex phenomenon (in this example, why ML practitioners quit) by zooming out to see the events and structures that lead to patterns of behavior.

A constant turnover could create another reinforcing feedback loop, and further contribute to project delays. This is because rehiring and ramp-up of new ML practitioners has a cost in terms of time and morale as teams continually restart the inevitable, but necessary, stages of group development as described in Tuckman's "storming-forming-norming-performing" model. The turnover also has a disruptive effect on the Nonaka-Takeuchi cycle of knowledge creation, which posits that tacit knowledge is grown into institutional knowledge through the cycle of socialization, externalization, combination and internalization (SECI).

One way to bring equilibrium or balance to this situation would be to introduce practices that reduce tedious manual tasks and improve quality, such as automated testing and deployment automation. These practices reduce tedious undifferentiated labor and free up ML practitioners for interesting higher-level work, which helps to increase satisfaction and reduce the likelihood of turnover.

Lastly, a key driver for bringing these agile engineering practices into ML is usually a culture of continuous improvement – the constant curiosity to look to other disciplines and high performing teams to find ways to improve our own effectiveness. In

contrast, a culture of "this is how ML has always been done" can be a blocker for the cross-pollination of these engineering practices into the world of ML.

Zooming back out, systems thinking recognizes that a system's components are interconnected and that changes in one part of the system can have ripple effects throughout the rest of the system. This means that to truly understand and improve a system, we need to consider the system *as a whole* and how all of its parts work together.

Thankfully, there is a philosophy that can help us improve information flow in the interconnections between the elements of an ML delivery system, and that is *Lean*.

Using Lean to Improve ML Delivery Systems

In this section, we'll start with a crash course of what Lean is and how it can help us deliver ML products more effectively. Then we'll enumerate each of the five disciplines which are required in ML delivery, and describe the key principles and practices that provide the fast feedback that ML teams need to iterate towards building the right product.

As a quick caveat, each of these five disciplines warrants a book – if not a collection of books – and the principles and practices we lay out in this chapter are by no means exhaustive. Nonetheless, they form a substantial start and they are principles and practices that we would bring to any ML project to help us deliver ML solutions effectively. This section will chart our path at a high level, and we will dive into details in the remaining chapters of the book.

What is Lean, and why should ML practitioners care?

In ML projects (as with many other software or data projects), it's common for teams to experience various forms of waste. For example, you may have invested time and effort to get a feature "done", only to realize eventually that the feature did not have demonstrable value for the customer. You may have experienced back-and-forth handoffs and waiting between one or more teams in order to release a feature to customers. Or maybe you've had your flow unexpectedly disrupted by a defect or bug in your codebase.[4] All of these wastes contribute to negative outcomes such as release delays and missed milestones, more work (and the feeling that there just isn't enough time to finish all the work), stress, and consequently low team morale.

4 Lean helpfully provides a nuanced classification of waste, also known as the "eight deadly wastes", which enumerate common inefficiencies that can occur in the process of delivering value to customers. The three examples in this paragraph refer to overproduction, waiting, and defects respectively. The remaining five types of waste are: transport, overprocessing, inventory, motion and under-utilized talent.

If you have experienced any of these negative outcomes, first of all, welcome to the human condition. These are challenges we've personally experienced and will continue to experience to some extent because no system can be 100% waste-free or noise-free. With that said, Lean principles and practices can help us continuously get better at what we do and enable us to *minimize waste and maximize value*.

By identifying and eliminating waste, Lean helps organizations to increase value and efficiency and reduce costs, leading to a more competitive market position. Lean also focuses on the importance of customer feedback in driving continuous improvement. By involving the voice of the customer in the development and delivery process, teams can better understand their needs and build relevant products for customers.

Lean practices originated from Toyota in the 1950s and was initially known as the Toyota Production System (TPS). James P. Womack and Daniel T. Jones later refined and popularized it as Lean principles in their book "The Machine That Changed the World" (see Figure 1-5). The following Lean principles were key in transforming the automotive industry, manufacturing, and IT, among other industries.

