# CHAPTER 1
# Introduction to MLOps

*By Noah Gift*

> Since 1986, I have had a few more deaths, several from insufficient attention but mainly from deliberately pushing the limits in various directions—taking a chance in bonsai is a bit like taking a chance with love; the best outcome requires risky exposure to being hurt and no guarantee of success.
>
> —Dr. Joseph Bogen

One of the powerful aspects of science fiction is its ability to imagine a future without constraints. One of the most influential science fiction shows of all time is the TV show *Star Trek*, which first aired in the mid-1960s, approximately 60 years ago. The cultural impact inspired designers of technology like the Palm Pilot and hand-held cellular phones. In addition, *Star Trek* influenced the cofounder of Apple computer, Steve Wozniak, to create Apple computers.

In this age of innovation in machine learning, there are many essential ideas from the original series relevant to the coming MLOps (or Machine Learning Operations) industrial revolution. For example, *Star Trek* hand-held tricorders can instantly classify objects using pretrained multiclass classification models. But, ultimately, in this futuristic science-fiction world, domain experts like the science officers, medical officers, or captain of the ship, don't spend months training machine learning models. Likewise, the crew of their science vessel, the *Enterprise*, are not called data scientists. Instead, they have jobs where they often use data science.

Many of the machine learning aspects of this *Star Trek* science fiction future are no longer science fiction in the 2020s. This chapter guides the reader into the foundational theory of how to make this possible. Let's get started.

## Rise of the Machine Learning Engineer and MLOps

Machine learning (ML), with its widespread adoption globally, has created a need for a systematic and efficient approach toward building ML systems, leading to a sudden rise in demand for ML engineers. These ML engineers, in turn, are applying established DevOps best practices to the emerging machine learning technologies. The major cloud vendors all have certifications targeting these practitioners. I have experience working directly with AWS, Azure, and GCP as a subject matter expert on machine learning. In some cases, this includes helping create machine learning certifications themselves and official training material. In addition, I teach machine learning engineering and cloud computing at some of the top data science programs with Duke and Northwestern. Firsthand, I have seen the rise in machine learning engineering as many former students have become machine learning engineers.

Google has a Professional Machine Learning Engineer certification. It describes an ML engineer as someone who "designs, builds, and productionizes ML models to solve business challenges…" Azure has a Microsoft Certified: Azure Data Scientist Associate. It describes this type of practitioner as someone who "applies their knowledge of data science and machine learning to implement and run machine learning workloads…" Finally, AWS describes an AWS Certified Machine Learning specialist as someone with "the ability to design, implement, deploy, and maintain machine learning solutions for given business problems."

One way to look at data science versus machine learning engineering is to consider science versus engineering itself. Science gears toward research, and engineering gears toward production. As machine learning moves beyond just the research side, companies are eager for a return on investment in hiring around AI and ML. According to payscale.com and glassdoor.com, the results show that at the end of 2020, the median salary for a data scientist, data engineer, and machine learning engineer was similar. According to LinkedIn in Q4 of 2020, 191K jobs mentioned cloud, there were 70K data engineering job listings, 55k machine learning engineering job listings, and 20k data science job listings, as shown in Figure 1-1.

Another way to look at these job trends is that they are a natural part of the hype cycle of technology. Organizations realize that to generate a return on investment (ROI), they need employees with hard skills: cloud computing, data engineering, and machine learning. They also need them in a much larger quantity than data scientists. As a result, the decade of the 2020s may show an acceleration in treating data science as a behavior, rather than a job title. DevOps is behavior, just like data science. Think about the principles of both DevOps and data science. In both cases, DevOps and data science are methodologies for evaluating the world, not necessarily unique job titles.

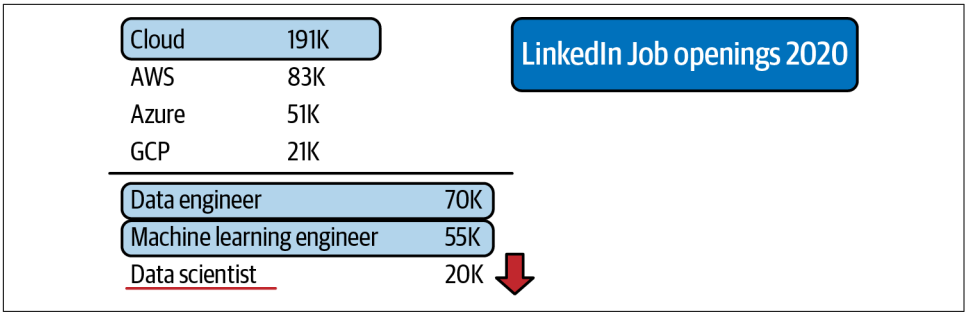


*Figure 1-1. Machine learning jobs*

Let's look at the options for measuring the success of a machine learning engineering initiative at an organization. First, you could count the machine learning models that go into production. Second, you could measure the impact of the ML models on business ROI. These metrics culminate in a model's operational efficiency. The cost, uptime, and staff needed to maintain it are signals that predict a machine learning engineering project's success or failure.

Advanced technology organizations know they need to leverage methodologies and tools that decrease the risk of machine learning projects failing. So what are these tools and processes used in machine learning engineering? Here is a partial list:

*Cloud native ML platforms*
: AWS SageMaker, Azure ML Studio, and GCP AI Platform

*Containerized workflows*
: Docker format containers, Kubernetes, and private and public container registries

*Serverless technology*
: AWS Lambda, AWS Athena, Google Cloud Functions, Azure Functions

*Specialized hardware for machine learning*
: GPUs, Google TPU (TensorFlow Processing Unit), Apple A14, AWS Inferentia Elastic inference

*Big data platforms and tools*
: Databricks, Hadoop/Spark, Snowflake, Amazon EMR (Elastic Map Reduce), Google Big Query

One clear pattern about machine learning is how deeply tied it is to cloud computing. This is because the raw ingredients of machine learning happen to require massive compute, extensive data, and specialized hardware. Thus, there is a natural synergy with deep integration with cloud platforms and machine learning engineering. Further supporting this is that the cloud platforms are building specialized platforms to

enhance ML's operationalization. So, if you are doing ML engineering, you are probably doing it in the cloud. Next, let's discuss how DevOps plays a role in doing this.

