Continuous Delivery of Data, AI/ML Models and Applications (CDAMLA):

The process for developing, deploying, and continuously improving Machine Learning (ML) applications is more complex than traditional enterprise software.

ML applications are impacted by changes in three key dimensions:

- Code
- Model
- Data

ML Applications behavior is often complex and hard to predict due to myriad, conditional driven-driven code paths executed. Thus, they are harder to test, harder to explain, and even harder to improve.

An additional complication is that in real-world Machine Learning (ML) systems, only a small fraction of the system is comprised of actual ML code. An extensive set of infrastructure and processes are needed to support the construction and evolution of ML systems. There are many types of technical debt that accumulate in such systems related to data dependencies, model complexity, reproducibility, testing, monitoring, and changes in the external world.

The industry standard approach to bring automation, quality, and discipline to create a reliable and repeatable process to release software into production is Continuous Delivery. But besides code changes, changes to ML models and the data used to train them are additional types of changes that need to be incorporated into the continuous delivery process to support continuous delivery of ML applications.

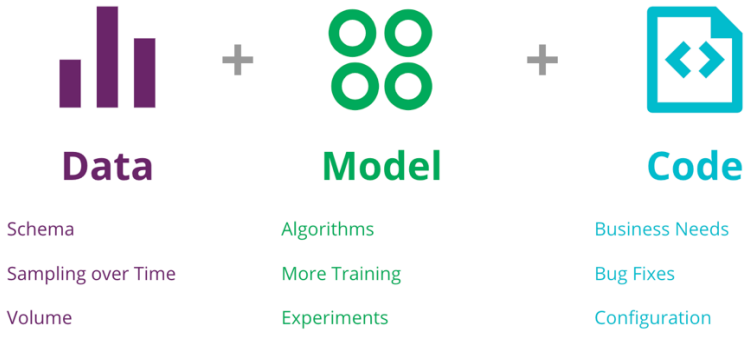


Figure 1: The 3 dimensions of change in a Machine Learning application — data, model, and code — along with a few reasons for them to change

Therefore, the definition of Continuous Delivery i.e. Continuous Delivery for AI & Machine Learning Applications (CDAMLA) has to be extended to incorporate these new elements and challenges that exist in real-world machine learning systems. CDAMLA is defined as follows:

Continuous Delivery for AI & Machine Learning Applications (CDAMLA) *is a software engineering approach in which a cross-functional team produces machine learning applications*

based on code, data, and models in small and safe increments that can be reproduced and reliably released at any time, in short adaptation cycles.

The definition encompasses the principles of continuous delivery which are:

- ***Software engineering approach:*** *Enables teams to efficiently produce high quality software*
- ***Cross-functional team:*** *Experts with different skill sets and workflows across data engineering, data science, machine learning engineering, development, operations, and other knowledge areas work collaboratively taking advantage of the skills and strengths of each team member*
- ***Producing software based on code, data, and machine learning models:*** *All artifacts of the ML software production process require different tools and workflows that must be versioned and managed accordingly*
- ***Small and safe increments:*** *The release of the software artifacts is divided into small increments, which allows visibility and control around the levels of variance of its outcomes, adding safety into the process*
- ***Reproducible and reliable software release:*** *While the model outputs can be non-deterministic and hard to reproduce, the process of releasing ML software into production is reliable and reproducible, leveraging automation as much as possible*
- ***Software release at any time:*** *It is important that the ML software could be delivered into production at any time. Even if organizations do not want to deliver software all the time, it should always be in a releasable state. This makes the decision about when to release it a business decision rather than a technical one*
- ***Short adaptation cycles:*** *Short cycles means development cycles are in the order of days or even hours, not weeks, months or even years. Automation of the process with quality built in is key to achieve this. This creates a feedback loop that allows you to adapt your models by learning from its behavior in production*

The technical components that are important to implement CDAMLA are described using a sample ML application as an exemplar. Concepts with demonstrations on how different tools can be used together to implement the full end-to-end process are detailed. Where appropriate, alternative tool choices to the ones showcased are highlighted. Areas of development and research, given the practice is in its early stages, are also discussed.