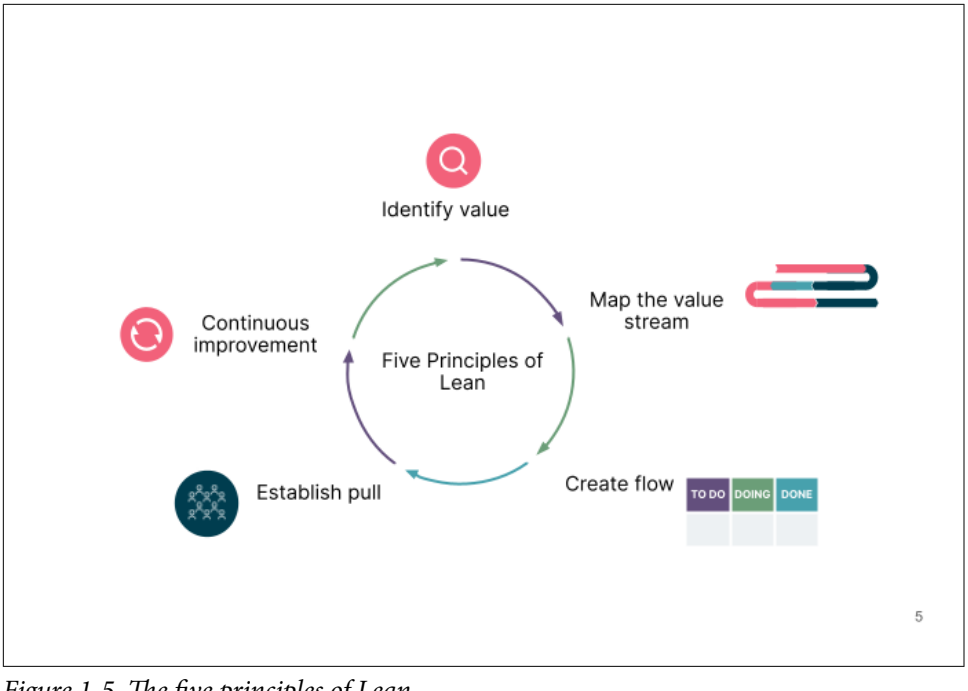

Figure 1-5. The five principles of Lean

Principle 1: Identify value
: Determine what is most valuable to the customer and focus on maximizing that value

Principle 2: Map the value stream
: Identify the steps in the process that add value and eliminate those that do not

Principle 3: Create flow
: Streamline the process to create a smooth and continuous flow of work

Principle 4: Establish pull
: Use customer demand to trigger production and avoid overproduction

Principle 5: Continuous improvement
: Continuously strive for improvement and eliminate waste in all areas of the value chain

In our experience delivering ML products, Lean steers us towards value-creating work, which then creates a positive feedback loop of customer satisfaction, team morale and delivery momentum. For example, instead of "pushing" out features because they are technically interesting, we first identify and prioritize features that will bring the most value to users (principle 1) and "pull" it into our delivery flow when the demand has been established (principle 4). In contrast, in instances where we didn't practice this, we'd end up investing time and effort to complete a feature that added complexity to the codebase without any no demonstrable value. To those with keen Lean eyes, yes – you've just spotted waste!

Value stream mapping (principle 2) is a tool that lets us visually represent all the steps and resources involved in delivering a unit of value (e.g. a feature in a product) to customers. Teams can use this tool to identify waste, work towards eliminating waste, and improve the flow of value (principle 3).

To map your team or product's value stream, you can follow these steps:

1. Identify the product or service being mapped. This could be a single product or an entire process.
2. Identify the current state map. Create a visual representation of the current process, including all steps and materials (including time and labor) involved from raw materials to finished product.
3. Identify value-added and non-value-added activities. Determine which steps add value to the product or service and which do not.
4. Identify waste. Look for areas of overproduction, waiting, defects, overprocessing, excess inventory, unnecessary motion, excess transport, unnecessary use of raw materials, and unnecessary effort.
5. Create a future state map. Based on the analysis of the current state map, redesign the process to eliminate waste and create a more efficient flow of materials and information.

6. Implement changes. Put the redesigned process into practice and continuously monitor and improve (principle 5)

Now that we have a basic working knowledge of Lean, let's look at how Lean intersects with the five disciplines to help us shorten feedback loops and enable us to rapidly iterate towards a valuable product. When put together, these practices help create several emergent characteristics in our system of delivering ML products: faster feedback, cheaper failures, predictable delivery, and most importantly, valuable outcomes.

Product

The product discipline is the most important to effective ML delivery because without it, none of the other disciplines (e.g. ML, data, engineering) matters. When we don't understand the business model and the users' needs, we find ourselves in a vacuum that is quickly filled with unsubstantiated assumptions, which tends to lead to teams over-engineering unvalidated features, and ultimately waste. A product-oriented approach helps ML teams begin with the end in mind, and continuously test our assumptions, and ensure that they are building solutions that are relevant to the needs of their users.

With the Lean mindset, we recognize that all our ideas are based on assumptions that need to be tested and that many of these assumptions may be proven wrong. Lean provides a set of principles and practices to test our hypotheses, for example through prototype testing, safe-to-fail experiments, build-measure-learn, among others. Each experiment gives us valuable learnings that help us make informed decisions to persevere, pivot or stop. By pivoting or ditching bad ideas early on, we can save time and resources and focus on ideas that will bring value to customers. Lean helps us move more quickly and "execute on opportunities by building the right thing at the right time and stop wasting people's time on ideas that are not valuable."[5]

As Henrik Kniberg (Spotify) puts it, "product development isn't easy. In fact, most product development efforts fail, and the most common reason for failure is building the wrong product."[6] The goal here is not to avoid failure, but to fail more quickly and safely by creating fast feedback loops for building empathy and for learning, so that we can iterate towards success. Let's look at some practices that can help us achieve that.