## What Is MLOps?

Why isn't machine learning 10X faster? Most of the problem-building machine learning systems involve everything surrounding machine learning modeling: data engineering, data processing, problem feasibility, and business alignment. One issue with this is a focus on the "code" and technical details versus solving the business problem with machine learning. There is also a lack of automation and the issue of HiPPO (Highest Paid Person's Opinions) culture. Finally, much of machine learning is not cloud native and uses academic datasets and academic software packages that don't scale for large-scale problems.

The quicker the feedback loop (see Kaizen), the more time to focus on business problems like the recent issues of rapid detection of Covid, detecting mask versus nonmask computer vision solutions deployed in the real world, and faster drug discovery. The technology exists to solve these problems, yet these solutions aren't available in the real world? Why is this?

What is Kaizen? In Japanese, it means improvement. Using Kaizen as a software management philosophy originated in the Japanese automobile industry after World War II. It underlies many other techniques: Kanban, root cause analysis & five why's, and Six Sigma. To practice Kaizen, an accurate and realistic assessment of the world's state is necessary and pursues daily, incremental improvements in the pursuit of excellence.

The reason models are not moving into production is the impetus for the emergence of MLOps as a critical industry standard. MLOps shares a lineage with DevOps in that DevOps philosophically demands automation. A common expression is *if it is not automated, it's broken*. Similarly, with MLOps, there must not be components of the system that have humans as the levers of the machine. The history of automation shows that humans are the least valuable doing repetitive tasks but are the most valuable using technology as the architects and practitioners. Likewise, coordination between developers, models, and operations must coordinate through transparent teamwork and healthy collaboration. Think of MLOps as the process of automating machine learning using DevOps methodologies.

What is DevOps? It combines best practices, including microservices, continuous integration, and continuous delivery, removing the barriers between Operations and Development and Teamwork. You can read more about DevOps in our book *Python for DevOps* (O'Reilly). Python is the predominant language of scripting, DevOps, and machine learning. As a result of this, this MLOps book focuses on Python, just as the DevOps book focused on Python.

With MLOps, not only do the software engineering processes need full automation, but so do the data and modeling. The model training and deployment is a new wrinkle added to the traditional DevOps lifecycle. Finally, additional monitoring and instrumentation must account for new things that can break, like data drift—the delta between changes in the data from the last time the model training occurred.

A fundamental problem in getting machine learning models into production is the immaturity of the data science industry. The software industry has embraced DevOps to solve similar issues; now, the machine learning community embraces MLOps. Let's dive into how to do this.

# DevOps and MLOps

DevOps is a set of technical and management practices that aim to increase an organization's velocity in releasing high-quality software. Some of the benefits of DevOps include speed, reliability, scale, and security. These benefits occur through adherence to the following best practices:

*Continuous integration (CI)*
: CI is the process of continuously testing a software project and improving the quality based on these tests' results. It is automated testing using open source and SaaS build servers such as GitHub Actions, Jenkins, Gitlab, CircleCI, or cloud native build systems like AWS Code Build.

*Continuous delivery (CD)*
: This method delivers code to a new environment without human intervention. CD is the process of deploying code automatically, often through the use of IaC.

*Microservices*
: A microservice is a software service with a distinct function that had little to no dependencies. One of the most popular Python-based microservice frameworks is Flask. For example, a machine learning prediction endpoint is an excellent fit for a microservice. These microservices can use a wide variety of technologies, including FaaS (function as a service). A perfect example of a cloud function is AWS Lambda. A microservice could be container-ready and use CaaS (container

as a service) to deploy a Flask application with a Dockerfile to a service like AWS Fargate, Google Cloud Run, or Azure App Services.

*Infrastructure as Code*
: Infrastructure as Code (IaC) is the process of checking the infrastructure into a source code repository and "deploying" it to push changes to that repository. IaC allows for idempotent behavior and ensures the infrastructure doesn't require humans to build it out. A cloud environment defined purely in code and checked into a source control repository is a good example use case. Popular technologies include cloud-specific IaC like AWS Cloud Formation or AWS SAM (Serverless Application Model). Multicloud options include Pulumi and Terraform.

*Monitoring and instrumentation*
: Monitoring and instrumentation are the processes and techniques used that allow an organization to make decisions about a software system's performance and reliability. Through logging and other tools like application performance monitoring tools such as New Relic, Data Dog, or Stackdriver, monitoring and instrumentation are essentially collecting data about the behavior of an application in production or data science for deployed software systems. This process is where Kaizen comes into play; the data-driven organization uses this instrumentation to make things better daily or weekly.

*Effective technical communication*
: This skill involves the ability to create effective, repeatable, and efficient communication methods. An excellent example of effective technical communication could be adopting AutoML for the initial prototyping of a system. Of course, ultimately, the AutoML model may be kept or discarded. Nevertheless, automation can serve as an informational tool to prevent work on an intractable problem.

*Effective technical project management*
: This process can efficiently use human and technology solutions, like ticket systems and spreadsheets, to manage projects. Also, appropriate technical project management requires breaking down problems into small, discreet chunks of work, so incremental progress occurs. An antipattern in machine learning is often when a team works on one production machine model that solves a problem "perfectly." Instead, smaller wins delivered daily or weekly is a more scalable and prudent approach to model building.

Continuous integration and continuous delivery are two of the most critical pillars of DevOps. Continuous integration involves merging code into a source control repository that automatically checks the code's quality through testing. Continuous delivery is when code changes are automatically tested and deployed, either to a staging environment or production. Both of these techniques are a form of automation in the spirit of Kaizen or continuous improvement.

A good question to ask is who on the team should implement CI/CD? This question would be similar to asking who contributes to taxes in a democracy. In a democracy, taxes pay for roads, bridges, law enforcement, emergency services, schools, and other infrastructure, so all must contribute to building a better society. Likewise, all MLOps team members should help develop and maintain the CI/CD system. A well-maintained CI/CD system is a form of investment in the future of the team and company.

