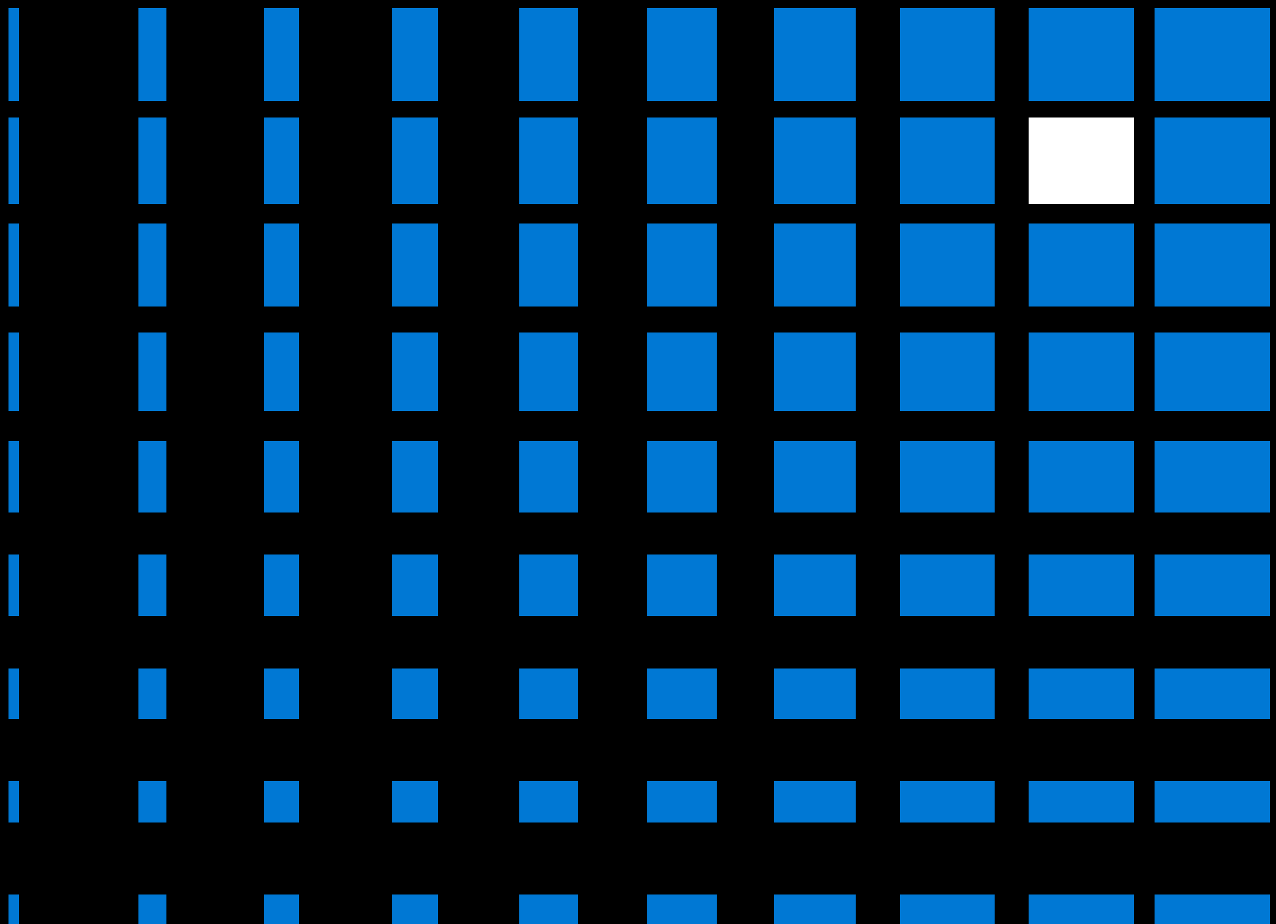
White paper

Microsoft Azure

Drive Efficiency and Productivity with Machine Learning Operations

Implement DevOps for your machine learning pipeline

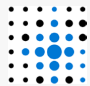


If you're a data scientist or machine learning (ML) engineer, Azure Machine Learning is built for you. It has built-in MLOps, making it easy to implement DevOps for machine learning through comprehensive support for all phases of the ML life cycle.

With all that Azure Machine Learning provides, you'll no longer be forced to work in inefficient ways or be burdened with mundane tasks. Instead, you'll have the means to streamline and automate the end-to-end ML life cycle and tie it into existing DevOps processes, so you can collaborate with app developers and work at the same cadence when building ML-infused apps.

Who should read this

We wrote this paper for ML professionals who are looking for a better way to work—with access to the same types of sophisticated tooling and streamlined engineering practices that developers working in a modern DevOps environment take for granted. We start by examining existing ML workflows, including some major coverage gaps from a tooling and automation perspective. We then introduce MLOps and what's needed for it. Following that, we explore how Azure Machine Learning features and capabilities map into the MLOps and DevOps life cycles. Finally, we provide a quick guide to getting started with Azure Machine Learning and exploring all the ways it can make you more productive.



Get started with Azure Machine Learning today

To start using Azure Machine Learning, you'll need an Azure account. If you don't already have an Azure subscription, you can create one for free and take advantage of a USD 200 credit you can spend during the first 30 days.

1. Go to the Azure Machine Learning Free Account page, and select **Start free**.
2. Sign in with your Microsoft (or GitHub) account. If you don't already have a Microsoft account, you'll be able to create one.
3. Fill in your name, phone number, and email address.
4. Verify your identity with a text code and your credit card information. (We won't charge your card until your trial is complete and you're ready to choose a subscription plan.)

After you select **Continue**, you'll be ready to go hands-on with Azure Machine Learning. Let us know if you have any questions.