Prototype testing. Prototypes help us test our ideas with users in a cost-effective way and allow us to validate – or invalidate – our assumptions and hypotheses. They can

5 Jez Humble et al., Lean Enterprise (O'Reilly).

6 Jez Humble et al., Lean Enterprise (O'Reilly).

be as simple as "hand-sketched" drawings of an interface that users would interact with, or they can be clickable interactive mockups. In some cases, we may even opt for "Wizard of Oz" prototypes, which is a real working product, but with all product functions carried out manually behind the scenes, unbeknownst to the person using the product.[7] (It's important to note that "Wizard of Oz" is for prototype testing, not for running production systems. This misapplication, which came to be termed as "artificial artificial intelligence", involves unscalable human effort to solve problems that AI can't solve.)

Whichever you pick, prototype testing is especially useful in ML product delivery because we can get feedback from users before any costly investments in data, ML and MLOps. Prototype testing helps us shorten our feedback loop from weeks or months (time spent on engineering effort in data, ML and MLOps) to days. Talk about fast feedback!

Discovery. Discovery provides a structure for navigating uncertainty, and consists of a rapid, time-boxed, iterative set of activities involving various stakeholders and customers in order to create a clear vision, a shared understanding of the problem, and options for a path forward. As eloquently articulated in Lean Enterprise, the process of creating a shared vision always starts with *clearly defining the problem* because having a clear problem statement helps the team focus on what is important and ignore distractions. By building a shared understanding of our goals and what we hope to achieve, we can improve our ability to come up with better solutions.

Discovery makes extensive use of visual artifacts to canvas, externalize, debate, test and evolve ideas. Some useful visual ideation canvases include the Lean Canvas and Value Proposition Canvas. During discovery, we intentionally put the business at the center and to create ample space for the voice of the customer – gathered through activities such as user journey mapping, contextual enquiry, customer interviews, among others – as we formulate and test hypotheses about the problem/solution fit and product/market fit of our ideas.

Lean Enterprise has an excellent chapter on Discovery, and we would encourage you to read it for an in-depth understanding of how you can structure and facilitate Discovery workshops in your organization. Discovery is also not a one-and-done activity – the principles and practices can be practiced continuously as we build, measure, and learn our way towards building products that customers value.

7 Jeremy Jordan has written an excellent in-depth article describing how we can prototype and iterate on the user experience using design tools to communicate possible solutions.

Delivery

The delivery discipline is primarily focused on the shaping, sizing and sequencing of work in three horizons: from near to far (user stories or features, iterations, and releases). It also pertains to how our teams operate and encompasses team shapes, ways of working (e.g. stand-ups and retrospectives), team health (e.g. psychological safety, and morale), and delivery risk management.

If the product discipline is concerned with what we build and why, the delivery discipline speaks to *how* we execute our ideas. The *how* is further broken down into data, engineering and machine learning, and *delivery* here refers to the non-technical aspects of delivery.

Lean recognizes that talent is an organization's most valuable asset, and the delivery discipline reinforces that belief by creating structures that minimize impediments in our systems of work, and amplify each teammate's contributions and collective ownership. When done right, delivery practices can help us reduce waste and improve the flow of value.

Delivery is an often overlooked but highly critical aspect of building ML products. If we get all the other disciplines right but neglect *delivery*, we will likely be unable to deliver our ML product to users in a timely and reliable manner (we will explain why in a moment). This can lead to decreased customer satisfaction, eroded competitiveness, missed opportunities, and ultimately, failure to achieve the desired business outcomes.

Let's take a look at some fundamental delivery practices.

Vertically sliced work. A common pitfall in ML delivery is the horizontal slicing of work, where we sequentially deliver functional layers of a technical solution (e.g. data lake, ML platform, ML models, UX interfaces) from the bottom-up. The downside of this approach is that users can only experience the product and provide valuable feedback after months of significant engineering investment. In addition, horizontal slicing naturally leads to late integration issues when horizontal slices come together, which increases the risk of release delays.

To mitigate this, we can slice work and stories vertically. A vertically sliced story refers to a story that is defined as an independently shippable unit of value, which contains all of the necessary functionality from the user-facing aspects (e.g. a front-end) to the more back-endish aspects (e.g. data pipelines, ML models). Your definition of "user-facing" will differ depending on who your users are. For example, if you are a platform team delivering an ML platform product for data scientists, the user-facing component may be a command-line utility instead of a frontend application.

The principle of vertical slicing can apply more broadly beyond individual features as well. This is what vertical slicing looks like, in three horizons:

- At the level of *stories*, we articulate and demonstrate business value in each story.
- At the level of *iterations*, we regularly demonstrate value to users by delivering a collection of vertically sliced stories within a reasonable timeframe
- At the level of *releases*, we plan, sequence and prioritize a collection of stories that is focused on creating demonstrable business value

Vertically sliced teams, or cross functional teams. Another common pitfall in ML delivery is splitting teams by function, for example by having data science, data engineering, product engineering in separate teams. This structure leads to two main problems. First, teams inevitably get caught in backlog coupling, which is the scenario where one team needs to depend on (i.e. be blocked by) another team in order to deliver a feature. In one informal analysis, backlog coupling increased the time to complete a task by an average of 10 to 12 times.