An ML system is also a software system, but it contains a unique component: a machine learning model. The same benefits of DevOps can and do apply to ML systems. The embrace of automation is why new approaches like Data Versioning and AutoML hold many promises in capturing the DevOps mindset.

## An MLOps Hierarchy of Needs

One way to think about machine learning systems is to consider Maslow's hierarchy of needs, as shown in Figure 1-2. Lower levels of a pyramid reflect "survival," and true human potential occurs after basic survival and emotional needs are met.

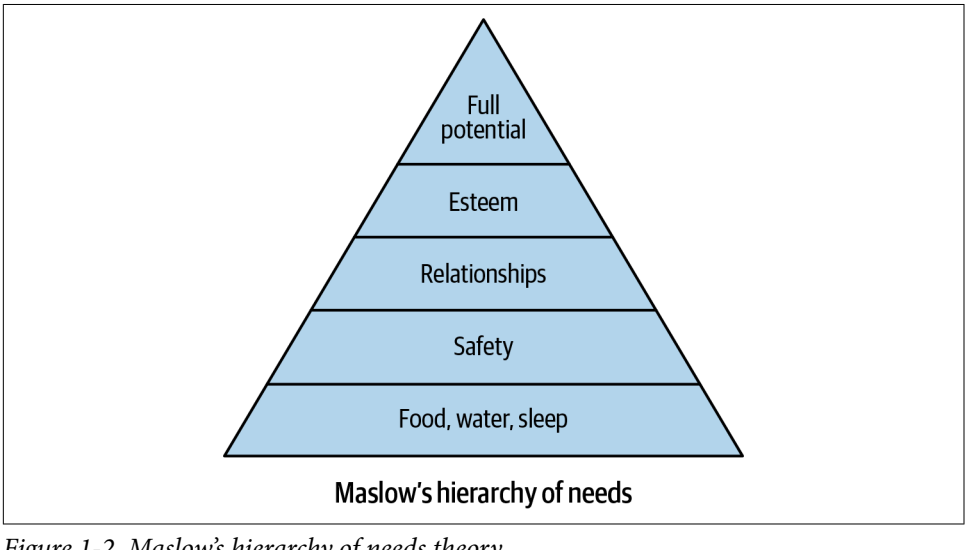


*Figure 1-2. Maslow's hierarchy of needs theory*

The same concept applies to machine learning. An ML system is a software system, and software systems work efficiently and reliably when DevOps and data engineering best practices are in place. So how could it be possible to deliver the true potential of machine learning to an organization if DevOps' basic foundational rules don't exist or data engineering is not fully automated? The ML hierarchy of needs shown in Figure 1-3 is not a definitive guide but is an excellent place to start a discussion.

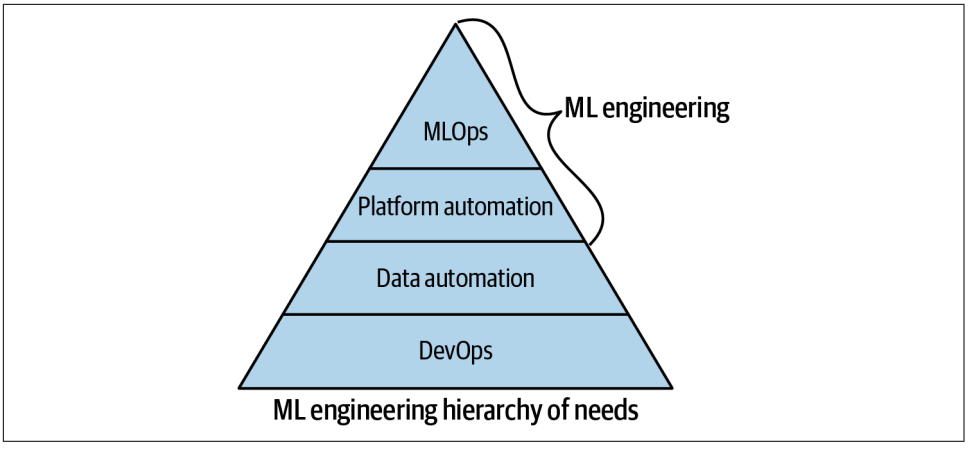


*Figure 1-3. ML engineering hierarchy of needs*

One of the major things holding back machine learning projects is this necessary foundation of DevOps. After this foundation is complete, next is data automation, then platform automation, and then finally true ML automation, or MLOps, occurs. The culmination of MLOps is a machine learning system that works. The people that work on operationalizing and building machine learning applications are machine learning engineers and/or data engineers. Let's dive into each step of the ML hierarchy and make sure you have a firm grasp of implementing them, starting with DevOps.

## Implementing DevOps

The foundation of DevOps is continuous integration. Without automated testing, there is no way to move forward with DevOps. Continuous integration is relatively painless for a Python project with the modern tools available. The first step is to build a "scaffolding" for a Python project, as shown in Figure 1-4.

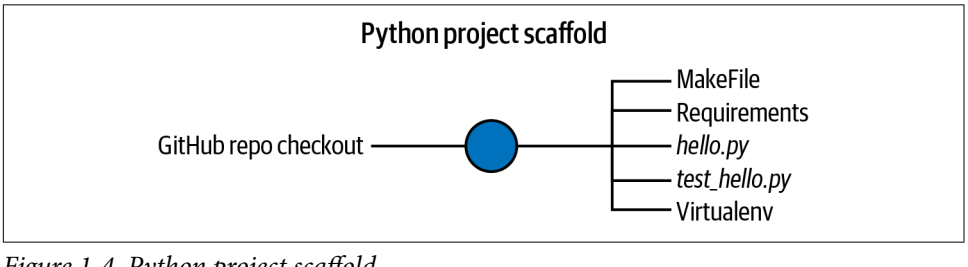


*Figure 1-4. Python project scaffold*

The runtime for a Python machine learning project is almost guaranteed to be on a Linux operating system. As a result, the following Python project structure is straightforward to implement for ML projects. You can access the source code for this example on GitHub for reference as you read this section. The components are as follows:

*Makefile*
: A *Makefile* runs "recipes" via the `make` system, which comes with Unix-based operating systems. Therefore, a Makefile is an ideal choice to simplify the steps involved in continuous integration, such as the following. Note that a Makefile is a good starting point for a project and will often evolve as new pieces need automation.