Contents

05 Introduction
06 Framing the challenge
- 07 How things work today
- 07 So what's missing?
- 08 The desired state

9 MLOps–a life-cycle approach
- 11 Source control and reproducibility
- 11 Versioning and storage
- 12 Packaging
- 13 Validation
- 13 Deployment
- 14 Continuous retraining

15 Azure Machine Learning is MLOps-ready
17 How it works
- 18 Train a reproducible model
- 22 Package your model
- 23 Validate your model

Contents

24 Deploy your model

24 Monitor your model

25 Continuous retraining

25 Wrapping it up

27 Getting started with Azure Machine Learning

28 Step-by-step tutorials

28 Samples

28 How-to guides

28 References

© 2019 Microsoft Corporation. All rights reserved.
This document is provided "as is." Information and views expressed in this document, including URL and other internet website references, may change without notice. You bear the risk of using it. This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

Introduction

If you work with code at all, you've probably heard of DevOps. An approach to application life-cycle management, it employs an (ideally, fully automated) continuous integration / continuous deployment (CI/CD) pipeline to streamline the process of building, testing, and deploying new code into a production environment. This provides several benefits, from an easier way of working for application developers to faster and more efficient operationalization of new code, all contributing to the accelerated delivery of business value.

While the goals of DevOps have remained the same over the past several years, the means of achieving those goals have evolved to support new technologies. Cloud-based apps and containerization are two good examples. But what about machine learning (ML), which is playing an increasingly greater role in modern applications? Recommender systems, image classifiers, facial recognition scanners, fraud detection pipelines for advertising clicks, and speech-to-text translation services are all good examples of ML-infused scenarios. They're enabled by the data scientists who build and validate ML models and by the ML engineers who package, deploy, and run those models. Yet these same people rarely have access to the sophisticated tooling and streamlined, standardized engineering practices that developers working in a modern DevOps environment take for granted.

The net effect is that data scientists and ML engineers often work in inefficient ways and are burdened with manual tasks, such as handoffs to developers within whose apps their ML models will ultimately run. This creates an impediment to using ML models on the same cadence—and in the same DevOps processes—as the application, which ultimately slows the delivery of ML-enabled applications and reduces the business return on investing in such solutions.

In the rest of this paper, we:

- Take a deeper look at existing ML workflows—including some major coverage gaps from a tooling and automation perspective.
- Introduce *MLOps*—a construct that brings data scientists and ML engineers into the modern DevOps world—and what's needed for it.
- How Azure Machine Learning delivers on this MLOps vision—including an explanation of its key capabilities and how they map into the full ML and DevOps life cycles.

01 /

Framing the challenge

DevOps is the standard way to manage application life cycles through a pipeline of version control, test, build, and deployment tools. In the best case, it means a fully automated CI/CD pipeline—from a software engineer submitting the code into central version control (most commonly a Git repository) to building, testing, and deployment to a production environment.

So, what might a similar CI/CD solution for ML look like? It would require pipelines for automation, validation of models for functionality and performance, and support for deployment to the infrastructure used for inferencing. This becomes particularly challenging when data changes over time and models need to be retrained regularly, as is the case in many large-scale, ML-infused systems. Complexity grows further as models are deployed to a hybrid of intelligent edge plus intelligent cloud.

However, because the ML field is young compared to traditional software development, best practices and solutions around ML life-cycle management have yet to solidify. Model development is often done on the data scientist's laptop, maybe in a Jupyter notebook or other tool, and orchestration is often done manually, or ad hoc, using custom code and scripts—as was the case in traditional app development before DevOps best practices emerged.

How things work today

DevOps teams strive to automate all their workflows, and for the most part today they have. But data scientists tend to be in a silo outside the automation, which creates an impediment to using ML models on the same cadence—and in the same DevOps processes—as app developers, who have well-established practices for getting their apps built, tested, and shipped.

For developers, the process begins by writing code, which gets checked in to the front end of the DevOps pipeline. Data scientists, on the other hand, tend to do a lot more experimentation upfront. They spend a lot of time shaping data, figuring out which features are most relevant to their scenario, determining which algorithm is best to solve their problem, and tuning associated hyperparameters. Only after they've found the right recipe for all this are they ready to train their ML model—the first step in the ML pipeline we've defined. (And although it's outside the scope of this paper, there's a good deal of work done further upstream by data engineers, who typically cleanse, standardize, and otherwise wrangle the data before handing it off to the data scientist.)

After the data scientist creates a viable ML model, that's typically when they begin to interact with a developer for integration of that model into an app. Typically, this is done via manual embedding, or by sharing of basic interfaces. Integration costs can climb if featurization code needs to be rewritten to work with the inferencing stack, if there's a mismatch between the data that's available during training versus inferencing, and so on.

So what's missing?

Today, that's pretty much how the process works. Now let's examine where it begins to fall short.

First, you need **source control** as a means of enabling collaboration with other data scientists in addition to developers. To ensure reproducibility, it's paramount that all artifacts used to generate ML models (including as training code and input data) are captured via source control. This is important because there are many scenarios where models must be retrained and redeployed, such as when the model is dependent on time series data or on a specific set of input signals that may change over time.

Next, you'll want a **reproducible training pipeline** to streamline model development by automating repetitive tasks. Automation, in turn, will increase the velocity at which new models are generated, which creates the need for **model storage and versioning**. You'll want to do this in a way that promotes easy discoverability, sharing, and collaboration. You'll also want access control and traceability so that you can control who has access to (and know who used) what.

After you have the model you want, you'll need to address **model packaging**, which involves capturing the dependencies required for the model to run in its target inferencing environment. Containerization is the obvious choice; containers are the de facto unit of execution today across both the cloud and intelligent edge. You'll also want to consider model formats that are agnostic of training and serving fabric, which is where reusable formats such as Open Neural Network Exchange (ONNX) can be useful.

Next, you'll need to address **model validation**. This validation can be as simple as basic unit tests for the training code, an A/B comparison against a previous version of a model, or an end-to-end functional and performance test suite.