The second problem is the manifestation of Conway's Law, which is the phenomenon where teams design systems and software that mirror their communication structure. For example, if data science, data engineering and product engineering were three distinct teams, each of them could very likely implement a different way to solve a problem (e.g. how to persist a model's predictions), just because the path of least resistance steers us toward finding local optimizations rather than coordinating shared functionality.

A better practice would be to identify the capabilities that naturally cohere for a given product, and build a cross-functional team around the product – from the front-facing elements (e.g. experience design, UI design) to back-end elements (e.g. ML, MLOps, data engineering, etc.). This is known as the Inverse Conway Maneuver. This brings three major benefits:

Improving information flow
: The shared context and cadence reduces the friction of discussing and iterating on all things, e.g. design decisions, prioritization calls, assumptions to validate, etc. Instead of having to coordinate a meeting between multiple teams, we can just discuss an issue during or after the team stand-up.

Reducing back-and-forth handoffs and waiting
: If the slicing is done right, the cross-functional team should be autonomous – that means the team is empowered to design and deliver features without depending on or waiting on another team.

Reducing blindspots through diversity

> Having a diverse team with different capabilities and perspectives can help ensure that the machine learning project is well-rounded and takes into account all of the relevant considerations. For example, an UX designer could create prototypes to test and fine-tune ideas with customers before we invest significant engineering effort in machine learning.

To prevent cross-functional squads from becoming yet another source of silos, it also helps to set up cross-team interaction points (e.g. a community of practice) as a way to cross-pollinate learnings, enhance organization-wide cultural alignment, and to again mitigate Conway's Law.

With that said, we'd like to note that there is no one-size-fits-all team shape and the right team shape for your organization depends on many factors. Interaction modes and team shapes will also change over time as products and teams evolve.

Ways of working. Ways of working (WoW) refer to the processes, practices, and tools that a team uses to deliver product features. It includes, but is not limited to, agile ceremonies (e.g. stand-ups, retros, feedback), user story workflow (e.g. kanban, story kickoffs, pair programming, desk checks[8]), quality assurance (e.g. automated testing, manual testing, "stopping the line" when defects occur), among others.

One common trap that teams fall into is to follow the form but miss out on the substance or intent of these ways of working. For example, stand-up updates can sometimes be so generic ("I worked on X yesterday and will continue working on it today") that they contain no useful information. Instead, each of these WoW practices should help the team have context-rich information, e.g. "I'm getting stuck in Y" and "oh I've faced that before, and I know a way to help you". This improves shared understanding, creates alignment, and provides each team member with information that helps to improve their flow.

Measuring delivery metrics. One often-overlooked practice – even in agile teams – is capturing delivery metrics (e.g. iteration velocity, cycle time, defect rates, among others) over time. If we think of the team as a production line (producing creative solutions, and not cookiecutter widgets), these metrics can help us to regularly monitor delivery health and raise flags when we're veering off track from the delivery plan or timelines.

---

8 A Desk Check refers to the practice of having a short (e.g. 15-minute) huddle with the team when a pair believes the development work for a feature is complete. Not everyone has to be there, but it helps to have the product, engineering and quality lens at the desk check.
We find that having a brief walk-through of the definition of done, and how the pair delivered the feature can invite a focused and open discussion, and saves team members from multiple instances of context-switching and waiting in a long-drawn back-and-forth conversation on a chat group.

The objective nature of these metrics help to ground planning conversations in data and help the team actually "see" (in quantitative estimates) the work ahead and how well they are tracking towards their target. In a safe environment, these metrics would be used purely for continuous improvement to help us improve our "production line" over time and help us meet our product delivery goals. This benefits not only the customer, but also individuals on the team and the business.

Engineering

> Crucially, the rate at which we can learn, update our product or prototype based on feedback, and test again, is a powerful competitive advantage. This is the value proposition of the lean engineering practices.
>
> —Jez Humble, Joanne Molesky, Barry O'Reilly, Lean Enterprise

All of the agile engineering practices that we outline below are oriented towards one thing: *shortening feedback loops*. The quote above from Lean Enterprise articulates it well – an effective team is one that can make the required changes (in code, data, or ML models) and rapidly test and release these changes.

Automated testing. Automated tests provide a solid and essential foundation for building a product that is easy to maintain and evolve. Tests give us *fast feedback* on changes and lets us know in a fast and automated fashion on whether everything is still working as expected.

Automated tests allow us to rapidly respond to the only constant in life: change. To iterate towards a better ML product, effective teams welcome valuable changes in various aspects of our product: new business requirements, feature engineering strategies, modeling approaches, training data, among others. Without automated tests, changes become error-prone, tedious, and stressful. When we change one part of the codebase, the lack of tests forces us to take on the burden of manually testing the entire codebase to ensure that a change (e.g. in feature engineering logic) hasn't caused a degradation (e.g. in model quality, API behavior in edge cases, etc.). This contributes to an overwhelming amount of time, effort, and cognitive load spent on non-ML work.