If your project uses a Python virtual environment, you source it before you work with a Makefile, since all a Makefile does is run commands. It is a common mistake for a newcomer to Python to confuse Makefile with a virtual environment. Similarly, suppose you use an editor like Microsoft Visual Studio Code. In that case, you will need to tell the editor about your Python virtual environment so it can accurately give you syntax highlighting, linting, and other available libraries.

*Make install*
: This step installs software via the `make install` command

*Make lint*
: This step checks for syntax errors via the `make lint` command

*Make test*
: This step runs tests via the `make test` command:

```
install:
        pip install --upgrade pip &&\
                pip install -r requirements.txt
lint:
        pylint --disable=R,C hello.py

test:
        python -m pytest -vv --cov=hello test_hello.py
```

### Why a Makefile?

A common reaction to hearing about a Makefile from an absolute beginner to Python is "Why do I need this?" Generally, it is healthy to have skepticism about things that appear to add work. In the case of a Makefile, though, they are actually less work because they keep track of complicated build steps that are very difficult to remember and type out correctly.

A great example is a lint step with the `pylint` tool. With a Makefile, you only need to run `make lint`, and the same command can run inside a continuous integration server. The alternative approach is to type out the complete directive each time you need it, such as the following:

```
pylint --disable=R,C *.py
```

This sequence is very prone to errors and quite tedious to repeatedly type over your project's life. Instead, it is much simpler to type the following:

```
make lint
```

When you embrace the Makefile approach, it simplifies your workflow and makes it easier to integrate your project into a continuous integration system. There is less code to type, and this is always a good thing for automation. Further, Makefile commands are recognized by shell auto-completion, making it easy to "tab-complete" the steps.

*requirements.txt*
: A *requirements.txt* file is a convention used by the `pip` installation tool, the default installation tool for Python. A project can contain one or more of these files if different packages need installation for different environments.

*Source code and tests*
: The Python scaffolding's final portion is to add a source code file and a test file, as shown here. This script exists in a file called *hello.py*:

```
def add(x, y):
    """This is an add function"""

    return x + y


print(add(1, 1))
```

Next, the test file is very trivial to create by using the `pytest` framework. This script would be in a file *test_hello.py* contained in the same folder as *hello.py* so that the `from hello import add` works:

```
from hello import add


def test_add():
    assert 2 == add(1, 1)
```

These four files: *Makefile*, *requirements.txt*, *hello.py*, and *test_hello.py* are all that is needed to start the continuous integration journey except for creating a local Python virtual environment. To do that, first, create it:

```
python3 -m venv ~/.your-repo-name
```

Also, be aware that there are generally two ways to create a virtual environment. First, many Linux distributions will include the command-line tool `virtualenv`, which does the same thing as `python3 -m venv`.

Next, you source it to "activate" it:

```
source ~/.your-repo-name/bin/activate
```

Why create and use a Python virtual environment? This question is ubiquitous for newcomers to Python, and there is a straightforward answer. Because Python is an interpreted language, it can "grab" libraries from anywhere on the operating system. A Python virtual environment isolates third-party packages to a specific directory. There are other solutions to this problem and many developing tools. They effectively solve the same problem: the Python library and interpreter are isolated to a particular project.

Once you have this scaffolding set up, you can do the following local continuous integration steps:

1. Use `make install` to install the libraries for your project.
   The output will look similar to Figure 1-5 (this example shows a run in GitHub Codespaces):

```
$ make install
pip install --upgrade pip &&\
        pip install -r requirements.txt
Collecting pip
  Using cached pip-20.2.4-py2.py3-none-any.whl (1.5 MB)
[.....more output suppressed here......]
```

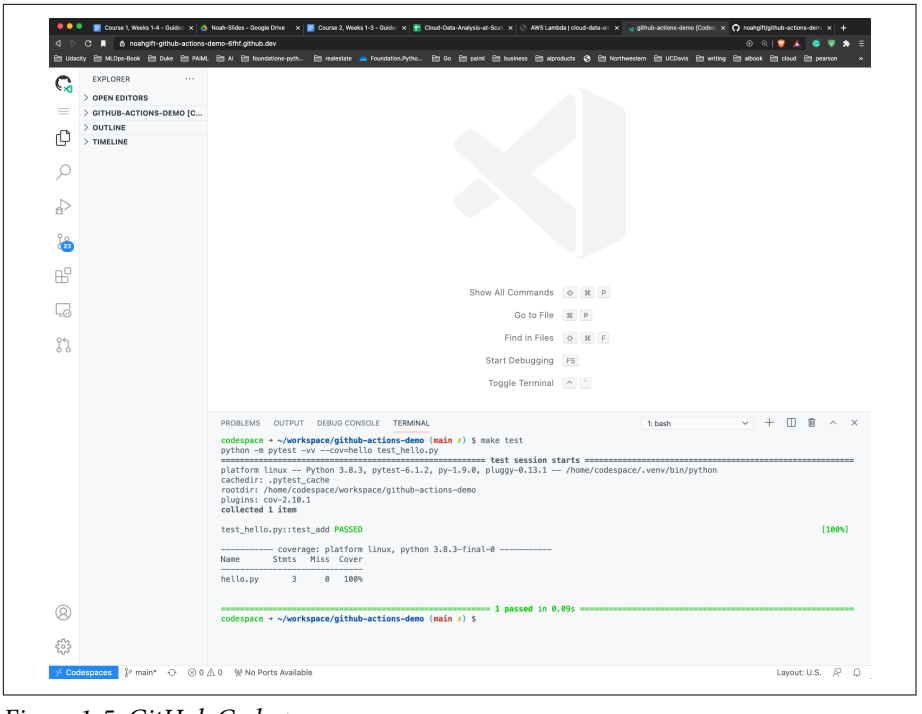


*Figure 1-5. GitHub Codespaces*

2. Run `make lint` to lint your project:

```
$ make lint
pylint --disable=R,C hello.py

-----------------------------------
Your code has been rated at 10.00/10
```

