Chapter 15. Model Management and Delivery

In this chapter, we'll be discussing model management and delivery. We'll start with a discussion of experiment tracking, and we'll introduce MLOps and discuss some of the core concepts and levels of maturity for implementing MLOps processes and infrastructure. We'll also discuss workflows at some depth, along with model versioning. We'll then dive into both continuous delivery and progressing delivery.

Experiment Tracking

Experiments are fundamental to data science and ML. ML in practice is more of an experimental science than a theoretical one, so tracking the results of experiments, especially in production environments, is critical to being able to make progress toward your goals. We need rigorous processes and reproducible results, which has created a need for experiment tracking.

Debugging in ML is often fundamentally different from debugging in software engineering, because it's often about a model not converging or not generalizing instead of some functional error such as a segmentation fault or stack overflow. Keeping a clear record of the changes to the model

and data over time can be a big help when you're trying to hunt down the source of the problem.

Even small changes, such as changing the width of a layer or the learning rate, can make a big difference in both the model's performance and the resources required to train the model. So again, tracking even small changes is important.

And don't forget that running experiments, which means training your model over and over again, can be very time-consuming and expensive. This is especially true for large models and large datasets, particularly when you're using expensive accelerators such as GPUs to speed things up, so getting the maximum value out of each experiment is important.

Let's step back and think for a minute about what it means to track experiments. First, you want to keep track of all the things you need in order to duplicate a result. Some of us have had the unfortunate experience of getting a good result and then making a few changes that were not well tracked—and then finding it hard to get back to the setup that produced that good result.

Another important goal is being able to meaningfully compare results. This helps guide you when you're trying to decide what to do in your next experiment. Without good tracking, it can be hard to make comparisons of more than a small number of experiments. So it's important to track and

manage all the things that go into each of your experiments, including your code, hyperparameters, and the execution environment, which includes things such as the versions of the libraries you're using and the metrics you're measuring.

Of course, it helps to organize them in a meaningful way. Many people start by taking freeform notes, which is fine for a very small number of simple experiments, but quickly becomes a mess.

And finally, because you're probably working in a team with other people, good tracking helps when you want to share your results with your team. That usually means that as a team you need to share common tooling and be consistent.

In this section, we will look at experimenting in notebooks, and we'll discuss tools for experiment tracking.

Experimenting in Notebooks

At a basic level, especially when you're just starting out on a new project, most or all of your experiments might be in a notebook. Notebooks are powerful and friendly tools for ML data and model development, and they allow for a nice, iterative development process, including inline visualizations. However, notebook code is usually not directly promoted to production and is often not well structured. One of the reasons that it's not usually promoted is that notebooks aren't just product code. They often

contain *notebook magics*, which are special annotations that only work in the notebook environment, and development-focused code such as code to check the values of things and code to generate visualizations, which you rarely want to include in a production workflow.

But when you're experimenting with notebooks, you do want to make sure to track those experiments, and the following tools that can help with this:

`nbconvert`

: Can be used to extract just the Python code from a notebook, among other things

`nbdime`

: Enables diffing and merging of Jupyter Notebooks

`Jupytext`

: Converts and synchronizes notebooks with a matching Python file, and much more

`neptune-notebooks`

: Helps with versioning, diffing, and sharing notebooks

So, for example, to make sure that when you extract the Python from your notebook it will actually run, you can use `nbconvert`:

```
jupyter nbconvert –to script train_model.ipynb py
```

```
python train_model.py
```

This should extract the code from your notebook so that you can then try running it. If it fails, there were things happening in your notebook that your code depended on, like perhaps notebook magics.

Experimenting Overall

As you move from simple, small experiments into production-level experiments, you'll quickly outgrow the pattern of putting everything in a notebook.

Not just one big file

You should plan to write modular code, not monolithic code, and the earlier in the process you do this, the better. Because you'll tend to do many core parts of your work repeatedly, you'll develop reusable modules that will become high-level tools, often specific to your environment, infrastructure, and team. Those will save you a lot of time, and they'll be much more robust and maintainable. They'll also make it easier to understand and reproduce experiments. The simplest form of these is just directory hierarchies, especially if your whole team is working in a monorepo.

But in a more advanced and distributed workflow, you should be using code repositories and versioning with commits, unit testing, and continuous

integration. These are powerful and widely available tools for managing large projects, including ML experiments. In these cases, you probably want to keep experiments separate if you're using a shared monorepo with your team so that your commits don't version the rest of the team's repo.

Tracking runtime parameters

As you perform experiments, you're often changing runtime parameters, including your model's hyperparameters. It's important to include the values of those parameters in your experiment tracking, and how you set them will determine how you do that. A simple and robust method is to use configuration files, and change those values by editing those files. The files can be versioned along with your code for tracking.

Another option is to set your parameters on the command line, but this requires additional code to save those parameter values and associate them with your experiment. This means including code along with your experiment to save those values in a datastore somewhere. This is an additional burden, but it also makes those values easily available for analysis and visualization, rather than having to parse them out of a specific commit of a config file. Of course, if you do use config files, you can also include the code along with your experiment to save those values in a datastore somewhere, which gives you the best of both worlds.

Here is an example of what the code to save your runtime parameter values might look like if you were setting your runtime parameters on the command line. This example uses the Neptune-AI API:

```
parser = argparse.ArgumentParser()
parser.add_argument('--number_trees')
parser.add_argument('--learning_rate')
args = parser.parse_args()
neptune.create_experiment(parser=vars(args))
```

```
# experiment logic
```

Tools for Experiment Tracking and Versioning

Along with your code and your runtime parameters, you also need to version your data. Remember: your data reflects a snapshot of the world at the time when the data was gathered, and, of course, the world changes. If you're adding new data, purging old data, or cleaning data, it will change the results of your experiments. So just like when you make changes in your code, your model, or your hyperparameters, you need to track versions of your data. You might also change your feature vector as you experiment to add, delete, or change features. That needs to be versioned!

Fortunately, there are good tools for data versioning, including the following:

ML Metadata

Abbreviated MLMD, this is a library for recording and retrieving metadata associated with ML developer and data scientist workflows, including datasets. MLMD is an integral part of TensorFlow Extended (TFX), but it's designed so that it can also be used independently.

Artifacts

From Weights & Biases, this includes dataset versioning with deduplication, model tracking, and model lineage.

Neptune

This includes data versioning, experiment tracking, and a model registry.

Pachyderm

While you experiment in a separate branch of your repo, Pachyderm lets you continuously update the data in your master branch.

Delta Lake

From Databricks, this runs on top of your existing data lake and provides data versioning, including rollbacks and full historical audit trails.

Git LFS

An extension to Git, this replaces large files such as audio samples, videos, datasets, and graphics with text pointers inside Git.

lakeFS

This is an open source platform that provides a Git-like branching and committing model that scales to petabytes of data.

DVC

This is an open source version control system for ML projects that runs on top of Git.

When working in ML, you are constantly experimenting. Very quickly, it becomes vital to be able to compare the results of different experiments, but looking across lots of experiments at once can be confusing at first. As you gain experience with the tools, you'll get more comfortable, and it will be easier to focus on what you're looking for. It's a good idea to log everything you're experimenting with, tag experiments with a few consistent tags that are meaningful to you, and add notes. Developing these habits can keep things much more organized and help you collaborate with your team.

TensorBoard

TensorBoard is an amazing tool for analyzing your training, which makes it very useful for understanding your experiments. One of the many things that you can do with TensorBoard is to log metrics. Here is the code to log a confusion matrix at the end of every epoch:

```
logdir = "logs/image/" + datetime.now().strftime(
tensorboard_callback = keras.callbacks.TensorBoar
                                                log_di
cm_callback = keras.callbacks.LambdaCallback(
                                on_epoch_end=log_confu
model.fit(... callbacks=[tensorboard_callback, cm
```

Figure 15-1 shows the display of a confusion matrix in TensorBoard. These kinds of visual representations of metrics are often much more meaningful than just the data itself.

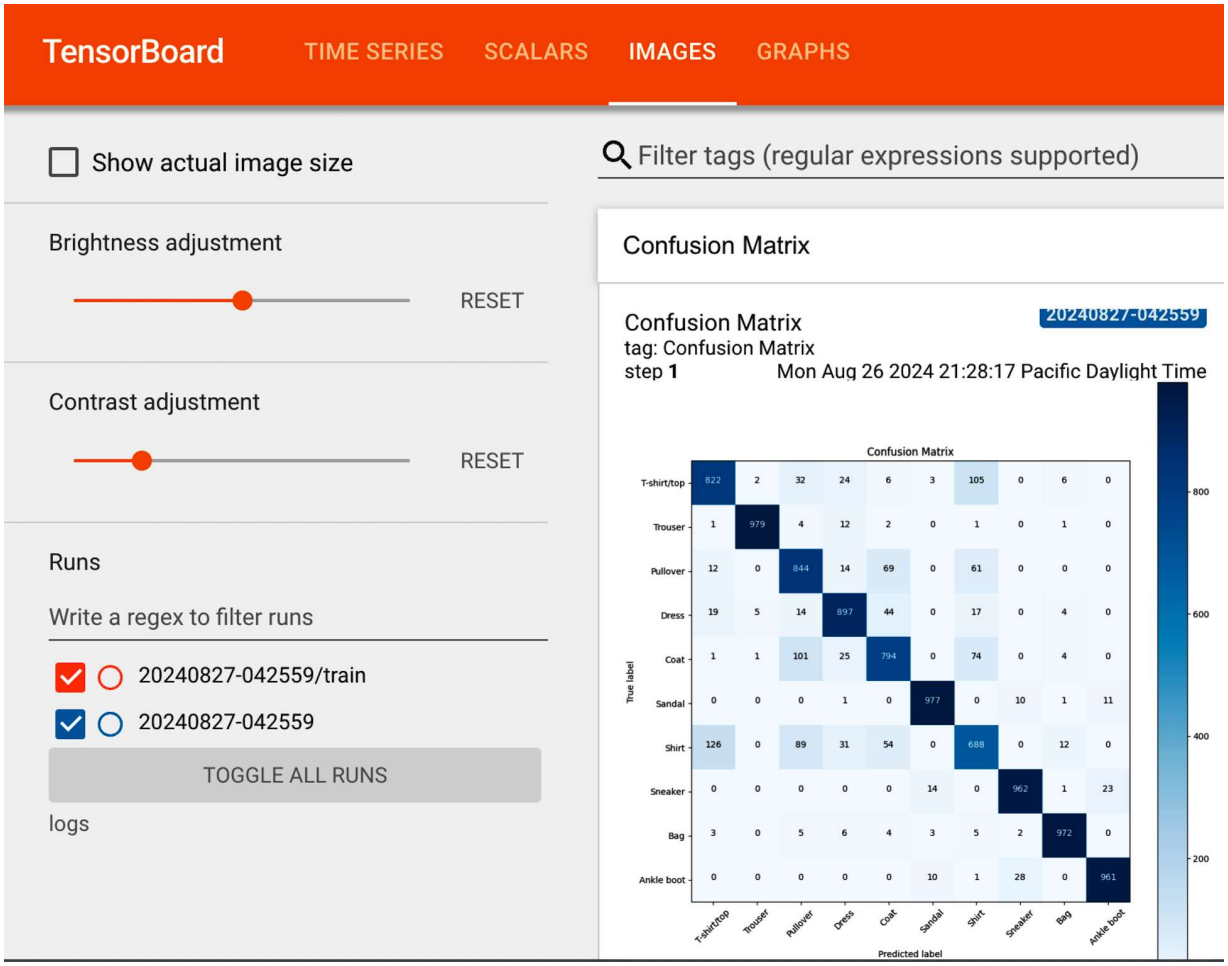

Figure 15-1. Displaying a confusion matrix in TensorBoard

Well-designed data visualizations give you a clear idea of how your model is doing, in this case, by examining a confusion matrix. By default, the dashboard displays the image summary for the last logged step or epoch. You can use the slider to view earlier confusion matrices. Notice how the matrix changes significantly as training progresses, with darker squares coalescing along the diagonal and the rest of the matrix tending toward 0 and white. This means your classifier is improving as training progresses. The ability to visualize the results as the model is training, and not just when training is complete, can also give you insights into your experiments.