In contrast, having a set of comprehensive automated tests will help teams to accelerate experimentation, reduce cognitive load and get fast feedback. In practice, it can bring a night-and-day difference in how quickly we can execute on our ideas and get stories properly done.

Refactoring . The second law of thermodynamics tells us that the universe tends towards disorder (a.k.a. entropy), and our codebases – ML or otherwise – are no exception. With every feature delivered and with every "quick hack", the codebase grows increasingly convoluted and brittle. This makes the code increasingly hard to understand and consequently, modifying code becomes painful and error-prone.

ML projects that lack automated tests are especially susceptible to exponential complexity because without automated tests, refactoring can be tedious to test and is highly risky. Consequently, refactoring becomes an effortful undertaking that gets relegated to the backlog graveyard. As a result, we create a vicious cycle for ourselves and it becomes increasingly difficult for ML practitioners to evolve their ML solutions.

In an effective team, refactoring is something that is so safe and easy to do that we can do some of it as part of feature delivery, not as an afterthought. Such teams are able to do this typically for three reasons:

- They have a comprehensive tests that gives them fast feedback on whether a refactoring had preserved behavior (as it should)
- They've configured their code editor and leveraged the ability of modern code editors to execute refactoring actions (e.g. rename variable, extract function, change signature, etc.)
- The amount of tech debt and/or workload is at a healthy level. Instead of feeling crushed by pressure, they have the capacity to refactor where necessary as part of feature delivery to improve the readability and quality of the codebase.

Code editor effectiveness. As alluded to in the previous point, modern code editors have many powerful features that can help code contributors write code more effectively. The code editor can take care of low-level details so that our cognitive capacity remains available for solving higher-level problems.

For example, instead of renaming variables through a manual search and replace, the code editor can rename all references to a variable in one shortcut. Instead of manually searching for the syntax for importing a function (e.g. `cross_val_score()`), we can hit a shortcut and the IDE can automatically import the function for us.

When configured properly, the code editor becomes a powerful assistant (even without AI coding technologies) and can allow us to execute our ideas, solve problems and deliver value more effectively.

Continuous delivery for machine learning (CD4ML). Continuous Delivery for Machine Learning (CD4ML) is the application of Continuous Delivery principles and practices to ML applications. It enables teams to test and release changes (in code, data, and/or models) in small and safe increments that can be reproduced and reliably released at any time, in short adaptation cycles.

Through automation, comprehensive testing, and observability, CD4ML improves quality, reduces toil, accelerates execution, and allows ML practitioners to focus on solving higher-level (and likely more interesting) problems rather than on tedious repetitive tasks.

CD4ML comes with the following technical components (see Figure 1-6):

- Discoverable and accessible data
- Reproducible model training
- Model serving
- Testing and quality
- Experiments tracking
- Model deployment
- Continuous delivery orchestration
- Model monitoring and observability

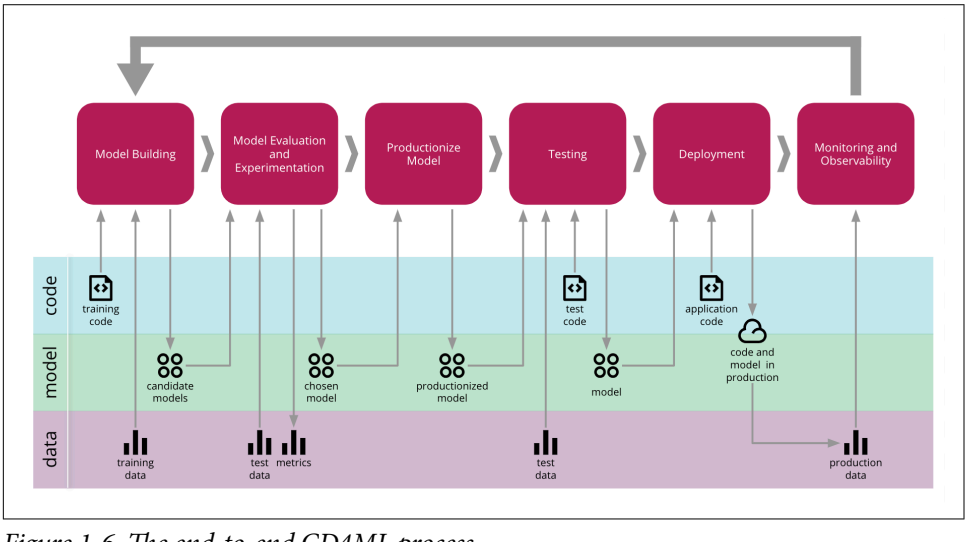

Figure 1-6. The end-to-end CD4ML process.

Machine learning

The ML discipline feels like it needs no introduction for the intended audience of this book, but for completeness, we will do so anyway. The ML discipline involves more than just knowing how to train, improve and consume ML models. It also encompasses practices such as: ML problem framing, ML system design, designing for explainability, responsible AI, and ML governance, among others.