3. Run `make test` to test your project:

```
$ make test
python -m pytest -vv --cov=hello test_hello.py
===== test session starts ====
platform linux -- Python 3.8.3, pytest-6.1.2,\
/home/codespace/.venv/bin/python
cachedir: .pytest_cache
rootdir: /home/codespace/workspace/github-actions-demo
plugins: cov-2.10.1
collected 1 item

test_hello.py::test_add PASSED

---------- coverage: platform linux, python 3.8.3-final-0 ----------
Name        Stmts   Miss  Cover
```

```
------------------------------
hello.py       3      0   100%
```

Once this is working locally, it is straightforward to integrate this same process with a remote SaaS build server. Options include GitHub Actions, a Cloud native build server like AWS Code Build, GCP CloudBuild, Azure DevOps Pipelines, or an open source, self-hosted build server like Jenkins.

## Configuring Continuous Integration with GitHub Actions

One of the most straightforward ways to implement continuous integration for this Python scaffolding project is with GitHub Actions. To do this, you can either select "Actions" in the GitHub UI and create a new one or create a file inside of these directories you make as shown here:

```
.github/workflows/<yourfilename>.yml
```

The GitHub Actions file itself is straightforward to create, and the following is an example of one. Note that the exact version of Python sets to whatever interpretation the project requires. In this example, I want to check a specific version of Python that runs on Azure. The continuous integration steps are trivial to implement due to the hard work earlier in creating a Makefile:

```
name: Azure Python 3.5
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Set up Python 3.5.10
      uses: actions/setup-python@v1
      with:
        python-version: 3.5.10
    - name: Install dependencies
      run: |
        make install
    - name: Lint
      run: |
        make lint
    - name: Test
      run: |
        make test
```

An overview of what GitHub Actions looks like when run on "push" events from a GitHub repository is shown in Figure 1-6.

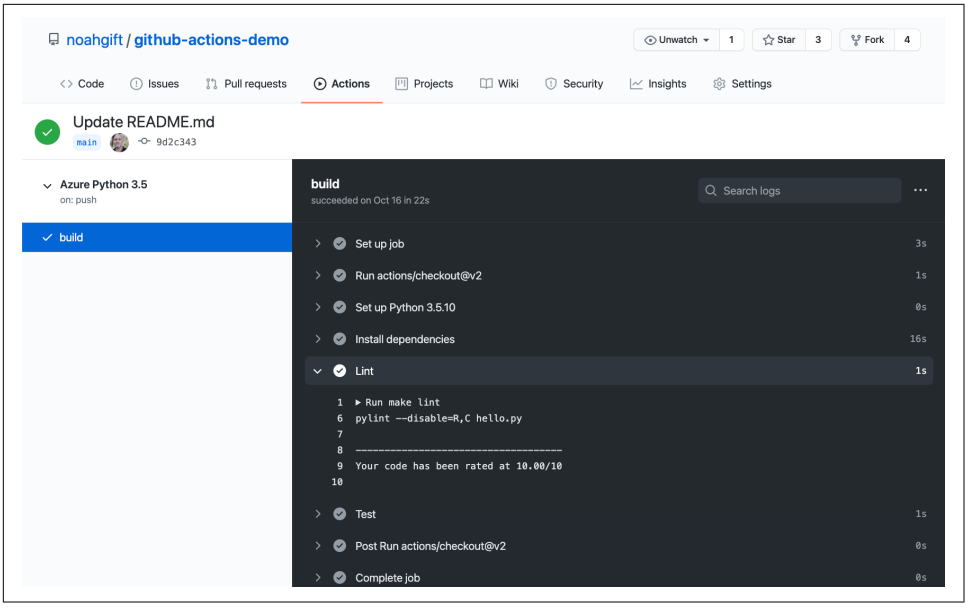


*Figure 1-6. GitHub Actions*

This step completes the final portion of setting up continuous integration. A continuous deployment—i.e., automatically pushing the machine learning project into production—would be the next logical step. This step would involve deploying the code to a specific location using a continuous delivery process and IaC (Infrastructure as Code). This process is shown in Figure 1-7.

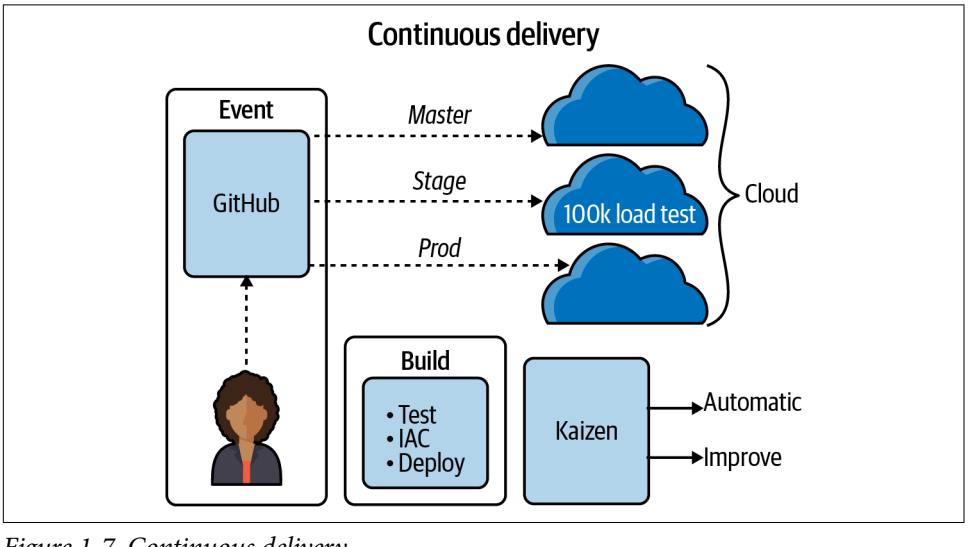


*Figure 1-7. Continuous delivery*

## DataOps and Data Engineering

Next up on the ML hierarchy of needs is a way to automate the flow of data. For example, imagine a town with a well as the only water source. Daily life is complicated because of the need to arrange trips for water, and things we take for granted may not work, like on-demand hot showers, on-demand dishwashing, or automated irrigation. Similarly, an organization without an automated flow of data cannot reliably do MLOps.