Tools for organizing experiment results

As you continue to experiment, you'll be looking at each result as it becomes available and starting to compare results. Organizing your experimental results from the start is important to help you understand your own work when you revisit it later, and help your team understand it as well. You'll want to make sure it is easy to share and is accessible so that you and your team can collaborate, especially when working on larger projects. Tagging each experiment and adding your notes will help both you and your team, and it will help avoid having to run experiments more than once.

Tooling that enables sharing can really help. For example, you can use the experiment management tool provided by Neptune AI to send a link that shares a comparison of experiments. This makes it easy for you and your team to track and review progress, discuss problems, and inspire new ideas.

First, like many infrastructure decisions, there are significant advantages to using a managed service, including security, privacy, and compliance. But one of the most important features is having a persistent, shareable link to your dashboards that you can share with your team and not have to worry about setting it up and maintaining it. Having a searchable list of all the experiments in a project can also be incredibly useful. Tools such as Vertex TensorBoard (or similar cloud-based tools) are a big help and a huge improvement over spreadsheets and notes.

However, you can take your ML projects to the next level with creative iterations. In every project, there is a phase where a business specification is created that usually includes a schedule, budget, and the goals of the project. The goals are usually a set of key performance indicators (KPIs), business metrics, or if you are incredibly lucky, actual ML metrics. You and your team should choose what you think might be achievable business goals that align with the project, and start by defining a baseline approach. Implement your baseline, and evaluate it to get your first set of metrics.

Often, you'll learn a surprising amount from those first baseline results. They may be close to meeting your goals, which tells you this is likely to be an easy problem, or your results may be so far off that you'll start to wonder about the strength of the predictive signal in your data, and start considering more complex modeling approaches.

There is a tendency to focus on modeling metrics, and unfortunately much of the tooling also has that focus, but it's important to remember that since you're doing production ML, you primarily need to meet your business goals for latency, cost, fairness, privacy, General Data Protection Regulations (GDPR), and so forth. Focusing on ML metrics can sometimes distract you from those business goals.

Introduction to MLOps

Almost everything we discuss in this book can be considered MLOps in a very broad sense. But now let’s take a closer look at MLOps in a narrower sense and develop an understanding of different levels of maturity of MLOps processes and infrastructure.

Data Scientists Versus Software Engineers

First, let’s understand the two key roles within a typical ML engineering team: data scientist and software engineer. Thinking about these roles will help you understand why production ML makes it very valuable for data scientists to evolve into domain experts who can both develop predictive models and build production ML solutions. You’ll also learn how AI components are parts of larger systems and explore some of the challenges in engineering an AI-enabled system.

Thinking first about data scientists, especially those coming from a research or academic background, we can make some broad generalizations about what they do. They often work on fixed datasets that are provided to them, and they focus on optimizing model metrics such as accuracy. They tend to spend much of their time prototyping in notebooks. Their training usually makes them experts in modeling and feature engineering, while model size, cost, latency, and fairness are often not a central focus of their work.

Software engineers, however, tend to be much more focused on building products, so concerns such as cost, performance, stability, scalability, maintainability, and schedule are much more important to them. They identify strongly with customer satisfaction and recognize infrastructure needs such as scalability. They have a strong focus on quality, testing, and detecting and mitigating errors, and they are keenly aware of the need for security, safety, and fairness. They also, however, tend to view their work product as basically static, with changes being primarily the result of bug fixes or new features. Changes in the data as the world around them changes are not typically a primary concern when simply doing software development.

ML Engineers

In between pure data scientists and pure software engineers is a somewhat newer profession, that of ML engineer. An ML engineer combines most of the depth of a data scientist in modeling, feature engineering, and statistical approaches with a software engineer’s strong understanding of cost, performance, stability, scalability, maintainability, and schedule. ML engineers are often not as deep in either specialization as pure data scientists and software engineers are, but their ability to combine the two disciplines makes them extremely valuable members of a development team.

ML in Products and Services

ML and AI are quickly becoming critical for more and more businesses, creating whole new categories of products and services. Currently, the ingredients for applying ML have already been made accessible with large datasets, inexpensive on-demand compute resources, and increasingly powerful accelerators for ML such as GPUs and TPUs on several cloud platforms like AWS, Azure, and Google Cloud. There have been rapid advances in ML research in computer vision, natural language understanding, and recommendation systems, where there's an increased demand for applying ML to offer new capabilities.

Because of that, investment in ML and AI has been soaring and is likely to only increase. All of this drives an evolution of product-focused engineering practices for ML, which is the basis for the development of MLOps.

MLOps

Just as software engineering evolved with the creation of DevOps to be much more robust and well organized, ML is evolving with the creation of MLOps.

DevOps is an engineering discipline that focuses on deploying and managing software systems in production. It was developed over decades of experience and learning in the software development industry. Some of the

potential benefits that it offers include reducing development cycles, increasing deployment velocity, and ensuring dependable releases of high-quality software.

Like DevOps, MLOps is an ML engineering culture and practice that aims at unifying ML system development (or Dev) and ML system operation (Ops). Unlike DevOps, ML systems present unique challenges to core DevOps principles, including the following:

- Continuous integration, which for ML means you do not just test and validate code or components, but also do the same for data, schemas, and models
- Continuous delivery, which isn't just about deploying a single piece of software or a service, but is a system, or more precisely an ML pipeline, that deploys a model to a prediction service automatically

As ML emerges from research, disciplines such as software engineering, DevOps, and ML need to converge, forming MLOps. With that comes the need to employ novel DevOps automation techniques dedicated for training and monitoring ML models. That includes continuous training, a new property that is unique to ML systems, which automatically retrains models for both testing and serving.

And once you have models in production, it's important to catch errors and monitor inference data and performance metrics with continuous

monitoring. This part is similar to many pure software deployments, which often include monitoring with dashboards and other tooling.

Figure 15-2 shows the major phases in the lifecycle of an ML solution.

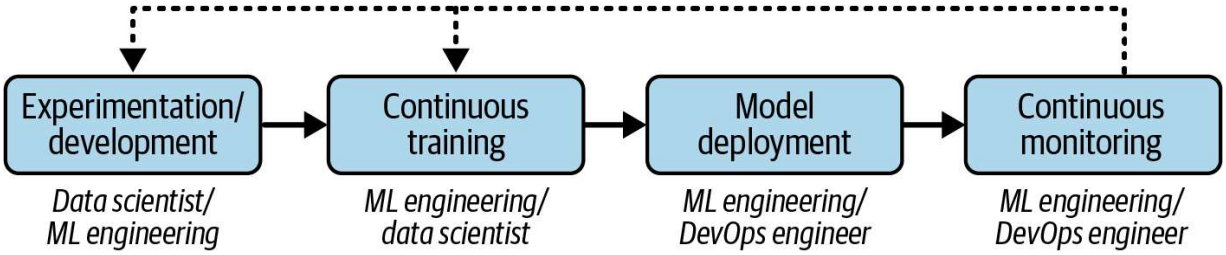

Figure 15-2. MLOps lifecycle (source: Salama et al., 2021)

Usually as a data scientist or ML engineer you start by shaping data and developing an ML model, and you continue by experimenting until you get results that meet your goals. After that, you typically go ahead and set up pipelines for continuous training, unless you already used a pipeline structure for your experimenting and model development, which we would encourage you to consider. Next, you turn to model deployment, which involves more of the operations and infrastructure aspects of your production environment and processes, and then continuous monitoring of your model, systems, and the data from your incoming requests.

The data from those incoming requests will become the basis for further experimentation and continuous training. So as you go from continuous training to model deployment, the tasks evolve into something that traditionally a DevOps engineer would be responsible for. That means you need a DevOps engineer who understands ML deployment and monitoring.

That need forms the basis for MLOps, which is a new practice for collaboration and communication between data scientists and operations professionals.

MLOps provides capabilities that will help you build, deploy, and manage ML models that are critical for ensuring the integrity of business processes. It also provides a consistent and reliable means to move models from development to production by managing the ML lifecycle.

Models also generally need to be iterated and versioned. To deal with an emerging set of requirements, the models change based on further training or real-world data that’s closer to the current reality. MLOps also includes creating versions of models as needed, and maintaining model version history. As the real world and its data continuously change, it’s critical that you manage model decay. With MLOps, you can ensure that by monitoring and managing the model results continuously, accuracy, performance, and other objectives and key requirements will be acceptable.

MLOps platforms also generally provide capabilities to audit compliance, access control, governance, testing and validation, and change and access logs. The logged information can include details related to access control, such as who is publishing models, why modifications are done, and when models were deployed or used in production.

You also need to secure your models from both attacks and unauthorized access. MLOps solutions can provide some functionality to protect models from being corrupted by infected data, being made unavailable by denial-of-service attacks, or being inappropriately accessed by unauthorized users.

Once you've made sure your models are secure, trustable, and good to go, it's often a good practice to establish a central repository where they can be easily discovered by your team. MLOps can include that by providing model catalogs for models produced, and a searchable model marketplace. These model discovery solutions should provide information to track the data origination, significance, model architecture and history, and other metadata for a particular model.

MLOps Methodology

Let's look at how MLOps processes evolve and mature as teams become more established and sophisticated.

MLOps Level 0

Fundamentally, the level of automation of the data, modeling, deployment, and monitoring systems determines the maturity of the MLOps process. As the maturity increases, both the reliability and velocity of training and deployment increase.

The objective of an MLOps team is to design and operate automated processes for training and deploying ML models, including robust and comprehensive monitoring. Ideally this means automating the entire ML workflow with as little manual intervention as possible. Triggers for automated model training and deployment can be calendar events, messaging, or monitoring events, as well as changes in data, model training code, and application code, or detected model decay.

Often teams will include data scientists, researchers, and ML engineers who can build state-of-the-art models, but their process for building and deploying models is completely manual. This approach defines level 0, as shown in Figure 15-3. Every step is manual, including data analysis, data preparation, model training, and validation. It requires manual execution of each step and manual transition from one step to another.

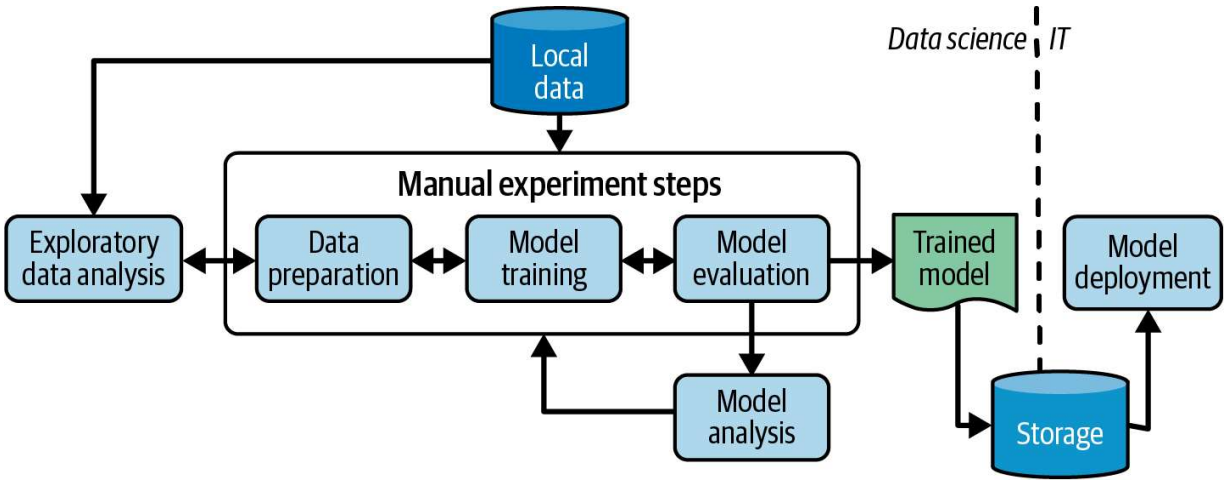

Figure 15-3. MLOps level 0 (source: Salama et al., 2021)

In a level 0 MLOps process, there is a disconnect between the ML research and operations teams. Among other things, this opens the door for potential training–serving skew. For example, let's assume data scientists hand over a trained model to the engineering team to deploy on their infrastructure for serving or batch prediction. This form of manual handoff could include putting the trained model in a filesystem somewhere, checking the model object into a code repository, or uploading it to a model registry. Then, engineers who deploy the model need to make the required input features available in production, potentially for low-latency serving, which can lead to training–serving skew.