Framing ML problems. In early and exploratory phases of ML projects, it's usually unclear what problem we should be solving, who we are solving it for, and most importantly, why we should solve it. In addition, it may not be clear what ML paradigm or model architectures can help us – or even what data we have or need – to solve the problem with ML. That is why it is important to have the capability

to frame ML problems, to structure and execute ideas, and to validate hypotheses with the relevant customers or stakeholders. The saying, "a problem well-defined is a problem half-solved," resonates well in this context.

There are various tools that can help us frame ML problems in a way that is executable and testable. One such tool is the ML Canvas, which provides a framework for connecting the dots between data collection, ML, and value creation. Another tool to help us systematically articulate, test our ideas in short cycles, and keep track of learnings over time is hypothesis-driven development (see Figure 1-7). Hypothesis-driven development helps us to formulate testable hypotheses and steers us towards measuring objective metrics to validate or invalidate ideas. It is yet another way to shorten feedback loops by running targeted, time-boxed experiments.

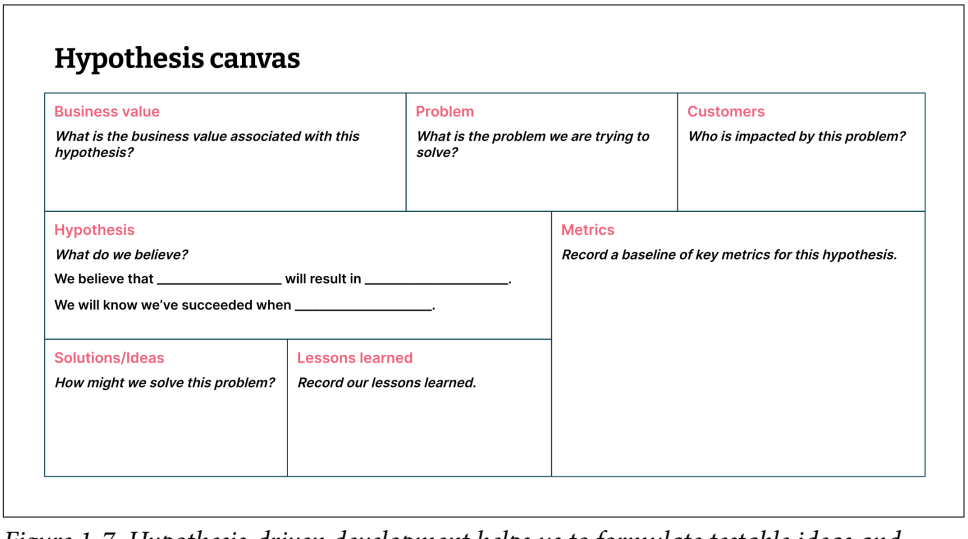

Figure 1-7. Hypothesis-driven development helps us to formulate testable ideas and steers us towards collective objective metrics to validate or invalidate ideas. It is yet another way to shorten feedback loops by running targeted, time-boxed experiments.

Note: the word "hypothesis" in this context is different, albeit similar, to how it's defined in statistics. In this context, a hypothesis is a testable assumption, and it is used as a starting point for iterative experimentation and testing to determine the most effective solution to the problem.

ML systems design. There are many parts to designing ML systems, such as collecting and processing the data needed by the model, selecting the appropriate ML approach and tools, evaluating the performance of the model, designing for explainability, considering access patterns and scalability requirements, understanding ML failure modes, identifying model-centric and data-centric strategies for iteratively improving the model, among others.

There is a great book that has been written on this topic, Designing ML Systems, and we encourage you to read it if you haven't already done so. Given that there's great literature on this topic, our book will be light on details on ML systems design.

Responsible AI. ML practitioners are often asked (or asking themselves) if a model is good enough, but it's equally important to ask and answer: good enough *for who*? Effective ML teams are able to instrument mechanisms for understanding not just why, but also *where*, our models are under-performing. They are also able to consider data-centric approaches for reducing model bias, such as by designing data collection loops and scalable labeling mechanisms to create more representative datasets.

We often also need to be able to understand and explain why a model behaved in a certain way under a certain scenario or data distribution. Explainability is useful not just to users and stakeholders, it is also an essential capability for ML practitioners to effectively carry out error analysis, which is a precursor to remedial strategies for improving the model. It may also be a requirement in certain industries from a governance and regulatory perspective.

ML governance . ML governance refers to the processes, policies, and practices that are put in place to ensure the responsible and ethical development, deployment, and use of ML models. ML governance involves a wide range of activities, including setting standards and guidelines for ML development, establishing procedures for model selection and evaluation, defining roles and responsibilities for ML stakeholders, and implementing mechanisms for monitoring and enforcing compliance with industry-level or organization-level regulations.