Many commercial tools are evolving to do DataOps. One example includes Apache Airflow, designed by Airbnb, then later open sourced, to schedule and monitor its data processing jobs. AWS tools include AWS Data Pipeline and AWS Glue. AWS Glue is a serverless ETL (Extract, Load, Transform) tool that detects a data source's schema and then stores the data source's metadata. Other tools like AWS Athena and AWS QuickSight can query and visualize the data.

Some items to consider here are the data's size, the frequency at which the information is changed, and how clean the data is. Many organizations use a centralized data lake as the hub of all activity around data engineering. The reason a data lake is helpful to build automation around, including machine learning, is the "near infinite" scale it provides in terms of I/O coupled with its high durability and availability.

A data lake is often synonymous with a cloud-based object storage system such as Amazon S3. A data lake allows data processing "in place" without needing to move it around. A data lake accomplishes this through near-infinite capacity and computing characteristics.

When I worked in the film industry on movies like *Avatar*, the data was immense; it did need to be moved by an excessively complicated system. Now with the cloud, this problem goes away.

Figure 1-8 shows a cloud data lake–based workflow. Note the ability to do many tasks, all in the exact location, without moving the data.

Dedicated job titles, like data engineer, can spend all of their time building systems that handle these diverse use cases:

- Periodic collection of data and running of jobs
- Processing streaming data
- Serverless and event-driven data
- Big data jobs
- Data and model versioning for ML engineering tasks

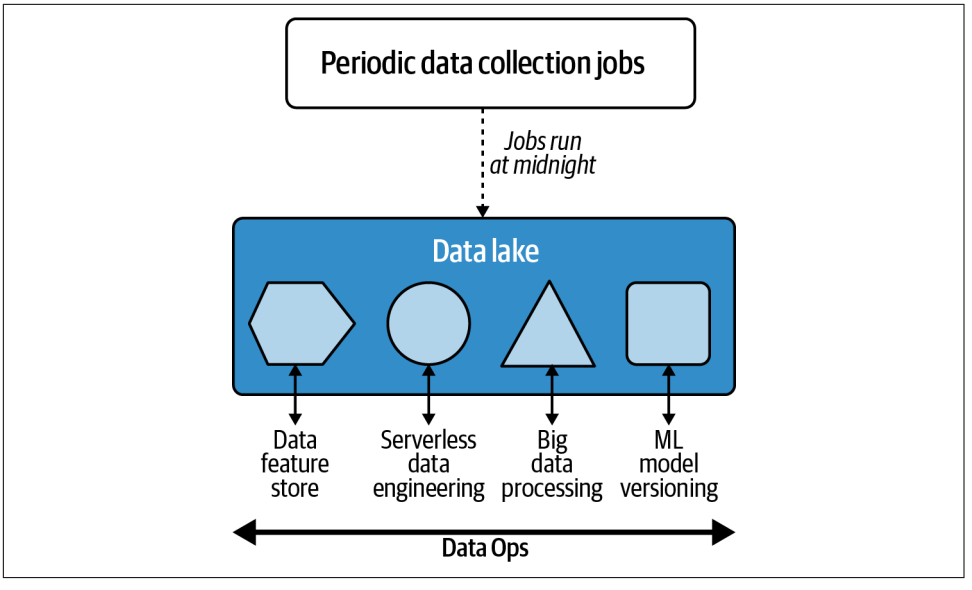


*Figure 1-8. Data engineering with a cloud data lake*

Much like a village without running water cannot use an automated dishwashing machine, an organization without data automation cannot use advanced methods for machine learning. Therefore, data processing needs automation and operationalization. This step enables ML tasks further down the chain to operationalize and automate.

## Platform Automation

Once there is an automated flow of data, the next item on the list to evaluate is how an organization can use high-level platforms to build machine learning solutions. For example, if an organization is already collecting data into a cloud platform's data lake, such as Amazon S3, it is natural to tie machine learning workflows into Amazon Sagemaker. Likewise, if an organization uses Google, it could use Google AI Platform or Azure to use Azure Machine Learning Studio. Similarly, Kubeflow would be appropriate for an organization using Kubernetes versus a public cloud.

An excellent example of a platform that solves these problems appears in Figure 1-9. Notice that AWS SageMaker orchestrates a complex MLOps sequence for a real-world machine learning problem, including spinning up virtual machines, reading and writing to S3, and provisioning production endpoints. Performing these infrastructure steps without automation would be foolhardy at best in a production scenario.