A level 0 process assumes that your models don't change frequently. New versions of models are probably only deployed a couple of times per year. Because of that, continuous integration (CI), and often even unit testing, is totally ignored. Instead, testing is often done manually. The scripts and notebooks that implement the experiment steps are source controlled, and they produce artifacts such as trained models, evaluation metrics, and visualizations. Since there aren't many model versions that need deployments, continuous deployment (CD) isn't considered.

A level 0 process focuses on deploying models, rather than deploying the entire ML system. It often lacks any monitoring to detect model performance degradation and other model behavioral drifts.

MLOps level 0 is common in many startups and small teams. This manual, data scientist–driven process might be sufficient when models are rarely changed or retrained. Over time, teams often discover too late that their models deliver below expectations. Their models don't adapt to change and can fail unexpectedly.

Fixing these problems requires active performance monitoring. Actively monitoring your model lets you detect performance degradation and model decay. It acts as a cue that it's time for new experimentation and/or retraining of the model on new data. This might include continuously adapting your models to the latest trends.

To meet these requirements you need to retrain your production models with the most recent data as often as necessary to capture the evolving and emerging patterns. For example, if you're using a recommender for fashion products, it should adapt to the latest fashion trends—which can change quickly. That requires you to have new data and to label it somehow, and at level 0 those are usually manual processes also.

MLOps Level 1

MLOps level 1, shown in Figure 15-4, introduces full pipeline automation.

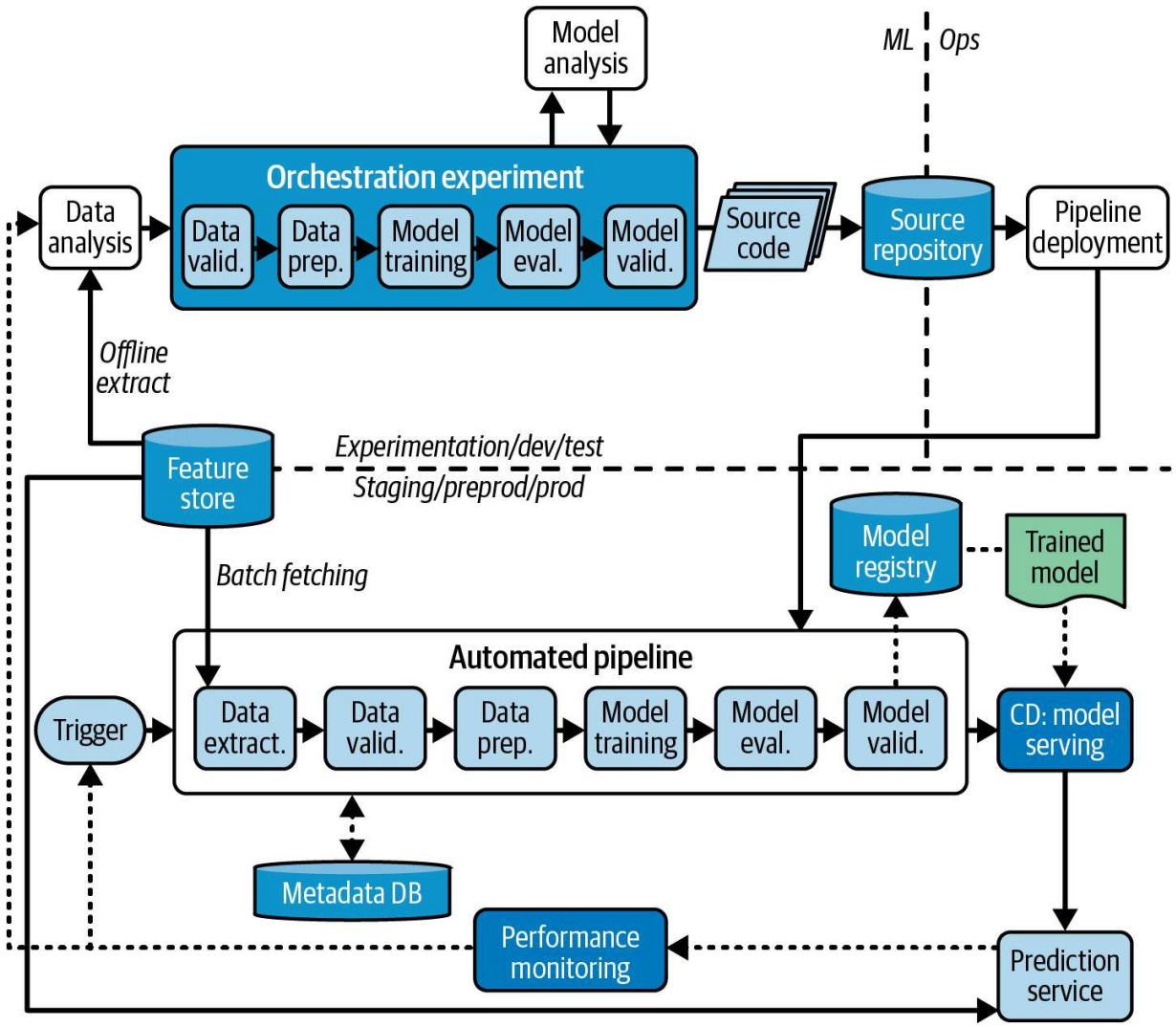

Figure 15-4. MLOps level 1 (source: Salama et al., 2021)

Automation for continuous training of the model is a primary goal of level 1. This enables you to implement CD of trained models to your server infrastructure. This requires that you implement automated data and model validation steps to the pipeline, pipeline triggers, and metadata management, in order to use new data to retrain models.

Level 1 implements repeatable training in your ML workflows. Notice here that the transition between steps is automated through orchestration. That enables you to rapidly iterate on your experiments, and it makes it easier to move the whole pipeline to production.

Now let's expand this out quite a bit to include the different environments—development, test, staging, preproduction, and production. Note that the architecture shown in Figure 15-7 (see "Components of an Orchestrated Workflow"), is typical, but different teams will implement this differently depending on their needs and infrastructure choices. In this architecture, the use of live pipeline triggers enables models to be automatically retrained using new data. The same pipeline architecture is used in both the development or experiment environment and the preproduction and production environments, which is a key aspect of an MLOps practice.

Components of ML pipelines need to be reusable, composable, and, in most cases, sharable across pipelines. Therefore, while the exploratory data analysis code can still live in notebooks, the source code for components must be modularized. In addition, components should ideally be containerized. You do this in order to decouple the execution environment from the custom code runtime. This also makes code reproducible between development and production environments. This essentially isolates each component in the pipeline, making them their own version of the runtime environment, which can potentially have different languages and libraries. Note that if the exploratory data analysis is done using production

components and a production-style pipeline, it greatly simplifies the transition of that code to production.

In production, an ML pipeline should continuously deliver new models that are trained on new data. Note that “continuously” means this happens in an automated process, in which new models might be delivered on a schedule or based on a trigger. The model deployment step is automated, which delivers the trained and validated model for use by a prediction service for online or batch predictions. In level 0, you simply deployed a trained model to production. You deploy a whole training pipeline in level 1, which automatically and recurrently runs to serve the trained model.

When you deploy your pipeline to production, it includes one or more of the triggers to automatically execute the pipeline. To train the next version of your model the pipeline needs new data. So, automated data validation and model validation steps are also required in a production pipeline.

Data validation is necessary before model training to decide whether you should retrain the model or stop the execution of the pipeline. This decision is automatically made based on whether or not the data is deemed valid. For example, data schema skews are considered anomalies in the input data, which means the components of your pipeline, including data processing and model training, would otherwise receive data that doesn’t comply with the expected schema. In this case, you should stop the pipeline and raise a notification so that the team can investigate. The team might release a fix or

an update to the pipeline to handle these changes in the schema. Schema skews include receiving unexpected features, not receiving all the expected features, or receiving features with unexpected values. Then there are data value skews, which are significant changes in the statistical properties of data, which require triggering a retraining of the model to capture these changes.

Model validation is another step that runs after you successfully train the model, given the new data. Here, you evaluate and validate the model before it's promoted to production. This offline model validation step may involve first producing evaluation metric values using the trained model on a test dataset to assess the model's predictive quality. The next step would be to compare the evaluation metric values produced by your newly trained model to the current model; for example, the current production model, a baseline model, or other model that meets your business requirements. Here, you make sure the new model performs better than the current model before promoting it to production. Also, you ensure that the performance of the model is consistent on various segments of the data. For example, your newly trained customer churn model might produce an overall better predictive accuracy compared to the previous model, but the accuracy values per customer region might have a large variance.

Finally, infrastructure compatibility and consistency with the prediction service API are some other factors that you need to consider before deploying your models. In other words, will the new model actually run on

the current infrastructure? In addition to offline model validation, a newly deployed model undergoes online model validation in either a canary deployment or an A/B testing setup during the transition to serving prediction for the online traffic.

An optional additional component for level 1 MLOps is a feature store. A *feature store* is a centralized repository where you standardize the definition, storage, and access of features for training and serving. Ideally a feature store will provide an API for both high-throughput batch serving and low-latency real-time serving for the feature values, as well as support for both training and serving workloads. A feature store helps you in many ways. First of all, it lets you discover and reuse available feature sets instead of re-creating the same or similar ones, avoiding having similar features that have different definitions by maintaining features and their related metadata.

Moreover, you can potentially serve up-to-date feature values from the feature store and avoid training–serving skew by using the feature store as the data source for experimentation, continuous training, and online serving. This approach makes sure the features used for training are the same ones used during serving. For example, for experimentation, data scientists can get an offline extract from the feature store to run their experiments. For continuous training, the automated training pipeline can fetch a batch of the up-to-date feature values of the dataset. For online prediction, the prediction service can fetch feature values, such as customer

demographic features, product features, and current session aggregation features.

Another key component of level 1 is the metadata store, where information about each execution of the pipeline is recorded in order to help with data and artifact lineage, reproducibility, and comparisons. This also makes errors and anomalies easier to debug. Each time you execute the pipeline, the metadata store tracks information such as which pipeline and component versions were executed; the start and end dates, times, and how long the pipeline took to complete each step; the input and output artifacts from each step; and more. This enables you to use the artifacts produced by each step of the pipeline, such as the prepared data, validation anomalies, and computed statistics, to seamlessly resume execution in case of an interruption. Tracking these intermediate outputs helps you resume the pipeline from the most recent step if the pipeline stopped due to a failed step, without having to restart the pipeline as a whole.