ML governance also involves addressing issues related to fairness, transparency, and accountability in ML models. This may involve implementing techniques to mitigate biases in data and models, creating mechanisms for explaining model decisions, and establishing processes for addressing complaints and grievances related to ML outcomes.

While "governance" typically has bureaucratic connotations, we'll demonstrate in this book that ML governance can be implemented in a lean and lightweight fashion.

Data

The availability of high-quality labeled data can make or break ML projects. The author of the paper, "The Unreasonable Effectiveness of Data", describes how ML algorithms – even if they are simple – can often achieve impressive results simply by being fed large amounts of data and allowed to learn from it. In order for an ML model to be accurate, it must be trained on a dataset that is representative of the problem it is trying to solve.

The quality of our ML models depends on the quality of our data. If our data in our training sample is biased (as compared to the distribution of the population dataset), then the model will learn and perpetuate the bias. As eloquently put, "when today's technology relies on yesterday's data, it will simply mirror our past mistakes and biases."[9] It will also likely make incorrect predictions for out-of-sample data points. This can lead to dangerous runaway feedback loops, where the model's biased predictions have an effect on the real world, which then further entrenches the bias in the data and subsequent models.

To deliver better ML solutions, teams can consider the following practices in the discipline of data.

Closing the data collection loop. As we train and deploy models, our ML system design should also take into consideration how we will collect and curate the model's predictions in production, so that we can label them and grow high-quality ground truth for evaluating and retraining models.

Labeling can be a tedious activity and is often the bottleneck. If so, we can also consider how to scale labeling through techniques such as active learning or weak supervision. If natural labels are available for our ML task, we should also design software and data ingestion pipelines that stream in the natural labels as they become available along with the associated features for the given sample. When collecting natural labels, we must also consider how to mitigate the risks of data poisoning attacks (more on this shortly).

Reducing data distribution shifts. The distance between training data and inference data is known as a data distribution shift, and is a common cause of ML system failure. Data distribution shifts could result from covariate shift, label shift, or concept drift and cause a model that was performing well (when evaluated using an in-sample validation dataset) to underperform in production when presented with non-stationary out-of-sample data.

This is comprehensively discussed in Designing Machine Learning Systems, so we won't go into detail on how the shifts happen and when to trigger a continual learning event to retrain a model. However, in this book, we will highlight ways to reduce the distribution shift between training and inference data (see Figure 1-8):

1. Ensure the distribution shift between training data and inference data is as small as possible, by: (i) ensure all feature engineering logic are symmetrically applied in both scenarios, and (ii) regularly refreshing the data that the model was trained on.

9 Patrick K. Lin, *Machine See, Machine Do: How Technology Mirrors Bias in Our Criminal Justice System* (New Degree Press, 2021)

2. Create data collection loops and data labeling mechanisms to ensure that our training data is regularly refreshed and is as production-like as possible.
3. In many situations, especially when the data contains personally identifiable information (PII), we cannot and should not have access to production data for training and testing in a non-production environment. In such scenarios, we can generate production-like synthetic data to ensure that we have high-quality production-like data for testing ML models.

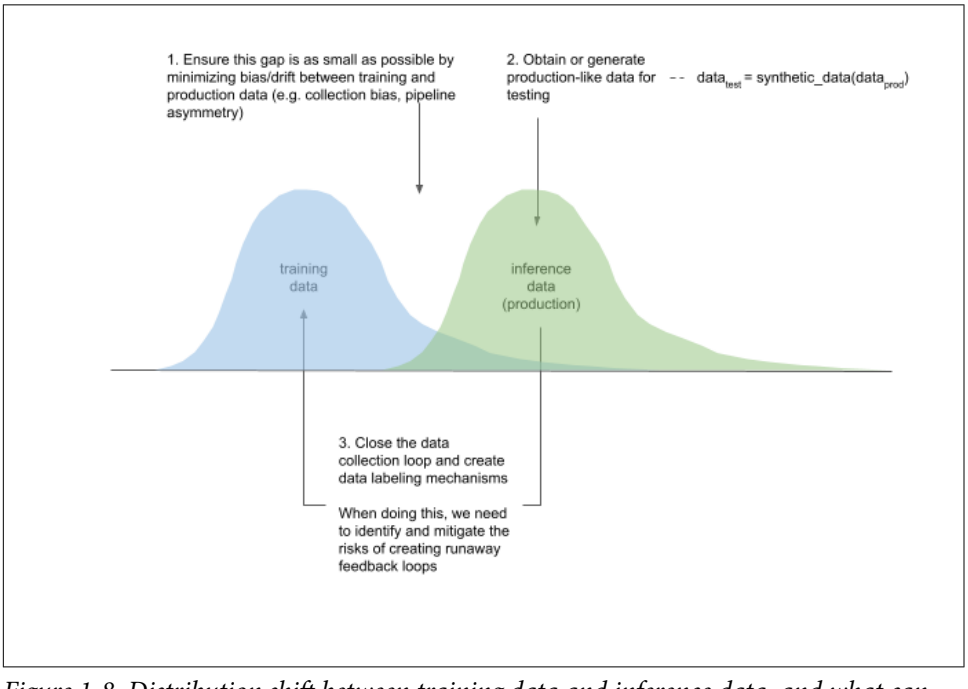

Figure 1-8. Distribution shift between training data and inference data, and what can we do to minimize the drift

Data security and privacy . As mentioned earlier in this chapter, data security and privacy is a cross-cutting concern that should be the responsibility of everyone in the organization, from product teams to data engineering teams and every team in between. The organization can ensure data security and privacy in several ways, such as by storing data securely in transit and at rest through the use of encryption and access controls. Teams should apply the principle of least privilege and ensure that only authorized individuals and systems can access the data.