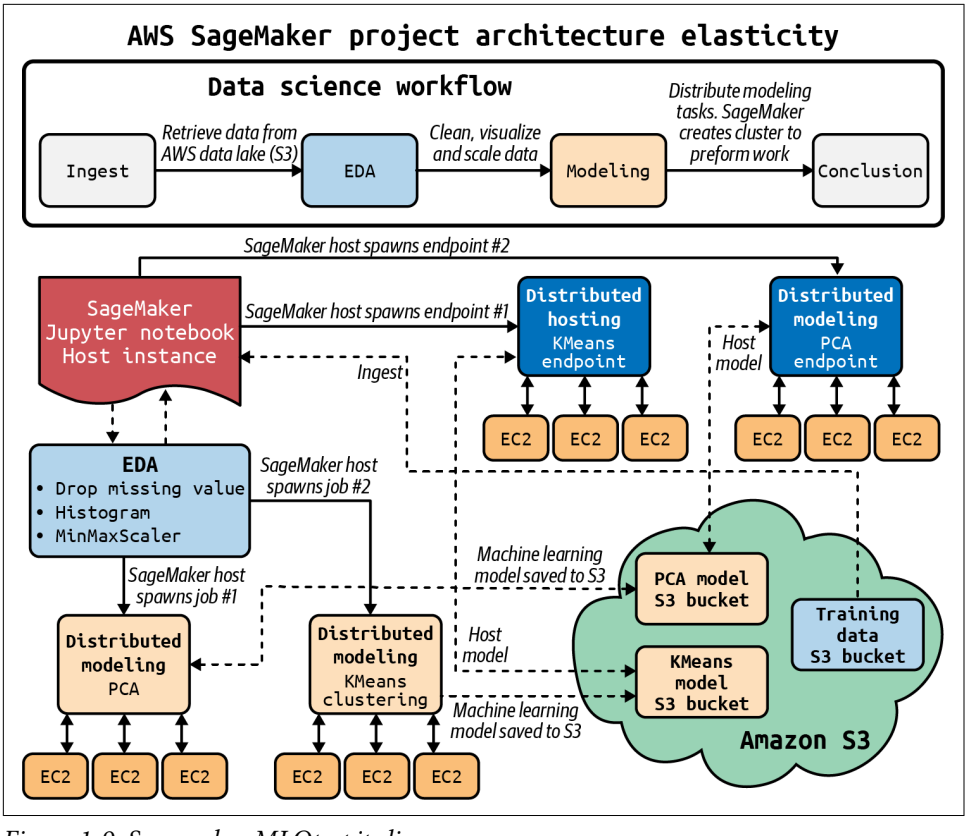


*Figure 1-9. Sagemaker MLOps pipeline*

An ML platform solves real-world repeatability, scale, and operationalization problems.

## MLOps

Assuming all of these other layers are complete (DevOps, Data Automation, and Platform Automation) MLOps is possible. Remember from earlier that the process of automating machine learning using DevOps methodologies is MLOps. The method of building machine learning is machine learning engineering.

As a result, MLOps is a behavior, just as DevOps is a behavior. While some people work as DevOps engineers, a software engineer will more frequently perform tasks using DevOps best practices. Similarly, a machine learning engineer should use MLOps best practices to create machine learning systems.

### DevOps and MLOps combined best practices?

Remember the DevOps practices described earlier in the chapter? MLOps builds on those practices and extends specific items to target machine learning systems directly.

One way to articulate these best practices is to consider that they create reproducible models with robust model packaging, validation, and deployment. In addition, these enhance the ability to explain and observe model performance. Figure 1-10 shows this in more detail.

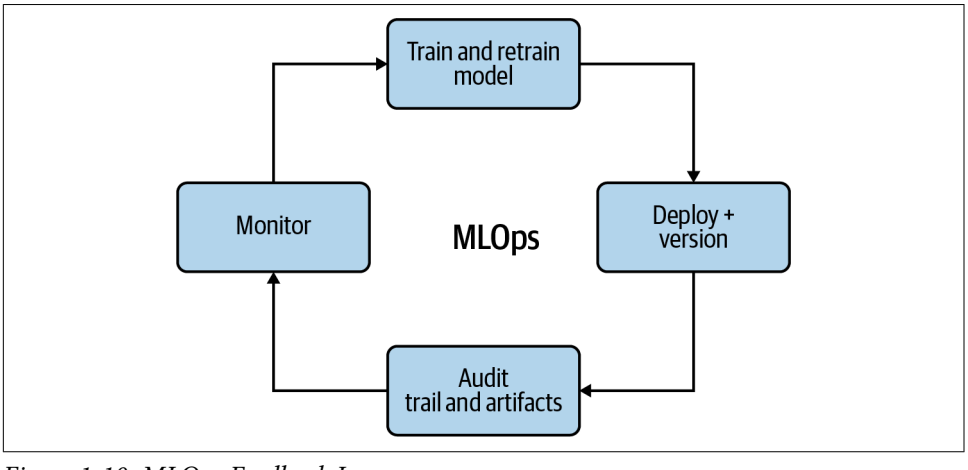


*Figure 1-10. MLOps Feedback Loop*

The feedback loop includes the following:

*Create and retrain models with reusable ML Pipelines*
: Creating a model just once isn't enough. The data can change, the customers can change, and the people making the models can change. The solution is to have reusable ML pipelines that are versioned.

*Continuous Delivery of ML Models*
: Continuous delivery of ML Models is similar to continuous delivery of software. When all of the steps are automated, including the infrastructure, using IaC, the model is deployable at any time to a new environment, including production.

*Audit trail for MLOps pipeline*
: It is critical to have auditing for machine learning models. There is no shortage of problems in machine learning, including security, bias, and accuracy. Therefore, having a helpful audit trail is invaluable, just as having adequate logging is critical in production software engineering projects. In addition, the audit trail is part of the feedback loop where you continuously improve your approach to the problem and the actual problem.

*Observe model data drift use to improve future models*

One of the unique aspects of machine learning is that the data can literally "shift" beneath the model. Thus, the model that worked for customers two years ago most likely won't work the same today. By monitoring data drift, i.e., the delta of changes from the last time a model training occurred, it is possible to prevent accuracy problems before causing production issues.

## Where Can You Deploy?

A key aspect of MLOps is creating a cloud native platform model and then deploying it to many different targets, as shown in Figure 1-11. This ability to build once and deploy many times is a critical feature in modern machine learning operations. On the one hand, deploying a model to an HTTP endpoint that can elastically scale is a typical pattern but not the only way. A new paradigm of edge machine learning uses dedicated, specialized processers called ASICS. Examples of this include Google's TPU and Apple's A14.

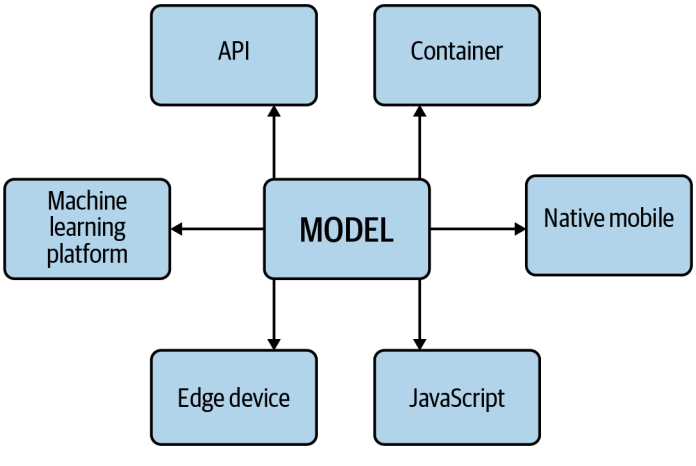


*Figure 1-11. Machine learning model targets*

In Figure 1-12, the cloud platform could use AutoML, as in Google AutoML vision, which deploys TensorFlow to TF Lite, TensorFlow.js, Core ML (an ML framework from Apple), a Container, or Coral (edge hardware that uses a TPU).