MLOps Level 2

At the current stage of the development of MLOps best practices, level 2 is still somewhat speculative. Figure 15-5 presents one of the current architectures, which is focused on enabling rapid and reliable update of the pipelines in production. This requires a robust automated CI/CD system to enable your data scientists and ML engineers to rapidly explore new ideas

and experiment. By implementing in a pipeline, they can automatically build, test, and deploy to the target environment.

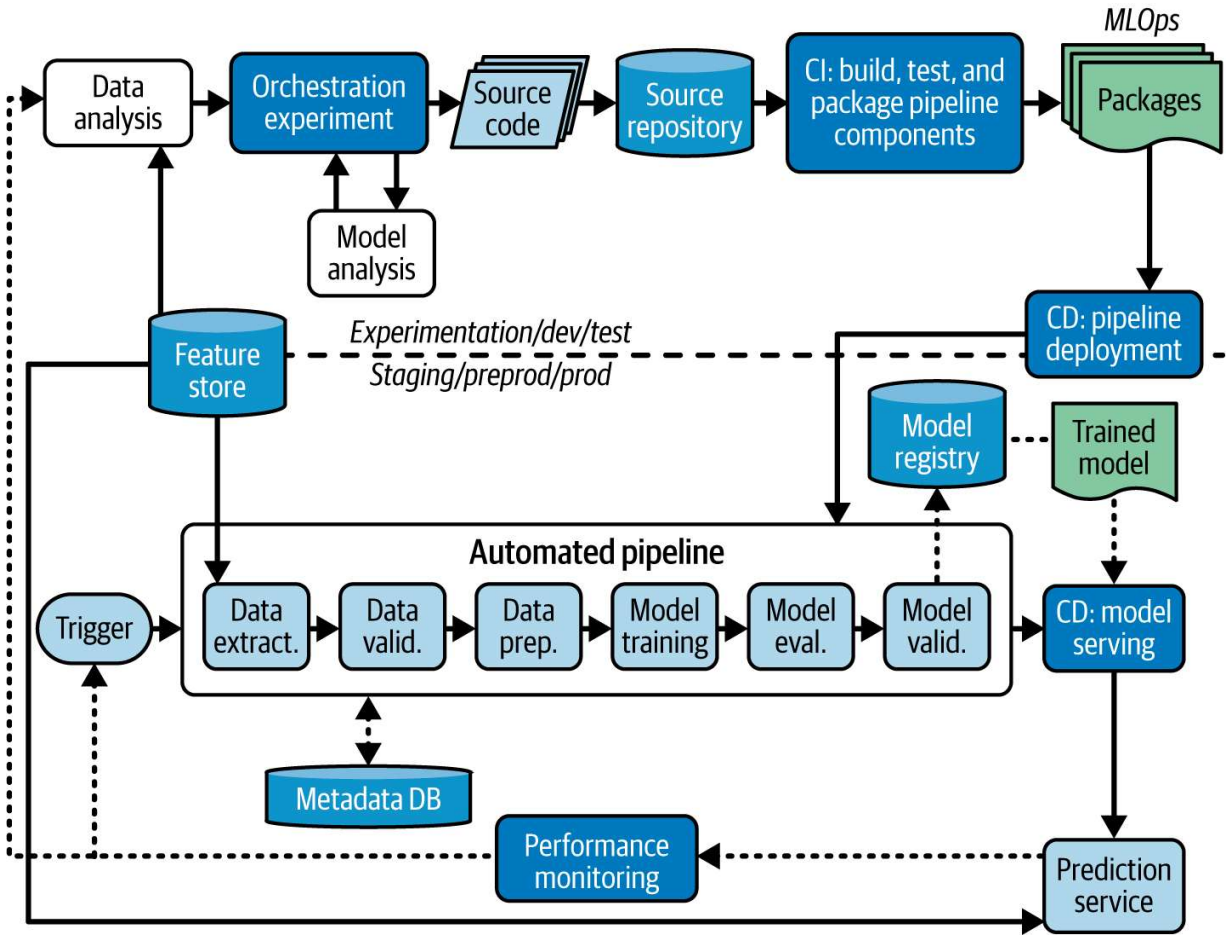

Figure 15-5. MLOps level 2 (source: Salama et al., 2021)

This MLOps setup includes components such as source code control, test and build services, deployment services, a model registry, a feature store, a metadata store, and a pipeline orchestrator. Since this is a lot to take in, let's look at the different stages of the ML CI/CD pipeline in a simplified and more digestible form, as shown in Figure 15-6.

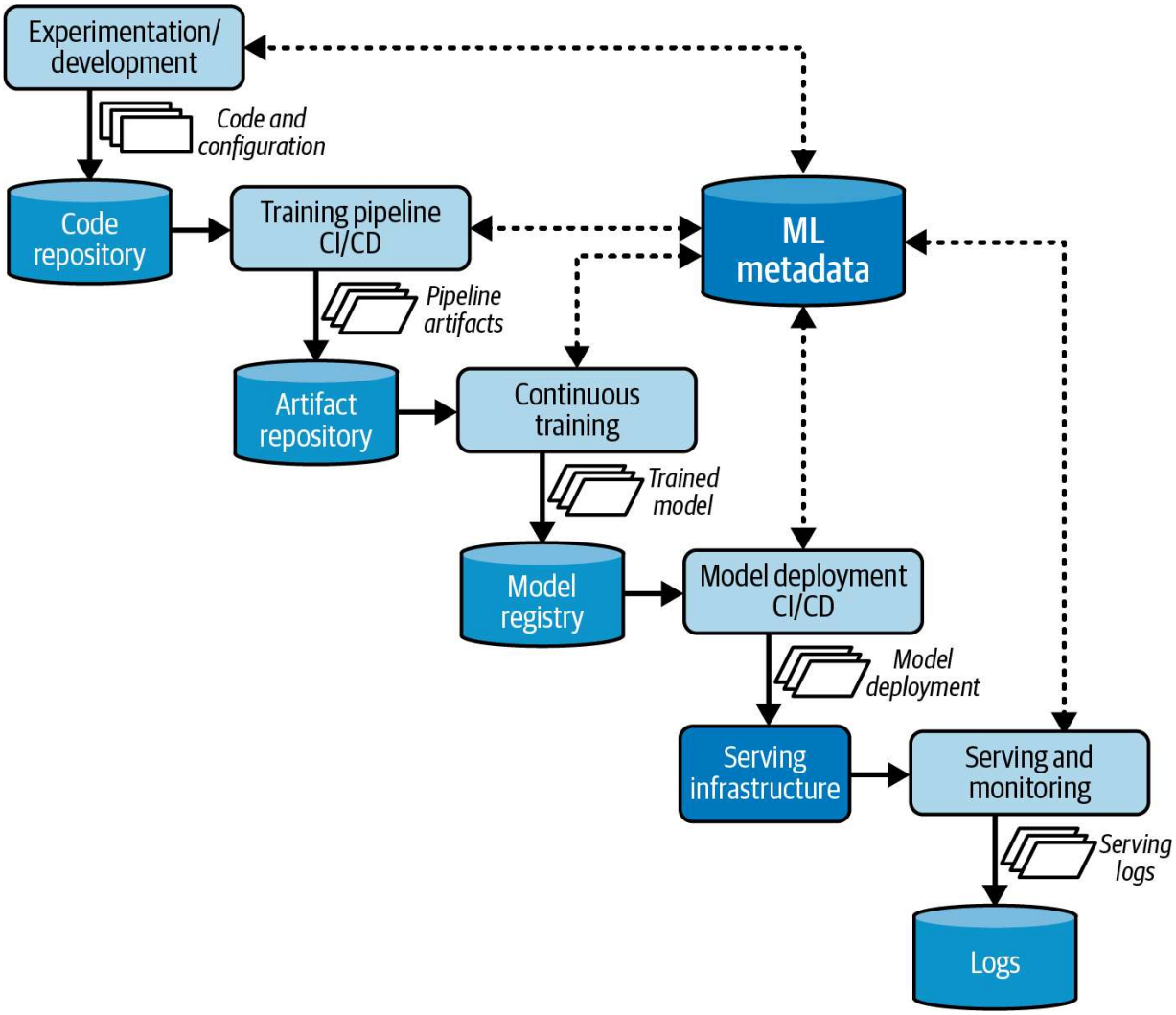

Figure 15-6. A simplified view of level 2 (source: Salama et al., 2021)

It begins with experimentation and development. This is where you iteratively try out new algorithms, new modeling, and/or new data, and orchestrate the experiment steps.

Next comes the CI/CD stage for the training pipeline itself. Here you build the source code and run various tests. The outputs of this stage are pipeline entities such as software packages, executables, and artifacts to be deployed in a later stage.

Next, the models are trained, including validation of the data and the model performance by running the pipeline based on a schedule or in response to a trigger. Once the model has been trained, the goal of the pipeline is to now deploy it using continuous delivery. This includes serving the trained model as a prediction service.

Finally, once all the models have been trained and deployed, it’s the role of the monitoring service to collect statistics on model performance based on live data. The output of this stage is the data collected in logs from the operation of the serving infrastructure, including the prediction request data, which will be used to form a new dataset to retrain your model.

Components of an Orchestrated Workflow

One of the key parts of an MLOps infrastructure is the training pipeline. Let’s look now at developing training pipelines using TFX, including ways to adapt your pipelines to meet your needs with custom components.

TFX is an open source framework that you can use to create ML pipelines. TFX enables you to implement your model training workflow in a wide variety of execution environments, including containerized environments such as Kubernetes. TFX pipelines organize your workflow into a sequence of components, where each component performs a step in your ML workflow.

TFX standard components provide proven functionality to help you get started building an ML workflow easily. You can also include custom components in your workflow, including creating components that run in containers and can use any language or library you can run in a container, such as performing data analysis using R. Custom components let you extend your ML workflow by enabling you to create components that are tailored to meet your needs, such as:

- Data augmentation, upsampling, or downsampling
- Anomaly detection
- Interfacing with external systems such as dashboards for alerting and monitoring

Figure 15-7 shows what a starter, or "Hello World," TFX pipeline typically looks like. The boxes show standard components that come with TFX out of the box. (This "Hello World" pipeline could just as easily show custom components that you created.) Most of these components are a training pipeline, but the two components on the bottom row, ExampleGen and Bulk Inference, are an inference pipeline for doing batch inference.

So, by mixing standard components and custom components, you can build an ML workflow that meets your needs while taking advantage of the best practices built into the TFX standard components. As a developer, you can often work with a high-level API, but it's useful to know the fundamentals of a component's anatomy. There are three main pieces:

- A component specification, which defines the component's input and output contract. This contract specifies the component's input and output artifacts, and the parameters that are used for the component execution.
- A component `Executor` class, which provides the implementation for the work performed by the component. It's the main code for a component, and typically this is where your code runs.
- A component class, which combines the component specification with the Executor for use in a TFX pipeline. It also includes the Driver and Publisher portions of the component.