Continuous Delivery of AI and Machine Learning Applications:

The Exemplar Application: Predicting Oil Price based on an ML Model

The notion here is to build a sample ML application that solves a public, O&G pertinent problem with a public dataset to illustrate CDAMLA. An Oil Price Prediction application is chosen. It

solves a common forecasting problem faced by many reservoir engineers i.e. predicting the price of Oil in the short term, based on historical data. This involves building a simple web application based on top of a ML model based solution on Kaggle (https://www.kaggle.com/javierbravo/a-tour-of-the-oil-industry/) & Github (https://github.com/pythonbravo/oil_price)

The Python based data science tools used are:

Pandas: https://pandas.pydata.org/pandas-docs/stable/10min.html

Seaborn: https://seaborn.pydata.org/

Scikit Learn: http://scikit-learn.org/stable/

Using a supervised learning algorithm and the scikit-learn Python library, a prediction model is trained using labeled input data. The model is integrated into a simple, python-flask based web application. The model and web application are then deployed to a production environment in the cloud after extensive testing. The figure below shows the high-level process.

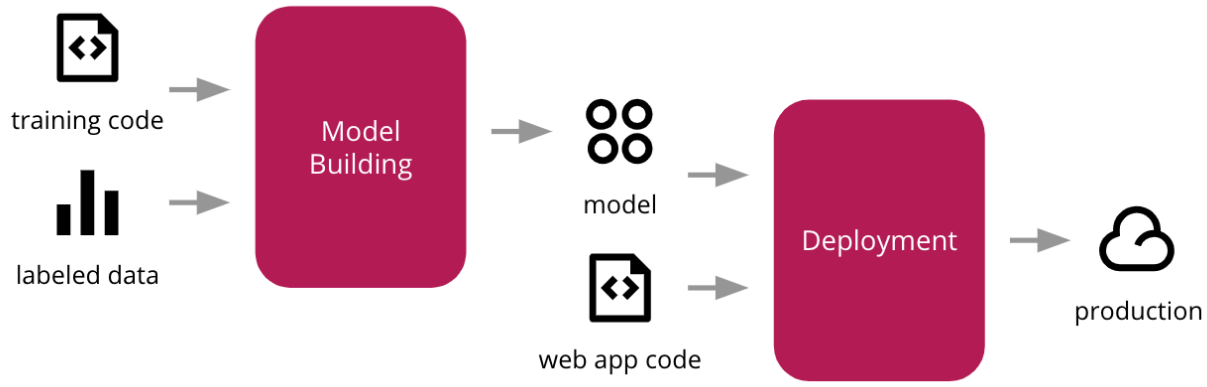


Figure 2: Initial process to train the ML model, integrate it with a web application, and deploy into production

Once deployed, the web application allows users to select a product and a date in the future, and the model will output its prediction of how many units of that product will be sold on that day.