When models are reproducible within an automated pipeline, model storage and versioning are solidified, and packaging and validation methods are in place, you'll have an end-to-end pipeline that can support the **deployment** of new models to a variety of platforms across all scenarios.

Finally, you'll need the ability to **monitor** your models in a production environment, as a means of understanding what data is being sent to your model and the predictions that it returns. You'll also want to monitor for data drift between your training dataset and inference data so that you know if and when your model needs to be **retrained**—and fed back into your automated pipeline.

The desired state

The end-to-end ML life cycle described earlier looks something like the one shown in Figure 1. In larger enterprises, a data scientist might own the first few steps and an ML engineer might own the rest. In smaller companies, a data scientist might own them all. For the sake of simplicity, in the rest of this paper, we attribute them all to the data scientist.

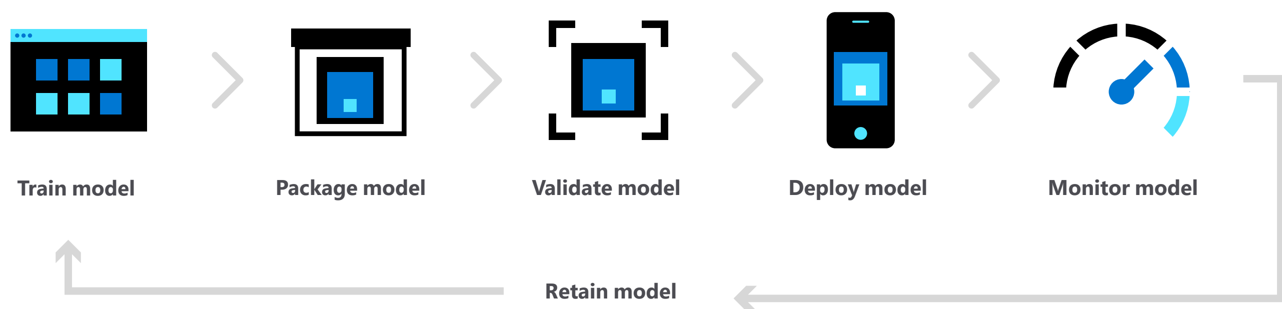


Figure 1. The end-to-end ML life cycle.

02 /

MLOps: A life-cycle approach

MLOps, a relatively new industry trend, is attempting to do for data scientists and ML engineers what DevOps does for developers. It's not a product or service, but rather a concept or a way of working. You might describe it as an approach for streamlining the end-to-end ML life cycle, in a way that ties into existing DevOps processes and tooling—so that data scientists, ML engineers, and app developers can all effectively collaborate and work at the same cadence toward delivering ML-infused apps.

That's a fairly accurate description, but it's also a lofty one. So, let's break things down further and examine what exactly an MLOps approach needs to provide. What capabilities are needed? Can existing DevOps processes and tooling be used to deliver them? If so, to what extent? And what gaps in coverage does that leave?

To answer those questions, let's start by parsing the key requirements that we identified in bold text under So, what's missing? As a reminder, they include:

- Model source control and reproducibility—processes and procedures to make models reproducible (from source control to data retention policies).
- Model versioning and storage—facilitating easy sharing, collaboration, and model reuse in an environment that includes comprehensive access control and traceability.
- Model packaging—such as using containerization to capture all dependencies, and the use of ONNX for model interoperability and reuse across a variety of inferencing stacks.
- Model validation—unit testing, functional testing, and performance testing (both in isolation and when embedded in an application).
- Model deployment—an efficient process for building a model into an application or service, as required to enable an end-user scenario.
- Continuous retraining—as a means of handling the many scenarios where models must be retrained based on the availability of new data, continuously evolving data, or other signals.

On the following pages, we'll take a closer look at each of these key requirements. Of course, you'll also need to consider foundational elements, like support for common languages and frameworks.

Get started with Azure Machine Learning today

To start using Azure Machine Learning, you'll need an Azure account. If you don't already have an Azure subscription, you can create one for free and take advantage of a USD 200 credit you can spend during the first 30 days.

1. Go to the Azure Machine Learning Free Account page, and select **Start free**.
2. Sign in with your Microsoft (or GitHub) account. If you don't already have a Microsoft account, you'll be able to create one.
3. Fill in your name, phone number, and email address.
4. Verify your identity with a text code and your credit card information. (We won't charge your card until your trial is complete and you're ready to choose a subscription plan.)

After you select **Continue**, you'll be ready to go hands-on with Azure Machine Learning. Let us know if you have any questions.

Source control and reproducibility

Although many data scientists begin model development in a notebook or other environment that doesn't necessitate the use of source control, it can still be helpful in early phases for reusability and collaboration. When source control isn't used at all, several problems can arise:

- Models can't be reproduced, nor is there any traceability into how they were initially produced.
- Governance and compliance mandates are harder to meet.
- Teams can't collaborate across models.
- Source code is structured differently for each model.
- Data scientists can't clone and train models.
- It's harder for app developers to figure out how a model works and how to use it.

These are just a few reasons why source control is a necessary first step—a prerequisite to model reproducibility, collaboration, automation, traceability, and more. If you don't capture the training code *and* input data used to generate a model, you'll never be able to trace precisely which inputs produced the output of a model that you're using in production. Of course, there can be a delicate balance here, particularly when you're dealing with compliance mandates such as the EU's General Data Protection Regulation: although you want to ensure reproducibility by keeping data for long periods of time, you also need to give customers control of their data, including the ability to delete it.

Versioning and storage

As the model development life-cycle begins to mature and more models are produced by a growing number of data scientists, the need for effective model versioning and storage increases. Without it:

- **Sharing, collaboration, and reuse become more difficult.** Without a centralized system for logging all models, making them searchable, and allowing engineers to deploy them as microservices, organizations usually find that some teams are duplicating efforts while other teams need a model but don't have the resources to develop it.
- **You can't control who has access to (and know who used) what.** Making an organization's intellectual property freely available to anyone in the company is poor IT governance, which is why solutions for model storage need to incorporate access control. This should include ways to manage who can provide models and who can access them, plus full traceability—so that you can always determine who provided or who accessed what.