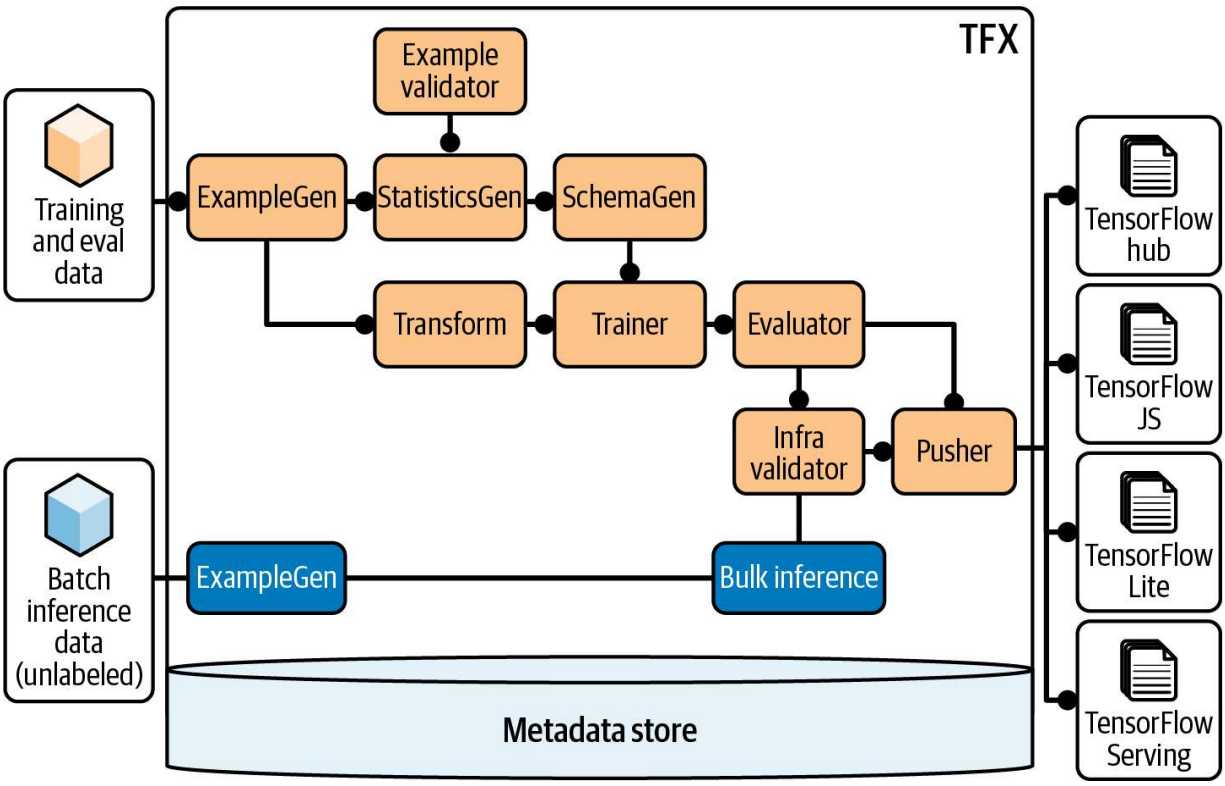

Figure 15-7. The "Hello World" of TFX

Note that this is the implementation style used by the TFX standard components and “full custom” style components, but there are two other styles for creating custom components, which we will discuss next.

When a pipeline runs a TFX component, the component is executed in three phases, as shown in Figure 15-8. First, the Driver uses the component specification to retrieve the required artifacts from the Metadata Store and pass them into the component. Next, the Executor performs the component’s work. Finally, the Publisher uses the component specification and the results from the Executor to store the component’s outputs in the Metadata Store.

NOTE

Most custom component implementations do not require you to customize the Driver or the Publisher. Typically, modifications to the Driver and Publisher should be necessary only if you want to change the interaction between your pipeline’s components and the Metadata Store, which is rare. If you only want to change the inputs, outputs, or parameters for your component, you only need to modify the component specification.

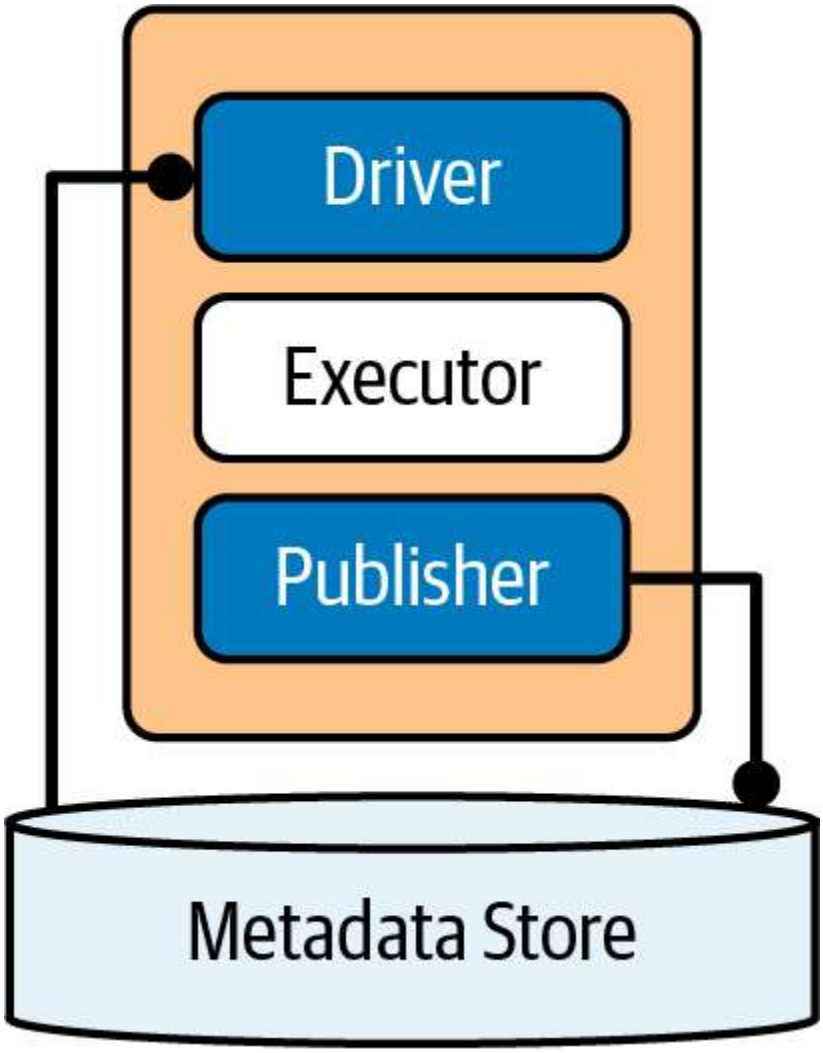

Figure 15-8. Component execution: Driver, Executor, Publisher, and the Metadata Store

Three Types of Custom Components

There are three types of custom components:

- Python function–based components: Are the easiest to build, easier than container-based components or fully custom components. They only

require a Python function for the Executor, with a decorator and annotations.

- Container-based components: Provide the flexibility to integrate code written in any language into your pipeline, by running your component in a Docker container. To create a container-based component, you create a component definition that is very similar to a Dockerfile and call a wrapper function to instantiate it.
- Fully custom components: Let you build components by defining the component specification, Executor, and component interface classes. This approach also lets you reuse and extend a standard component to meet your needs.

Python Function–Based Components

The Python function–based component style makes it easy for you to create TFX custom components by saving you the effort of defining a component specification class, Executor class, and component interface class. In this style, you write a function that is decorated and annotated with type hints. The type hints describe the input artifacts, output artifacts, and parameters of your component. Writing a custom component for simple model validation in this style is very straightforward:

```
@component
def MyValidationComponent(
    model: InputArtifact[Model],
```

```
    blessing: OutputArtifact[Model],
    accuracy_threshold: Parameter[int] = 10,
) -> OutputDict(accuracy=float):
    '''My simple customer model validation compor
    accuracy = evaluate_model(model)
    if accuracy >= accuracy_threshold:
        write_output_blessing(blessing)
    return {'accuracy': accuracy}
```

The component specification is defined in the Python function's arguments using type annotations that describe whether an argument is an input artifact, output artifact, or parameter. The function body defines the component's Executor. The component interface is defined by adding the `@component` decorator to your function. By decorating your function with the `@component` decorator and defining the function arguments with type annotations, you can create a component without the complexity of building a component specification, an Executor, and a component interface.

Container-Based Components

Container-based components are backed by containerized command-line programs, and creating one is in some ways similar to creating a Dockerfile. To create one, specify the necessary parameter values and call the

`create_container_component` function, passing the component definition, including the component name, inputs, outputs, and parameters:

```
from tfx.dsl.component.experimental import conta
from tfx.dsl.component.experimental import placeh
from tfx.types import standard_artifacts

grep_component = container_component.create_conta
    name='FilterWithGrep',
    inputs={'text': standard_artifacts.ExternalAr
    outputs={'filtered_text': standard_artifacts
    parameters={'pattern': str},
    ...
)
```

There are also other parts of the configuration, such as the image tag, which specifies the Docker image that will be used to create the container. For the body of the component, you have the command parameter that specifies the container entrypoint command line. As with Dockerfiles, this isn't executed within a shell unless you specify that in your command line. The command line can use placeholder objects that are replaced at compilation time with the input, output, or parameter values:

```
grep_component = container_component.create_conta
    ...
```

```
        image='google/cloud-sdk:278.0.0',
        command=[
            'sh', '-exc',
            '''
            ...
            ''',
            '--pattern', placeholders.placeholders.In
            '--text', placeholders.placeholders.Input
            '--filtered-text',
            placeholders.placeholders.OutputUriPlaceh
        ],
    )
```

The placeholder objects can be imported from `tfx.dsl.component.experimental.placeholders`. In this example, the component code uses `gsutil` to upload the data to Google Cloud Storage, so the container image needs to have `gsutil` installed and configured. This approach is more complex than building a Python function–based component, since it requires packaging your code as a container image. This approach is most suitable for including non-Python code in your pipeline or for building Python components with complex runtime environments or dependencies.

Fully Custom Components

This style lets you build components by directly defining the component specification, Executor class, and component class. This approach also lets you reuse and extend a standard component or other preexisting component to meet your needs. For example, if an existing component is defined with the same inputs and outputs as the custom component that you're developing, you can simply override the Executor class of the existing component. This means you can reuse a component specification and implement a new Executor that derives from an existing component. In this way, you reuse functionality built into existing components and implement only the functionality that is required.

The primary use of this component style is to extend existing components. Otherwise, if you don't need a containerized component, you should probably use the Python function style instead. However, developing a good understanding of this style will help you better understand all TFX components, so let's take a closer look at how to create a fully custom component.

Developing a fully custom component first requires defining a `ComponentSpec`, which contains a set of input and output artifact specifications for the new component. You must also define any non-artifact execution parameters that are needed for the new component:

```
class HelloComponentSpec(types.ComponentSpec):
 """ComponentSpec for Custom TFX Hello World Comp
 INPUTS = {
     # This will be a dictionary with input artif
     'input_data': ChannelParameter(type=standard
 }
 OUTPUTS = {
     # This will be a dictionary which this compo
     'output_data': ChannelParameter(type=standar
 }
   PARAMETERS = {
     # These are parameters that will be passed i
     # create an instance of this component.
     'name': ExecutionParameter(type=Text),
 }
```

There are three main parts of a component specification: the inputs, outputs, and parameters. Inputs and outputs are wrapped in *channels*, essentially dictionaries of typed parameters for the input and output artifacts. A parameter is a dictionary of additional `ExecutionParameter` items, which are passed into the Executor and are not metadata artifacts.