At an organizational level, there must be data governance and management guidelines that define and enforce clear policies to guide how teams collect, store, and use data. This can help ensure that data is used ethically and in compliance with relevant

laws and regulations. This is a very big topic, and we will dive into details in a later chapter.

Give yourself some massive pats on the back because you've just covered a lot of ground on the interconnected disciplines that are essential for effectively delivering ML solutions!

Before we conclude this chapter, we'd like to highlight how these practices can serve as *leading indicators* for positive or undesirable outcomes. For example, if we don't validate our product ideas with users early and often – we know how this movie ends – we are more likely to invest lots of time and effort into building the wrong product. If we don't have cross-functional teams, we are going to experience backlog coupling as multiple teams coordinate and wait on each other to deliver a change to users.

And this is not just anecdotal. In a scientific study on performance and effectiveness of tech businesses involving over 2800 organizations, the authors found that organizations that adopt practices such as continuous delivery, Lean, cross-functional teams, generative cultures, among others, exhibit higher levels of performance, such as faster delivery of features, lower failure rates, and higher levels of employee satisfaction.[10] In other words, these practices can actually be *predictors* of an organization's performance.

Conclusion

Let's recap what we've covered in this chapter. We started by looking at common reasons as to why ML projects fail, and we compared what ML delivery looks in a low-effectiveness and high-effectiveness environment. We then put on a systems thinking lens to identify subsystems or disciplines that are required for effective ML delivery, and looked at how Lean for principles and practices can help us reduce waste and maximize value. Finally, we took a whirlwind tour of practices in each of the five disciplines (product, delivery, engineering, data, and machine learning) which can help us deliver ML solutions more effectively.

From our interactions with various ML or data science teams across multiple industries, we continue to see a gap between the world of ML and the world of Lean software delivery. While that gap has narrowed in certain pockets – where ML teams could deliver excellent ML product experiences by adopting the necessary product, delivery, and engineering practices – the gulf remains wide for many teams (you can look at the "low-effectiveness environment" story earlier this chapter for signs of this gulf).

10 Forsgren, Nicole, Jez Humble, and Gene Kim. Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations. Upper Saddle River, NJ: Addison-Wesley, 2018.

To close this gap, the ML community requires a paradigm shift – a fundamental change in approach or underlying assumptions – to see that building an ML-driven product is not just an ML and data problem. It is also a *product* problem, which means to say it's a product, engineering, and delivery problem – and therefore it requires a holistic multi-disciplinary approach.

The good news is that you don't have to boil the ocean or reinvent the wheel – in each discipline, there are principles and practices that have helped teams successfully deliver ML product experiences. In the remainder of this book, we will explore these principles and practices, and how they can improve our effectiveness in delivering ML solutions.

An Invitation to Journey with Us

We have covered a lot of ground in this chapter. Depending on where you are on your journey and your experience, you may feel like the desired state that we've painted is insurmountable. Or you may feel excited that others have felt your pains and challenges, and that there's a better path.

Wherever you find yourself on this continuum, we hope that you'll take this book as an invitation. An invitation to adopt a beginner's mindset with us – to see that ML and other disciplines (e.g. engineering, delivery) are not binary choices. Rather, they are composable techniques that teams can use today to better leverage your existing capabilities – be it in ML, data or software engineering.

It's also an invitation to reflect on your team's or organization's ML projects, to notice areas of value and areas of waste. Where there is waste, we hope that the principles and practices we lay out in this book will help you find shorter and more reliable paths to your desired destination. They have certainly helped us (which is why we decided to write a book about this!) and they are principles and practices that we continue to bring to our ML projects.

We acknowledge that it takes more than willpower and good practices to effect change. It requires some level of organizational alignment, a conducive culture, staff enablement, among other factors (we will elaborate on building blocks of change in a later chapter). This book is written with the belief that teams, empowered with practical knowledge on how to effectively deliver ML solutions, can iterate towards better ways of doing things, deliver impactful outcomes, and effect and inspire change in their organization.

In the remaining chapters, we will slow down and elaborate on the principles and practices in a practical way. There will be applicable practices, frameworks and code samples that you can bring to your ML projects. We hope you're strapped in and excited for the ride.

In the next chapter, we will kick off Part I: Engineering Effectiveness with effective dependency management – principles and practices that can help you avoid dependency hell and create reproducible and production-like environments for running your code.