There are several strategies for model versioning. Depending on their size (for example, in comparing traditional ML classifiers with deep learning networks), models can be embedded directly into source control or stored on a versioned package feed. Regardless of which method you use, it's worth remembering that mature teams want validated and reusable models. To support this, model versioning must incorporate experiment history in some way, whether it's captured in a notebook, source control, or a central database. Such experiments can come from a variety of source locations, such as an SDK, an IDE, or a CI job—all are valid to track. Ideally, you'll want to capture both information on the model itself and on its relative accuracy for a specific scenario.

In addition, you'll want to consider model interoperability, as a means of making it easy for data scientists to experiment with a wide range of flexible tools—and ultimately get new ideas into production faster. An intermediate, cross-framework language for storing models (such as ONNX) can make it easy to reuse models across a variety of inferencing stacks—and enable switching frameworks in the training process without massive code rewrites on the inferencing side.

Packaging

Before you can put a model to use, you'll need to package it for your target inferencing environment. Containers are a logical choice because they're the default execution unit across both modern cloud environments and the intelligent edge. Containers also provide an easy way to capture runtime dependencies, including Conda environments, Python versioned libraries, and other libraries that the model might reference—all of which are required for the model to execute properly.

In packaging your model, you'll also want to consider open formats like Open Neural Network Exchange (ONNX), which can help optimize performance during inferencing. ONNX also enables you to train a model in one environment, using your preferred tool set, and then deploy it into a different environment for inferencing and prediction. PyTorch, MXNet, Caffe2, and several other popular frameworks all natively support ONNX, and there are converters for other frameworks like TensorFlow, Core ML, and scikit-learn. ONNX also is compatible with multiple runtimes, compilers, and visualizers. (You can find a complete list of all tools that support ONNX here.)

Validation

An efficient means of model validation is needed to train models continuously, ensure quality, and ultimately prevent bad models from reaching production. In addition to meeting desired functionality and performance requirements, a model must not crash or cause errors when loaded or when it's sent bad or unexpected inputs. Finally, it must not use too many system resources.

Ideally, model validation has two parts: unit and integration testing of the model itself, and functional and performance testing of the model as embedded into an app or service. For example, if you train a model with a different format of input data than what's available to the inferencing service, it might work well during the training process but perform poorly in production.

- **Testing the model itself.** Traditional unit and integration tests that are run on a small set of inputs should produce stable results. It's worth noting that, unlike a traditional (non-ML) app, these will be statistical results—that is, there will be a range of acceptable values (an expected target and its evolution) versus a finite sign-off criteria. This can also involve testing the data used to produce the model, as required to ensure that it matches what will be available during the scoring scenario in terms of schema and features.
- **Testing the app and model together.** You'll want to ensure that your model behaves correctly in the context of your larger app, which you can do by using an existing version of your app (or a stubbed-out variant of it) to execute relevant parts of the host app's own test suite. Such testing can also help ensure that data schemas (input/output) and behaviors for all base cases in an application are sufficiently covered. As they mature, most organizations build a custom stack for this level of model validation.

Deployment

When it comes to collaboration between data scientist and app developer, one of the largest pain points is the model deployment process. Typical problems include difficulty getting the model baked into an app or service, lack of mechanisms to prevent shipping a functional or performance regression, and the extensive time (and manual effort) associated with staging and release processes.

By streamlining model deployment, you'll give data scientists more time to focus on modeling tasks, and give developers more time to work on other aspects of their apps. To do this, you'll need best practices that make models easy to deploy, with quality and security, across a wide variety of inferencing targets.

Let's parse this in more detail, starting with the target for a model, which could be:

- Deployed as a standalone, containerized inferencing service.
- Used as part of a batch-processing pipeline.
- Embedded into an existing application or service.

Although the unit of deployment can vary, the goal remains the same: to make the creation and release of that deployment unit as simple, fast, and efficient as possible. To achieve this, an optimal solution for model deployment should have simple user interfaces that make it easy for engineers to deploy and monitor models, with minimal additional configuration. It should also help users with various levels of ML expertise to understand and analyze their data and models.

Some things that can make model deployment easier include:

- Providing a friendly model format, such as PNNL or ONNX.
- Simplifying the process to interact with the model—such as through code generation, API specifications, or other methods.
- Supporting a variety of inferencing targets—including cloud, app, edge, specialized hardware such as FPGAs, and dedicated frameworks such as Core ML and WinML.
- Incorporating secrets management and service endpoint management to simplify configuration.

Even with all the above, governing the release and deployment process can be a challenge, which is why you'll also need full access controls, manual and automated checks, and end-to-end auditability—especially when you're dealing with compliance issues, customer data, or both.

Continuous retraining

Finally, you'll want to establish a retraining loop. Most training pipelines are set up as workflows or dependency graphs that execute specific operations or jobs in a defined sequence: if a team needs to train over new data, the same workflow or graph can be executed again. Many real-world use cases require:

- Retraining based on the availability of new data (where the training code itself remains static).
- Retraining over evolving data (such as a moving window over the last *n* days of a log stream).
- Retraining based on other signals (such as data drift).

03 /

Azure Machine Learning is MLOps-ready

Azure Machine Learning, a cloud service, has built-in MLOps. It supports all phases of the ML model life cycle—with full support for popular, open-source Python packages such as scikit-learn, TensorFlow, PyTorch, and MXNet.