Next, you need an Executor class. Basically, this is a subclass of `base_executor.BaseExecutor`, with its `Do` function overridden. In the `Do` function, the arguments `input_dict`, `output_dict`, and `exec_properties` are passed in, which map to the `INPUTS`,

`OUTPUTS`, and `PARAMETERS` that are defined in `ComponentSpec`. For `exec_properties`, the values can be fetched directly through a dictionary lookup:

```
  class Executor(base_executor.BaseExecutor):
   """Executor for HelloComponent."""
   def Do(self, input_dict: Dict[Text, List[types.A
          output_dict: Dict[Text, List[types.Artifa
          exec_properties: Dict[Text, Any]) -> None
     ...
```

Continuing with implementing the Executor, for artifacts in the `input_dict` and `output_dict`, there are convenience functions available in the artifact utilities class of TFX that can be used to fetch an artifact's instance or its URI:

```
  class Executor(base_executor.BaseExecutor):
   """Executor for HelloComponent."""
   def Do(self, input_dict: Dict[Text, List[types.A
          output_dict: Dict[Text, List[types.Artifa
          exec_properties: Dict[Text, Any]) -> None
     ...
     split_to_instance = {}
     for artifact in input_dict['input_data']:
       for split in json.loads(artifact.split_names
         uri = artifact_utils.get_split_uri([artifa
```

```
        split_to_instance[split] = uri
    for split, instance in split_to_instance.items
      input_dir = instance
      output_dir = artifact_utils.get_split_uri(
          output_dict['output_data'], split)
      for filename in tf.io.gfile.listdir(input_di
        input_uri = os.path.join(input_dir, filena
        output_uri = os.path.join(output_dir, file
        io_utils.copy_file(src=input_uri, dst=outp
```

Now that the most complex part is complete, the next step is to assemble these pieces into a component class, to enable the component to be used in a pipeline. There are several steps. First, you need to make the component class a subclass of `base_component.BaseComponent`, or a different component if you're extending an existing component. Next, you assign class variables `SPEC_CLASS` and `EXECUTOR_SPEC` with the `ComponentSpec` and `Executor` classes, respectively, that you just defined:

```
from tfx.types import standard_artifacts
from hello_component import executor
class HelloComponent(base_component.BaseComponent
 """Custom TFX Hello World Component."""
 SPEC_CLASS = HelloComponentSpec
 EXECUTOR_SPEC = executor_spec.ExecutorClassSpec(
```

Next, we complete the fully custom component by implementing the `__init__` constructor, which will initialize the component. Here, you define the constructor function by using the arguments to the function to construct an instance of the `ComponentSpec` class and invoke the super function with that value, along with an optional name. When an instance of the component is created, type-checking logic in the `base_component.BaseComponent` class will be invoked to ensure that the arguments that were passed are compatible with the types defined in the `ComponentSpec` class:

```
class HelloComponent(base_component.BaseComponent
 """Custom TFX Hello World Component."""
 def __init__(self,
              input_data: types.Channel = None,
              output_data: types.Channel = None,
              name: Optional[Text] = None):
   if not output_data:
     examples_artifact = standard_artifacts.Examp
     examples_artifact.split_names = input_data.g
     output_data = channel_utils.as_channel([exam
   spec = HelloComponentSpec(input_data=input_dat
                             output_data=output_d
   super(HelloComponent, self).__init__(spec=spec
```

The last step is to plug the new custom component into a TFX pipeline. Besides adding an instance of the new component, you need to wire the upstream and downstream components to it. You can generally do this by referencing the outputs of the upstream component in the new component, and referencing the outputs of the new component in downstream components. Also, another thing to keep in mind is that you need to add the new component instance to the components list when constructing the pipeline:

```
def _create_pipeline():
 ...
 example_gen = CsvExampleGen(input_base=examples
 hello = component.HelloComponent(
     input_data=example_gen.outputs['examples'],
  statistics_gen = StatisticsGen(examples=hello.o
 ...
 return pipeline.Pipeline(
     ...
     components=[example_gen, hello, statistics_g
     ...
 )
```

TFX Deep Dive

After learning the basics of constructing a pipeline, in this section we will dive deeper into the architecture of TFX.

The TFX stack provides three components that decouple the authoring and execution of ML pipelines:

TFX SDK

A Python SDK used to *author* custom ML pipelines

Intermediate Representation (IR)

A portable serialized representation of a pipeline defined by the SDK

Runtime

A Python library that facilitates *executing* the pipeline IR using any generic orchestrator

TFX SDK

The TFX SDK is a Python library that provides everything necessary to arrive at a pipeline IR (see the next section). You use the TFX SDK to do the following:

Use or create components

The SDK provides access to off-the-shelf components (first and third party) and multiple options for the user to build their own custom components:

- Standard components: developed and supported by the TFX team for common ML tasks

- Custom components: developed and supported by each user for their own pipelines, mentioned in "Three Types of Custom Components"

Compose a pipeline

The SDK enables users to flexibly wire components together to compose a pipeline, leveraging advanced semantics, conditionals, and the exit handler.

Compile a pipeline to IR

The SDK provides a compiler that can be used to yield the pipeline IR with a single function call.

Intermediate Representation

The IR is a representation of the pipeline that is obtained by compiling an in-memory pipeline composed using the SDK into a protobuf message.

NOTE

Protocol buffer (protobuf) is a free and open source cross-platform data format used to serialize structured data. It is useful in developing programs that communicate with each other over a network or for storing data.

The TFX IR is a crucial abstraction that is at the heart of the portability and modularity of the TFX stack, enabling decoupling of pipeline authoring and execution. For end users, it enables better debugging and a more efficient support experience, while for platform developers, it provides a stable interface on which to build additional tooling and integrations.

Runtime

The TFX runtime can be used to turn any generic orchestrator (Kubeflow, Airflow) into an ML workflow execution engine. This runtime wraps each component in the pipeline as a schedulable unit and logs its execution in a Metadata Store to track the artifacts consumed and produced by it. This unique pattern enables key features in TFX, such as lineage tracking and data-driven orchestration. It is also the foundation from which to realize advanced pipeline topologies and other common ML needs.

Implementing an ML Pipeline Using TFX

Components

In addition to writing custom components, TFX also provides standard components to implement ML workflows. Let’s take a typical pipeline, shown in Figure 15-9, which requires the following tasks:

- Ingest data directly from a custom data source using a custom component.
- Calculate statistics for the training data using the StatisticsGen standard component.
- Create a data schema using the SchemaGen standard component.
- Check the training data for anomalies using the ExampleValidator standard component.
- Perform feature engineering on the dataset using the Transform standard component.
- Train a model using the Trainer standard component.
- Evaluate the trained model using the Evaluator standard component.
- If the model passes its evaluation, the pipeline adds the trained model to a queue for a custom deployment system using a custom component.

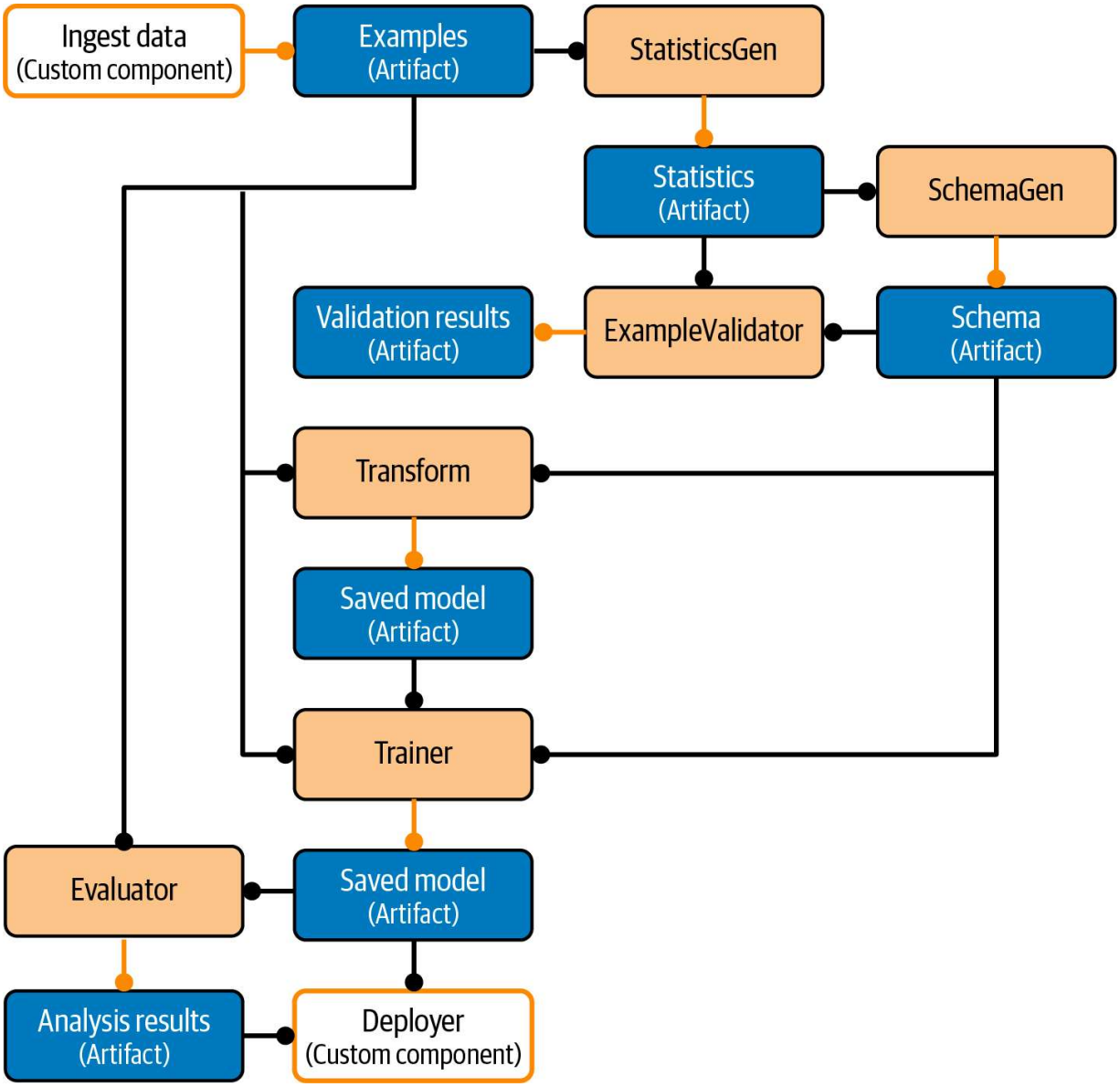

Figure 15-9. A typical TFX pipeline

Based on this analysis, an orchestrator executing this pipeline will run the following:

- The data ingestion and the StatisticsGen and SchemaGen component instances can be run sequentially.

- The ExampleValidator and Transform components can run in parallel since they share input artifact dependencies and do not depend on each other's output.
- After the Transform component is complete, the Trainer, Evaluator, and custom deployer component instances run sequentially.

For all the standard TFX components, check the TFX User Guide.

Advanced Features of TFX

There are some more advanced features and concepts in TFX and similar frameworks.

Component dependency

In TFX, components are chained together to form a pipeline. During a pipeline run, the orchestrator runs components according to their topological order in the pipeline. A component will only be triggered when all its upstream components finish. TFX considers several factors to calculate the execution order, and among them, component dependency is the main factor. There are two kinds of dependencies between components.

Data dependency

TFX figures out the data dependency automatically. No special declaration is needed. If an output artifact is consumed by another component as input,

a data dependency is automatically created. For example:

```
example_gen = ImportExampleGen(...)
statistics_gen = StatisticsGen(examples=example_g
```

Here, `statistics_gen` consumes an output artifact of `example_gen`, hence there is a data dependency between them. It means that `statistics_gen` must run after a successful execution of `example_gen`. (This is kind of self-evident because `statistics_gen` needs the output of `example_gen`.)

Task dependency

Sometimes there are some dependencies that are unknown to TFX, and in those cases, you need task dependency because TFX does not know about them. For example:

```
downstream_component.add_upstream_node(upstream_c
# Alternatively,
# upstream_component.add_downstream_node(downstre
```

In this case, there are two components in a TFX pipeline, and they do not necessarily share artifacts. If you want them to run sequentially, you need to

declare a task dependency using `.add_upstream_node()` or `.add_downstream_node()`; otherwise, TFX runs them in parallel.