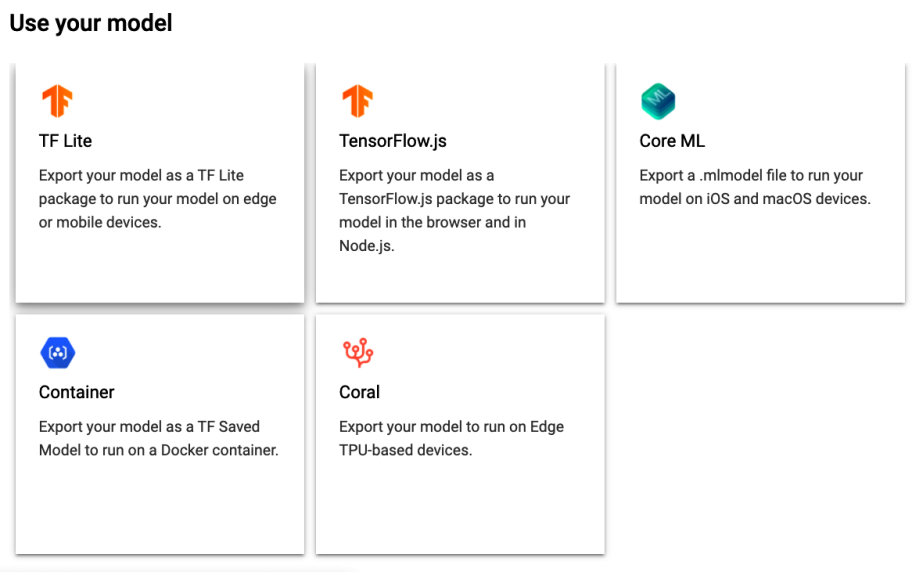


*Figure 1-12. GCP AutoML*

# Conclusion

This chapter discussed the importance of using DevOps principles in the context of machine learning. Beyond just software, machine learning adds the new complexities of managing both the data and the model. The solution to this complexity is to embrace automation the way the software engineering community has done with DevOps.

Building a bookshelf is different from growing a tree. A bookshelf requires an initial design then a one-time build. Complex software systems involving machine learning are more like growing a tree. A tree that grows successfully requires multiple dynamic inputs, including soil, water, wind, and sun.

Likewise, one way to think about MLOps is the rule of 25%. In Figure 1-13, software engineering, data engineering, modeling, and the business problem are equally important. The multidisciplinary aspect of MLOps is what makes it tough to do. However, there are many good examples of companies doing MLOps following this rule of 25%.

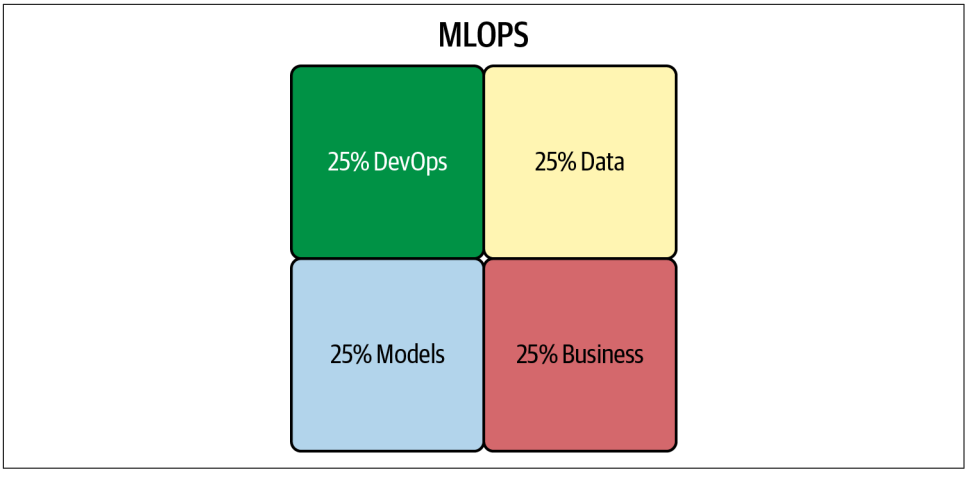


*Figure 1-13. Rule of 25%*

Tesla cars are one good example; they provide customers with what they want in the form of semi-autonomous vehicles. They also have excellent software engineering practices in that they do constant updates. Simultaneously, the car's system continuously trains the model to improve based on new data it receives. Another example of a product following the rule of 25% is the Amazon Alexa device.

Foundational skills necessary for MLOps are discussed in the next chapter. These include math for programmers, examples of data science projects, and a complete end-to-end MLOps process. By doing the recommended exercises at the end of this chapter, you put yourself in a great position to absorb the following content.

# Exercises

- Create a new GitHub repository with necessary Python scaffolding using a `Make file`, linting, and testing. Then, perform additional steps such as code formatting in your Makefile.
- Using GitHub Actions, test a GitHub project with two or more Python versions.
- Using a cloud native build server (AWS Code Build, GCP CloudBuild, or Azure DevOps Pipelines), perform continuous integration for your project.
- Containerize a GitHub project by integrating a Dockerfile and automatically registering new containers to a Container Registry.
- Create a simple load test for your application using a load test framework such as locust or loader io and automatically run this test when you push changes to a staging branch.

## Critical Thinking Discussion Questions

- What problems does a continuous integration (CI) system solve?
- Why is a CI system an essential part of both a SaaS software product and an ML system?
- Why are cloud platforms the ideal target for analytics applications? How do data engineering and DataOps assist in building cloud-based analytics applications?
- How does deep learning benefit from the cloud? Is deep learning feasible without cloud computing?
- Explain what MLOps is and how it can enhance a machine learning engineering project.