With Azure Machine Learning, you can:

- **Turn your training process into a reproducible pipeline**. Use ML pipelines to stitch together all the steps involved in your model-training process, from data preparation and feature extraction to hyperparameter tuning and model evaluation.
- **Register and track your ML models.** Use the model registry to store and version your trained models in your own workspace, in the Azure cloud, making them easy to track and organize.
- **Package and debug models.** Models are packaged into a Docker image before deployment. You can let this happen automatically in the background or manually specify the image. If you run into any problems, you can deploy locally for troubleshooting and debugging.
- **Validate and profile models.** Azure Machine Learning can profile your model to determine ideal CPU and memory settings for deployment. Model validation happens as part of this process, using data that you supply for profiling.
- **Convert and optimize models.** On average, converting your model to Open Neural Network Exchange (ONNX) format can yield a twofold performance increase. You can use Azure Machine Learning to create a new ONNX model or convert an existing model to ONNX.
- **Deploy almost anywhere.** With Azure Machine Learning, you can deploy your ML models as web services to the cloud, to your local development environment, or to Azure IoT Edge devices. Deployments can use CPU, GPU, or field-programmable gate arrays (FPGA) for inferencing. You can even embed ML models into analytics apps based on Microsoft Power BI.
- **Capture an end-to-end audit trail.** Azure Machine Learning integrates with Git to track where your code came from and with Azure ML Datasets to track and version your data. The Run History stores a snapshot of the code, data, and compute resources used to train a model, and the model registry captures all the metadata associated with your model.
- **Automate the end-to-end ML life cycle.** With the Azure Machine Learning extension, you can use Azure Pipelines to enable continuous integration by automatically starting a training run when you check a change into a Git repo. You can also create release pipelines that are triggered when new models are created in a training pipeline.

On the following pages, we'll take a deeper look at how Azure Machine Learning supports MLOps at each phase in the ML life cycle. Or, if you'd prefer, you can skip forward to Getting started with Azure Machine Learning to explore on your own.

04 /

How it works

Before we dive into the specific capabilities of Azure Machine Learning and how they enable MLOps, let's define what Azure Machine Learning is: a managed collection of cloud services for machine learning, which are offered in the form of a workspace and an SDK. Azure Machine Learning is designed to improve the productivity of the data scientists who build, train, and deploy machine learning models at scale and of the ML engineers who manage, track, and automate machine learning pipelines.

Diving a bit deeper, Azure Machine Learning consists of the following components:

- An SDK that plugs into any Python-based IDE, notebook, or CLI
- A compute environment that can support workloads of any scale and complexity—including the ability to scale up or scale out, built-in autoscaling, and the flexibility to use CPU-based or GPU-based infrastructure for training.
- A centralized model registry for keeping track of models and experiments, regardless of where or how they're created.
- Built-in support for Azure Container Instances, Azure Kubernetes Service, and Azure IoT Hub—for containerized deployment of models to the cloud and the intelligent edge.
- A monitoring service that captures and tracks metrics for models that are registered and deployed with Azure Machine Learning.

Now that we've covered that, let's break down what Azure Machine Learning brings to the table in each phase of the end-to-end ML life cycle—and how those capabilities fold into the traditional DevOps life cycle, as shown in Figure 2.

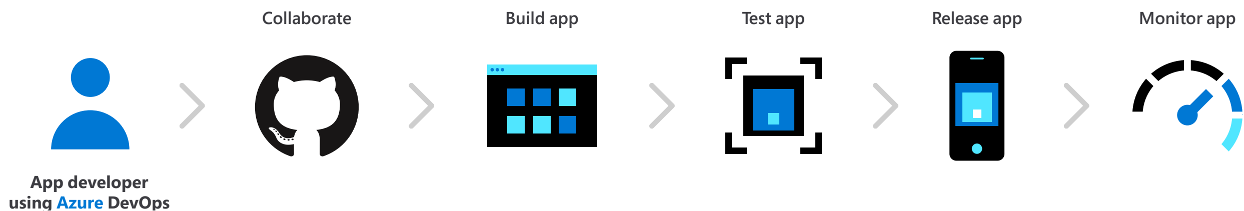


Figure 2. Typical DevOps workflow and processes.

On the following pages, we use the Azure Machine Learning names for features and capabilities to make it easier for you to become acquainted with them. And wherever our full online documentation provides more information on a topic than we have room to cover in this paper, we provide a link to that article.

Train a reproducible model

With Azure Machine Learning, you'll have everything you need to quickly and efficiently build and train a reproducible model.

Workspaces

The workspace, the top-level resource for Azure Machine Learning, provides a centralized place to work with all the artifacts that you create when using it. You can define user roles within your workspace, enabling you to share it with other users, teams, or projects.

A workspace can contain Notebook VMs, which are configured with the Python environment necessary to run Azure Machine Learning. You can create and run experiments (the training runs you use to build your models) by using the Azure Machine Learning SDK for Python, automated machine learning experiments (preview), or the Azure Machine Learning designer (preview). As you iterate, the workspace keeps a history of all training runs (including logs, metrics, output, and a snapshot of your scripts) to help you determine which run produces the best model. Your workspace is also where you define the compute targets that are used to run your experiments, and where you define and manage the datasets to use for model training and pipeline creation.

Code, dataset, and environment management

As you work, you'll need to manage and keep track of several things, starting with your code. Most likely, it's in a Git repository. Azure Machine Learning includes Git repository tracking: anytime you submit code artifacts to the service, you can specify a Git repository reference. This is done automatically when you're running from a CI/CD solution such as Azure Pipelines.

You'll also need to define and manage the data you use for model training. Azure Machine Learning datasets provide a means to version, profile, and snapshot your data, enabling you to reproduce your training process by having access to the same data. You can also compare dataset profiles to determine how much your data has changed or if you need to retrain your model.

In addition, you'll need to configure and manage the compute targets used to run your experiments, which is where ML environments managed by Azure Machine Learning can help. They're shared across all you can do with Azure Machine Learning, so you can use them to help simplify handoff from training to inferencing, or to reproduce a training environment locally. Environments provide automatic Docker image management and caching, plus tracking to help ensure reproducibility.