Importer

Importer is a system node that *creates* an artifact from data that is specified with a URI (often a file) as a desired artifact type:

```
  hparams_importer = Importer(
      source_uri='...',
      artifact_type=HyperParameters).with_id('hpara
  trainer = Trainer(
      ...,
      hyperparameters=hparams_importer.outputs['res
  )
```

The output channel from `Importer` can include `additional_properties` or `additional_custom_properties` attributes, which instruct an `Importer` to attach such `properties` or `custom_properties` when creating an artifact:

```
  adhoc_examples_importer = Importer(...)
  adhoc_examples_importer.outputs['result'].additio
```

As `Importer` is a node of the pipeline, it should be included in the `Pipeline(components=[...])` when creating a `Pipeline` instance.

Conditional execution

A conditional is the `if` statement in a pipeline. Because TFX converts the user's Python code into IR, you cannot use a Python `if` statement to customize control flow based on a pipeline runtime result (like component output), since that result is not known when the IR is created. TFX offers a conditional domain-specific language (DSL) to support branching based on component output. To use it, put the components that need to be conditionally executed under a `with` block. For example:

```
from tfx.dsl.experimental.conditionals import co
evaluator = Evaluator(...)
# Run pusher if evaluator has blessed the model.
with conditional.Cond(evaluator.outputs['blessing
                      [0].custom_property('blesse
  pusher = ServomaticPusher(...)
pipeline = Pipeline(
  ...,
  # Even though pusher's execution may be skipped
  # in the components list of the pipeline.
```

```
    components=[..., evaluator, pusher]
  )
```

The preceding code snippet can be translated to “run pusher if evaluator has blessed the model.” The line `evaluator.outputs['blessing'].future()[0].custom_property('blessed') == 1` is a predicate that is evaluated to `True` or `False` at runtime. The components declared under the `with` block are triggered if the predicate evaluates to `True`, and skipped otherwise.

Multiple components can be put under one conditional block. Those components are either all executed or all skipped, depending on the evaluation result of the predicate.

Managing Model Versions

Now let’s turn to another important topic in MLOps, managing model versions. Let’s start by looking at why version control is so important and examining some of the challenges of versioning models.

In normal software development, especially with teams, organizations rely on version control software to help teams manage and control changes to their code. But imagine if you didn’t have that! How would you enable

multiple developers to stay in sync? How would you roll back to a previous working version when there are problems? How would you do continuous integration? Just like with software development, when you're developing models you have all of these needs and more.

Generating models is an iterative process. During development, you typically generate several models and compare one against the other to evaluate the performance of each model. Each model version may have different code, data, and configurations. You need to keep track of all of this to properly reproduce results. This is where model versioning is important. Versioning will improve collaboration at different levels, from individual developers to teams and all the way up to organizations.

Approaches to Versioning Models

So how should you version your models? First, let's think about how you version software.

A typical convention is that you version software with a combination of three numbers. These numbers are the major version, the minor version, and a patch number of the release. The major version usually increases when you make incompatible API changes or introduce a major feature or functionality. The minor version is increased when you add functionality in a backward-compatible manner or add a minor feature, and the patch

number is increased when you make backward-compatible bug fixes. So, can you use a similar approach for your models?

As of this writing, there is no uniform standard that is widely accepted across the industry to version models. Different companies have adopted their own conventions for versioning, and as a developer in their organization you need to understand how they version their models.

Versioning proposal

One possible approach to consider is simple to understand and is in line with normal software versioning.

Let’s use a combination of three numbers and denote these as the major, minor, and pipeline versions:

`MAJOR`

: Incompatibility in data or target variable.

`MINOR`

: Model performance is improved.

`PIPELINE`

: Pipeline of model training is changed.

The major version will increment when you have an incompatible data change, such as a schema change or target variable change, that can render

the model incompatible with its prior versions when it’s used for predictions. The minor version will increment when you believe you’ve improved or enhanced the model’s results. Finally, the pipeline version will correspond to an update in the training pipeline, but it need not improve or even change the model itself.

But this is only one of many possible ways to version models. Next, let’s look into some other styles of versioning that are sometimes used.

Arbitrary grouping

In this format, the developer decides how to group a set of models as different versions of the same model. A well-known product that uses this format is Google Cloud AI Prediction. A good practice while following arbitrary grouping is to make sure the models solve the same ML tasks or use cases. Note, however, that while arbitrary grouping may not account for change of architecture, algorithms, input feature vectors, and so on, it does offer a high degree of flexibility for the developer.

Black-box functional model

Another style of versioning is known as black-box functional modeling, in which you view a model as a black box that implements a function to map the inputs to the outputs, with a fixed set of training data. The version of the

model changes only when the model implementation changes. This means that if either inputs or outputs or both change, a new model is defined.

Pipeline execution versioning

The last style of versioning to look at is known as pipeline execution versioning. In this style, you define a new version with each successful run of the training pipeline. Models will be versioned regardless of changes to model architecture, input, or output. A notable product that uses this style of versioning is TFX.

Model Lineage

One way to test a versioning style is to ask, can you leverage a framework's capability to retrieve previously trained models? For an ML framework to retrieve older models, the framework has to be internally versioning the models through some versioning technique.

Different ML frameworks may use different techniques to retrieve previously trained models. One technique is by making use of model lineage. Model lineage is a set of relationships among the artifacts that resulted in the trained model. To build model artifacts, you have to be able to track the code that builds them, as well as the data (including preprocessing operations) the model was trained and tested with. ML orchestration frameworks such as TFX will store this model lineage for

many reasons, including re-creating different versions of the model when necessary. Note that model lineage usually only includes those artifacts and operations that were part of model training and evaluation. Post-training artifacts and operations are usually not part of lineage.

Model Registries

A model registry is a central repository for storing trained models. Model registries provide an API for managing trained models throughout the model development lifecycle, and they are essential in supporting model discovery, model understanding, and model reuse, including in large-scale environments with hundreds or thousands of models. As a result, model registries have become an integral part of many open source and commercial ML platforms.

Along with the models themselves, model registries often benefit from storing metadata. Some model registries provide storage for serialized model artifacts. To improve the model discoverability within the model registry, it’s important to store some free text annotations and other structured or searchable properties of the models. And to promote model lineage, registries sometimes include links to other ML metadata stores.

Model registries promote model search and discoverability within your organization, and they can help improve the understanding of the model among your team. They can also help enforce a set of approval guidelines

that need to be followed when uploading models, which can help improve governance. By sharing models with your team, you are improving the chances of collaboration among your coworkers. Model registries can also help streamline deployments, and they can even provide a platform for continuous evaluation and monitoring.

Continuous Integration and Continuous Deployment

In more mature MLOps processes, and where more than a few models need to be managed, it’s important to implement a robust deployment process. This is especially true when model predictions are served online as part of a user-facing application. As in software development, implementing continuous deployment also becomes important for ML.

Continuous Integration

First, before deploying you need to make sure your code works, which you should determine through comprehensive unit testing. This is automated with CI, which triggers whenever new code is committed or pushed to your source code repository. It mainly performs building, packaging, and testing for the components. The quality of the testing will be determined by the coverage and quality of your unit test suite. If all tests pass, it delivers the tested code and packages to a continuous delivery pipeline. Of course, it

requires that your code is written to be testable, which is generally the case with well-written modular code but can be an issue with code that is poorly structured.

Let’s look at the main two types of tests that are performed during continuous integration: unit testing and integration testing. In unit testing, you test each component to make sure they are producing correct outputs. In addition to unit testing our code, which follows the standard practice for software development, there are two additional types of unit tests when doing CI for ML: the unit tests for our data and the unit tests for our model.

Unit testing for our data is not the same as performing data validation on our raw features. It’s primarily concerned with the results of our feature engineering. You can write unit tests to check whether engineered features are calculated correctly. It includes tests to check whether they are scaled or normalized correctly, one-hot vector values are correct, embeddings are generated and used correctly, and so forth. You will also do tests to confirm whether columns in data are the correct types, in the right range, and not empty, as well as similar checks based on the data type.

Your modeling code should also be written in a modular way that allows it to be testable. You need to write unit tests for the functions you use inside your modeling code to check whether the functions return their output in the correct shape and type, which for numerical features includes testing for NaN, and for string features includes testing for empty strings. You also

need to add tests to make sure the accuracy, error rates, area under the curve (AUC), and receiver operating characteristic (ROC) are above a performance baseline that you specify. Even if the trained model has acceptable accuracy, you need to test it against data slices to make sure the model is accurate for key subsets of the data, in order to avoid bias.

UNIT TESTING CONSIDERATIONS

While you should perform standard unit testing of your code, there are some additional considerations for ML:

- The design of your mocks is especially important for ML unit testing. They should be designed to cover your edge and corner cases, which requires you to think about each of your features and your domain and identify where those edge and corner cases are.
- Ideally your mocks should occupy roughly the same region of your feature space as your actual data would, but much more sparsely, of course, since your mocked dataset should be much smaller than your actual dataset in most cases.

If you've created good mocks and good tests, you should have good code coverage. But just to be sure, take advantage of one of the available libraries to test and track your code coverage.

Infrastructure validation acts as an early warning layer before pushing a model into production, to avoid issues with models that might not run or might perform badly when actually serving requests in production. It focuses on the compatibility between the model server binary and the model that is about to be deployed.

It’s a good idea to include infrastructure validation in your training pipeline so that as you train models you can avoid problems early. You can also run it as part of your CI/CD workflow, which is especially important if you didn’t run it during your model training.

Let’s take a look at an example of running infrastructure validation as part of a training pipeline. In a TFX pipeline, the InfraValidator component takes the model, launches a sandboxed model server with the model, and sees whether the model can be successfully loaded and optionally queried. If the model behaves as expected, it is referred to as “blessed” and is considered ready to be deployed. InfraValidator focuses on the compatibility between the model server binary—for example, TensorFlow Serving—and the model to deploy. Despite the name “infra” validator, it is the user’s responsibility to configure the environment correctly, and InfraValidator only interacts with the model server in the user-configured environment to see whether it works as expected. Configuring this environment correctly will ensure that infravalidation passing or failing will be indicative of whether the model would be servable in the production serving environment.

Continuous Delivery

CI is followed by continuous delivery (CD), which deploys new code and trained models to the target environment. It also ensures compatibility of code and models with the target environment, and for an ML deployment it should check the prediction service performance of the model to make sure the new model can be served successfully.

The full continuous integration/continuous delivery process and infrastructure is referred to as CI/CD. It includes two different forms of data analysis and model analysis. During experimentation, data analysis and model analysis are usually manual processes that are performed by data scientists. Once a model and code have been promoted to a production training pipeline, or if experimentation was done in a training pipeline, data and model analysis should be performed automatically.

As part of the promotion of the code to production, source code is committed to source code control, and CI is initiated. CD then deploys the production code to a production training pipeline, and models are trained. Trained models are then deployed to an online serving environment or batch prediction service. During serving, performance monitoring collects the performance metrics of the model from live data.

Progressive Delivery

Progressive delivery is a software development lifecycle that is built upon the core tenets of CI/CD, but is essentially an improvement over CI/CD. It includes many modern software development processes, including canary deployments, A/B testing, bandits, and observability. It focuses on gradually rolling out new features in order to limit potential negative impact, and gauging user response to new product features.

The process involves delivering changes first to small, low-risk audiences, and then expanding to larger and riskier audiences, thereby validating the results. It offers controls and safeguards like feature flags to increase speed and decrease deployment risk. This can often lead to faster and safer deployments, by implementing a gradual process for both rollout and ownership.

Progressive delivery usually involves having multiple versions deployed at the same time so that comparisons in performance can be made. This practice comes from software engineering, especially for online services. Each of the models performs the same task so that they can be compared. That includes deploying competing models, as in an A/B testing scenario, which is discussed in “A/B testing”; and deploying to shadow environments to limit the deployment risk, as in canary testing, which is discussed in “Canary Deployment”.

Blue/Green Deployment

A simple form of progressive delivery is blue/green deployment, where there are two production serving environments. As shown in Figure 15-10, requests flow through a load balancer that directs traffic to the currently live environment, which is called “Blue.”

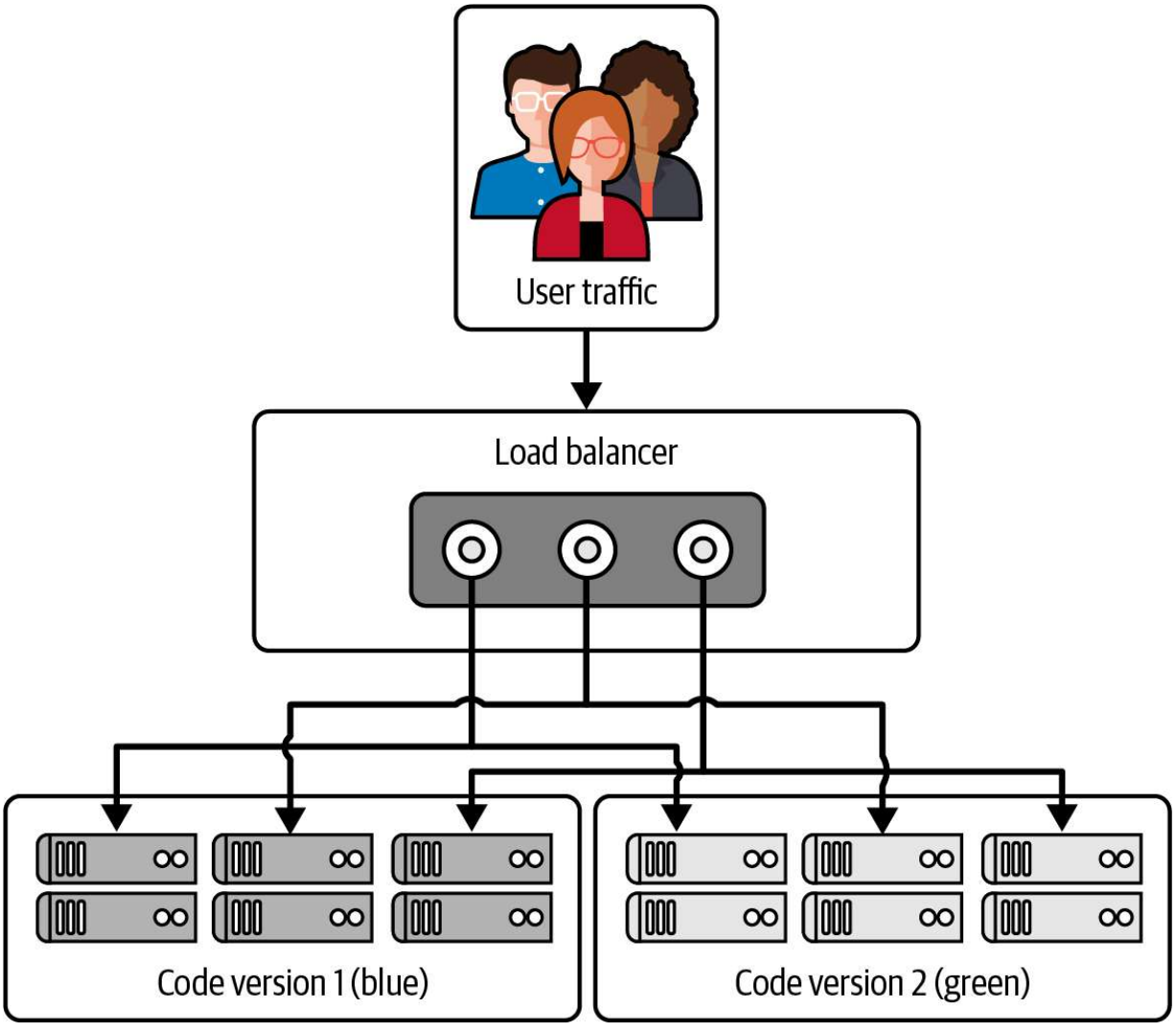

Figure 15-10. Blue/green deployment

Meanwhile, a new version is deployed to the “Green” environment, which acts as a staging setup where a series of tests are conducted to ensure performance and functionality. After passing the tests, traffic is directed to Green deployment. If there are any problems, traffic can be moved back to Blue. This means there is no downtime during deployment, rollback is easy, there is a high degree of reliability, and it includes smoke testing before going live.

Canary Deployment

A canary deployment is similar to a blue/green deployment, but instead of switching the entire incoming traffic from Blue to Green all at once, traffic is switched gradually. Figure 15-11 shows the first stage of a new canary deployment.

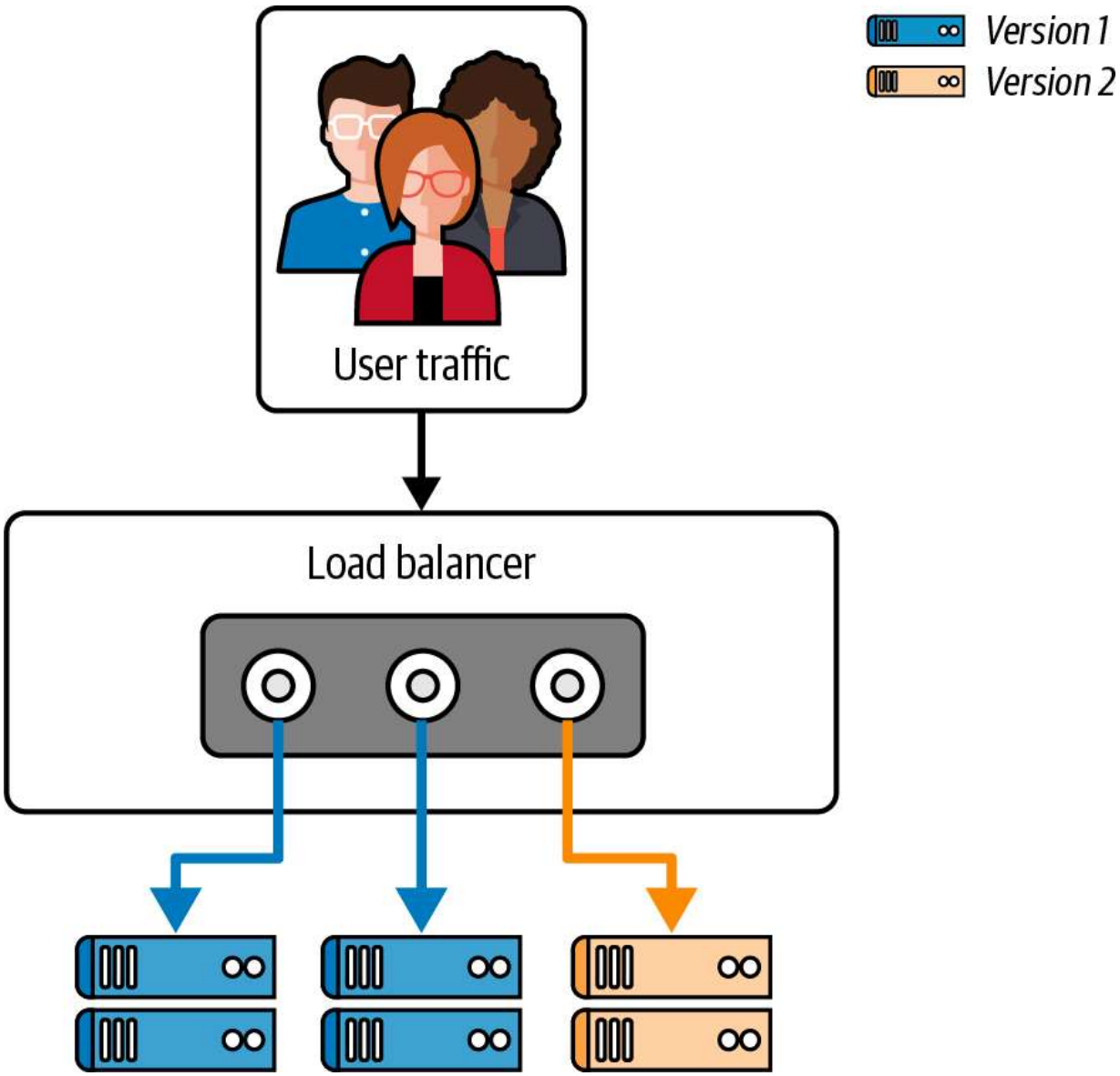

Figure 15-11. Canary deployment

As traffic begins to use the new version, the performance of the new version is monitored. If necessary, the deployment can be stopped and reversed, with no downtime and minimal exposure of users to the new version. Eventually, all the traffic is being served using the new version.

Live Experimentation

Progressive deployment is closely related to live experimentation. Live experimentation is used to test models to measure the actual business results delivered, or to capture data that is as closely associated with business results as you can actually measure. This is necessary because model metrics, which you use to optimize your models during training, are usually not exact matches for the business objectives.

For example, consider recommender systems. You train your model to maximize the click-through rate, which is how your data is labeled. But what the business actually wants to do is maximize profit. This is closely related to click-through, but not an exact match, since some clicks will result in more profit than others. For example, different products have different profit margins.

A/B testing

One simple form of live experimentation is A/B testing. In A/B testing you have at least two different models, or perhaps N different models, and you compare the business results between them to select the model that gives the best business performance. You do that by dividing users into two, or N, groups. You then route user requests to a randomly selected model. Note that it’s important here that the user continues to use the same model for

their entire session if they make multiple requests. You then gather the results from each model to select the one that gives the best results.

A/B testing is actually a widely used tool in many areas of science, not just ML. In a general sense, A/B testing is the process of comparing two variations of the same system, usually by testing the response to variant A versus variant B, and concluding which of the two variants is more effective. Often, A/B testing is used for testing medicines, with one of the variants being a placebo.

Multi-armed bandits

An even more advanced approach is multi-armed bandits. The multi-armed bandit approach is similar to A/B testing, but it uses ML to learn from test results, which are gathered during the test. As it learns which models are performing better, it dynamically routes more and more requests to the winning models. What this means is that eventually all the requests will be routed to a single model, or to a smaller group of similarly performing models. One of the major benefits of this is that it minimizes the use of low-performing models by not waiting for the end of the test to select the winner. The multi-armed bandit approach is a reinforcement learning model architecture that balances exploration and exploitation.

Contextual bandits

An even more advanced approach is contextual bandits. The contextual bandit algorithm is an extension of the multi-armed bandit approach, where you also factor in the customer's environment, or other context of the request, when choosing a bandit. The context affects how a reward is associated with each bandit, so as contexts change, the model should learn to adapt its bandit choice.

For example, consider recommending clothing choices to people in different climates. A customer in a hot climate will have a very different context than a customer in a cold climate.

Not only do you want to find the maximum reward, you also want to reduce the reward loss when you're exploring different bandits. When judging the performance of a model, the metric that measures reward loss is called *regret*, which is the difference between the cumulative reward from the optimal policy and the model's cumulative sum of rewards over time. The lower the regret, the better the model, and contextual bandits help with minimizing regret.

Conclusion

We've covered a lot in this chapter, including model management and delivery and experiment tracking. We also introduced the field of MLOps

and discussed some of the core concepts, including a look at ways to classify the levels of maturity for implementing MLOps processes and infrastructure. In addition, we discussed workflows in some depth, along with model versioning and ways to deliver your applications reliably, including both continuous delivery and progressing delivery. Finally, we explored some ways to do live experimentation on your models and applications.

Throughout this chapter, we've focused on managing your models and delivering your applications (which include your models) to your users reliably and cost-efficiently. For production applications, understanding these architectures and approaches is critical to business success. It's not enough to have a great model. You need to offer it to your users as a great application.