ML pipelines

You can use Azure Machine Learning pipelines to stitch together and streamline all the steps in your model development process—from data preparation and feature extraction to hyperparameter tuning, model evaluation, and ultimately deployment. By taking advantage of pipelines, you can optimize your workflows for speed, portability, and reuse, leaving you with more time to put your ML expertise to work.

Pipelines are constructed from multiple steps, which are distinct computational units. Each step can run independently and can use isolated compute resources. This allows multiple data scientists to work on the same pipeline at the same time without overtaxing compute resources. It also makes it easy to use different compute environments for each step.

After you design a pipeline, you'll probably still need to fine-tune the training loop within it. When you rerun a pipeline, the run jumps to the distinct steps that need to be rerun (such as an updated training script) and skips what hasn't changed. The same applies to unchanged scripts used for the execution of the step. All this can help you avoid unnecessarily rerunning costly and time-intensive steps (such as data ingestion and transformation) if the underlying data hasn't changed.

Tracking and monitoring experiments

As you iterate to find the best model, you'll want to track and monitor your experiments. The Azure Machine Learning SDK for Python and the Azure Machine Learning CLI provide various methods to monitor, organize, and manage your runs for training and experimentation. You can also log metrics for training runs by adding logging code to your training script, submitting an experiment run, monitoring that run, and inspecting the results. If you're using MLflow, an open-source library for managing ML experiments, you can use MLflow Tracking with Azure Machine Learning. In addition, you can use TensorBoard to inspect and understand experiment structure and performance.

Model registry

After you have a model that you like, you can register it in the model registry within your workspace, which makes it easy to store, version, organize, and keep track of all your trained models. Each registered model is a single logical container for one or more files, so if you have a model that's stored in multiple files, you can register them as a single model in your Azure Machine Learning workspace. When you download or deploy the registered model, it will include all the files that were registered.

Registered models are identified by name and version; when you register a model with the same name as an existing one, the registry increments the version. You can also provide additional metadata to use when searching for models. You can register models that were trained outside the Azure Machine Learning service, and you can't delete a registered model that's being used in an active deployment.

In many ways, the model registry forms the foundation of the ML life-cycle management process. It enables version control of models, stores model metrics, allows for one-click deployment, and tracks all deployments of your models—so that you can take action in case a model becomes stale or its efficacy is no longer acceptable. The model registry also plays a role in triggering other activities in the ML life cycle, such as when new changes appear or when metrics cross a threshold.

Automated ML and hyperparameter tuning

The process of trying out different algorithms and hyperparameter combinations until an acceptable model is found can be monotonous for data scientists. Although such iteration can yield massive gains in terms of the model efficacy, it imposes costs in terms of time and resources.

Automated ML in Azure Machine Learning uses concepts from the research paper on Probabilistic Matrix Factorization to implement an automated, parallelized pipeline, which tries out various algorithms and hyperparameter settings based on data heuristics—and then presents a set of models that are likely to be best suited for the given problem and dataset.

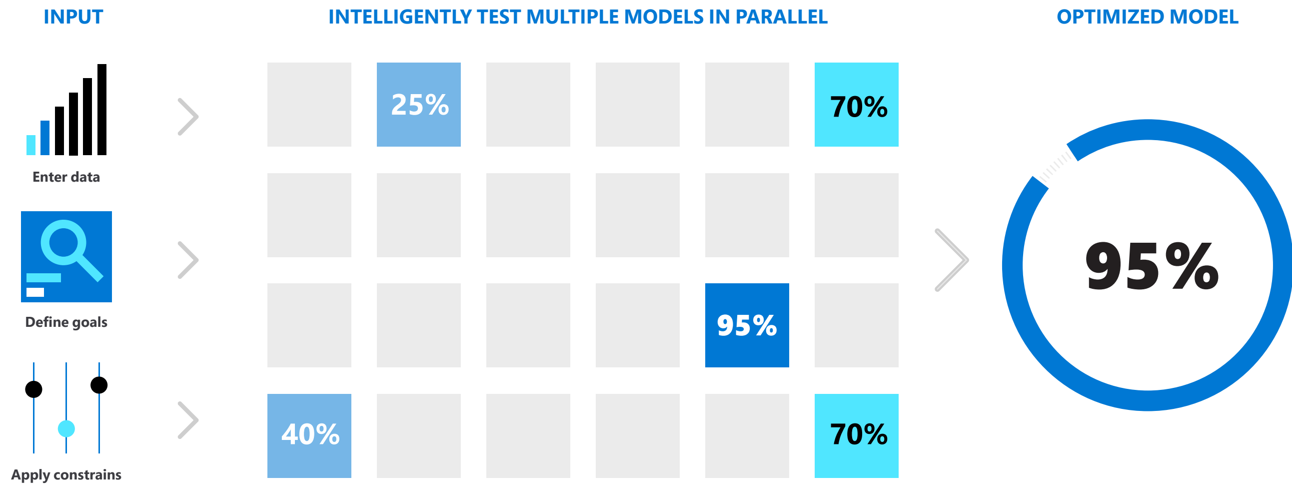


Figure 3. Automated machine learning can help find the best models for a given problem and dataset models for a given problem and dataset.

Automated ML supports classification, regression, and forecasting. It also includes features for handling missing values, early termination by a stopping metric, blocking algorithms that you don't want to explore, and other optimizations. In addition, its newly introduced UI mode (akin to a wizard) can help novice or nonprofessional data scientists to contribute more—and thus help enterprises run more ML initiatives without compromising high-value projects due to a limited supply of high-level talent.

Although you can use automated ML to streamline hyperparameter tuning, it's not the only way to do so. Azure Machine Learning also provides a hyperparameter tuning service, which supports random, grid, and Bayesian parameter sampling. You just configure the experiment by defining the search space, specifying the primary metric along with any early termination policies, and allocating resources. When you submit that experiment, the hyperparameter-tuning service handles mundane tasks such as job creation and monitoring. You can visualize the progress of your training runs by using a notebook widget that's provided in the Azure Machine Learning SDK.

Experimentation using Azure Machine Learning covers automated ML, the hyperparameter-tuning service, and other features related to machine learning experimentation.

Package your model

Before you can deploy your model (or test it), you'll need to package it for its target execution environment, which means capturing everything that it needs to run—including Conda environments, Python versioned libraries, and other libraries that the model references. Containers are the de facto standard for doing this; they're the default execution unit across almost all inferencing environments.

When you use Azure Machine Learning to deploy a model, it gets packaged into a Docker image that contains a web server to handle incoming requests. Azure Machine Learning provides a default Docker base image, so you don't have to worry about creating one unless you want to. You can use Azure Machine Learning environments to select a specific base image.

Using a custom Docker base image

You can also use a custom base image that you provide. Typically, you deploy a model using a custom Docker base image when you want to use Docker to manage dependencies, maintain tighter control over component versions, or save time during deployment. For example, you might want to standardize on a specific version of Python, Conda, or other component. Or you might want to preinstall software required by your model, especially when such installation takes a long time, so that you don't have to install it separately for each deployment.

ONNX

In packaging your model, you may want to consider using Open Neural Network Exchange (ONNX), an open format, as a means of maximizing performance during inferencing. Here's why: optimizing ML models for inference can be difficult because you need to tune the model and the inference library to make the most of the hardware capabilities. The problem gets harder when you want to optimize for different kinds of platforms (cloud/edge, CPU/GPU, and so on) because each one has different capabilities and characteristics. Complexity further increases when you have models from a variety of frameworks that need to run on a variety of platforms, making it highly time-consuming to optimize for all the different combinations.

With ONNX, you can train your model once, in one environment, using your preferred tool stack, and then deploy it into a different environment for inferencing and prediction. Models from many frameworks (including PyTorch, MXNet, Caffe2, TensorFlow, Core ML, scikit-learn, Keras, Chainer, MXNet, MATLAB, and others) can be exported or converted to the standard ONNX format.

When the models are in the ONNX format, they can be run on numerous platforms and devices. The ONNX Runtime, a high-performance inference engine for deploying ONNX models to

production, is optimized for both cloud and edge, and it works on Linux, Windows, and Mac. Written in C++, it also has C, Python, and C# APIs. The ONNX Runtime provides support for all the ONNX-ML specification, and it also integrates with accelerators on different hardware, such as TensorRT on NVIDIA GPUs.

The ONNX Runtime is used in high-scale Microsoft services such as Bing, Office, and Cognitive Services. Although the specific performance gains you'll experience are dependent on a number of factors, these Microsoft services have seen an average twofold performance gain on CPUs. The ONNX Runtime is also used as part of Windows ML on hundreds of millions of devices. Of course, you can also use the ONNX Runtime with Azure Machine Learning—it enables runtime bindings and easy portability to ONNX models—and benefit from its extensive production-grade optimizations, testing, and ongoing improvements.

Validate your model

After your model is packaged, you'll be ready to test it. Because it's packaged in a container, the ideal way to test it is with Azure Container Instances, which provides an easy, cost-effective mechanism for container deployment. You can easily deploy to an ACI container and then inference against it for testing. In fact, you won't even need to create a container ahead of time—it'll happen automatically as part of the deployment process.

Validation and profiling

With the model validation and profiling capabilities in Azure Machine Learning, you can provide sample input queries to help ensure that your model will perform as expected when it's deployed. Azure Machine Learning automatically deploys and tests the packaged model on a variety of inference CPU/memory configurations to determine the optimal settings to use when deploying your model. Model validation happens as part of all this, using data that you supply for the profiling process.

Model interpretability

As you test your model, you may question why it made the predictions it did. By using the various interpretability packages within the Azure Machine Learning Python SDK, you can verify hypotheses, validate that model behavior matches your objectives, and check for bias, all of which can ultimately help build trust with stakeholders. Using the classes and methods in the SDK, you can get importance values for both raw and engineered features (the data fields used to predict a target data point). During training, you can apply the interpretability classes and methods to understand

the model's global behavior (called global explanation) or specific predictions (called local explanation). Explainers also can be deployed along with the model and used for local explanation at inferencing time.

Deploy your model

Following a thorough round of validation, if everything looks good, you'll be ready to deploy your model into production. At this point, you have several options:

- If you want scale, flexibility, and extensive logging and monitoring, you may want to deploy to Azure Kubernetes Services, which can be sized however you need.
- If your models are small and you're certain you won't need to scale out, you can deploy to an Azure Container Instance, which also provides extensive monitoring and logging.
- If you're using Azure IoT Edge, you can deploy IoT Edge modules directly to Linux-based IoT devices.
- If inferencing performance is paramount, you can deploy to field programmable gate arrays (FPGAs) on Azure—it's the world's largest cloud investment. Before planning to do that, though, you should verify that your specific use case is supported.

You can deploy to several other environments, too, including GPU, Azure App Service, and Notebook VMs. If you have an existing model that was trained somewhere else, you can still use Azure Machine Learning to deploy it.

Monitor your model

After your model is deployed, you'll want to monitor it, as a means of understanding what data is being sent to your model, the predictions that it returns, and ultimately how it's being used. You'll also want to monitor for data drift so that you'll know if and when your model needs to be retrained.

To do this, you'll need to capture and analyze a lot of metrics, which you can do by enabling data collection. Collected data is stored in Azure Blob storage, making it easy to validate and analyze using whatever tools you prefer. You can also have Azure Machine Learning monitor for data drift—it's one of the top reasons why model accuracy degrades over time—and send you an email alert when it's detected. Finally, you can use Azure Application Insights with Azure Machine Learning, as a means of monitoring request rates, dependency rates, response times, failure rates, and more.

Continuous retraining

Over time, you may need to retrain your model to improve its accuracy, performance, or both. Ideally, you'll want this to be a fully automated CI/CD process. By using the Azure DevOps extension for Azure Machine Learning, you can watch both the Azure Machine Learning model registry and the GitHub repository containing your Python notebooks and scripts, and then trigger Azure Pipelines that automatically retrain and redeploy your model based on new code commits or when new versions of a model are registered.

Such capabilities can be extremely powerful. They enable data science teams to configure stages for build and release pipelines within Azure DevOps for their ML models, and thus fully automate the process. What's more, since Azure DevOps is the environment used to manage app life cycles, we've now come full circle by enabling data science teams and app development teams to collaborate seamlessly—and trigger new versions of ML-infused apps within the DevOps environment whenever certain conditions are met during the MLOps life cycle.

Wrapping it up

With Azure Machine Learning, you'll have everything you need to implement a comprehensive MLOps approach—including the ability to track, version, audit, and reuse every asset in the ML model life cycle. You'll also have the means to streamline and automate that life cycle from end to end, and to tie it into existing DevOps processes so that data scientists, ML engineers, and app developers can all collaborate and work at the same cadence in delivering ML-infused apps.

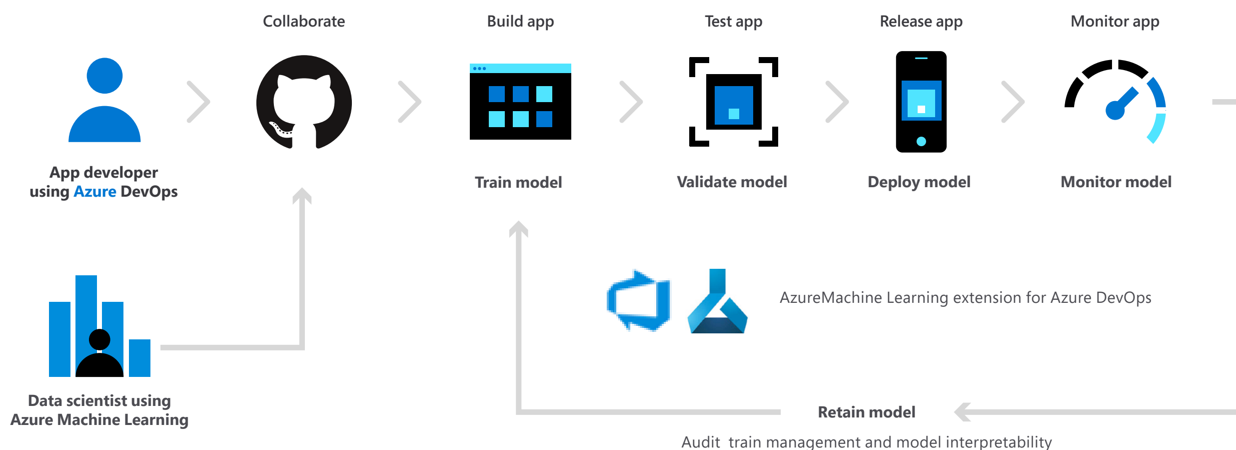


Figure 4. Azure Machine Learning provides data scientists and ML engineers with sophisticated tooling and streamlined practices, just like developers working in a DevOps environment.

Taken on their own, many of the features and capabilities we've discussed may seem logical and intuitive. However, until now, nobody has been able to bring it all together in a way that just works for everyone involved, enabling them to be immensely more productive.

If you're ready to try out Azure Machine Learning for yourself, just go to Getting started and we'll walk you through it, including how to sign up for a free trial. Or, if you're not ready for that just yet, here are some other resources:

- Learn more about Azure Machine Learning.
- Check out the Azure Machine Learning service MLOps repository on GitHub.
- Watch the Microsoft Build 2019 session on MLOps.
- Watch the Microsoft Ignite 2019 session on MLOps.
- Download our free e-books:
    - Principles of Data Science
    - Thoughtful Machine Learning with Python

05 /

Getting started with Azure Machine Learning

We've covered several key Azure Machine Learning concepts briefly in this guide, including pointers to the full online documentation for each one. If you want to read up on a topic that we haven't covered, you can find the full set of articles online; you'll find the first one here, with the rest listed immediately below it in the navigation pane. At a minimum, before getting started, you may want to review the overview of Azure Machine Learning service and its workflow model.

Step-by-step tutorials

Ready to jump into one of our tutorials? If you don't already have an Azure subscription, you can create a free account before you begin.

As a first step, you might want to try using the Python SDK to create your first experiment by setting up a workspace and development environment, and then training your first model. Or try one of our other Python SDK tutorials:

- Train a simple logistic regression image classification model, and then deploy it.
- Use automated machine learning to create a regression model.
- Batch score a classification model using the Python SDK and pipelines.

If you'd prefer to start with the visual interface, you can check out the tutorials for using it to train a regression model and then deploy it. Or try creating an automated machine learning experiment through the workspace landing page—all without writing a single line of code.

Samples

We also have a wealth of Python SDK–based code samples to help you get up to speed. If you completed the setup environment and workspace tutorial mentioned earlier, you'll already have a dedicated notebook server preloaded with the SDK and sample repository. If you'd prefer to bring your own notebook server for local development or get samples on the Data Science Virtual Machine, a customized VM image for doing data science, you'll need to take a few extra steps.

How-to guides

After completing a few tutorials and sample experiments, you'll be ready to start exploring all you can do with Azure Machine Learning on your own. Our online how-to guides can walk you through many of the things you'll likely want to do. There are too many to list in this document, but you can find the first one here, and the rest are listed immediately below it in the navigation pane.

References

Finally, if you need reference materials, check out our documentation on the Azure Machine Learning SDK for Python and the Azure Machine Learning CLI. There's also a full set of reference materials on the visual interface modules—again, see the navigation pane for a full list.