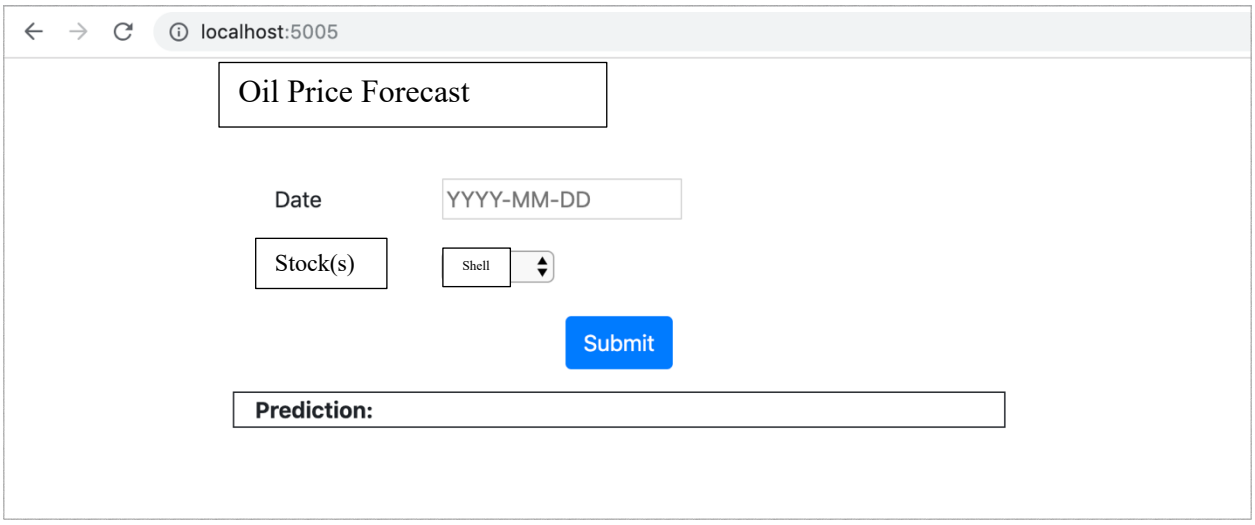


Figure 3: A Web UI to demonstrate the price prediction model in action

CDAMLA Process Implementation Challenges

Implementing the end-to-end CDAMLA process presents *two major challenges*.

1. *The organizational structure challenge*: Different teams typically own different parts of the process. There is a hand over without clear specifications and associated automation support for the process to enable crossing these inter-team boundaries. The teams typically are:

- *Data Engineers:* who build pipelines to make data accessible
- *Data Scientists:* who build and optimize ML models that are served as services
- *Machine Learning Engineers (Developers & Testers):* who integrate the ML models into applications and release it to production after testing

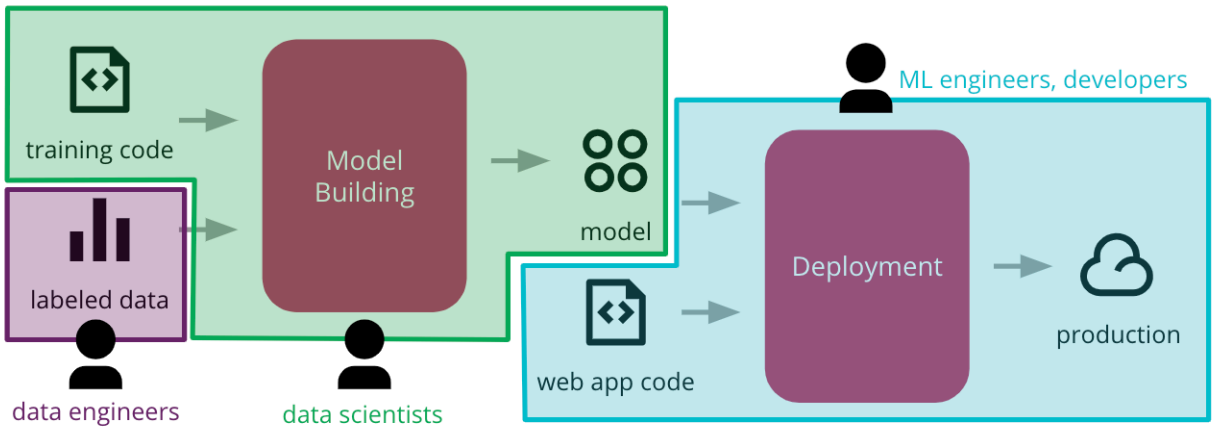


Figure 4: Common functional silos in ML organizations leading to barriers, limiting the ability to automate end-to-end process of deploying AI/ML applications to production

These structural siloes lead to delays and friction. Common symptoms of delay/friction are:

- Models that work in a lab environment and never leave the proof-of-concept phase
- Models that make it to production after delays in a very manual/ad-hoc manner
- Functioning production models quickly becoming stale and hard to update whenever live data pattern shifts occur or new data sources become available

Breaking down these barriers and enabling early and frequent collaboration throughout the engineering process is critical. Agile principles and DevOps can be used effectively to build cross-functional, outcome-oriented teams comprised of specialists from different disciplines to deliver end-to-end ML systems. The focus of this document though, is primarily on the technical components needed to address CDAMLA. This leads us into the second challenge.

2. *The CDAMLA technical challenge*: This relates to how the AI/ML application delivery processes can be made *repeatable, reproducible and auditable*. As these teams use different tools and follow different workflows, it becomes hard to automate it end-to-end. There are more artifacts to be managed beyond code. Versioning them is not straightforward. Some of them can be really large, requiring more sophisticated tools to store and retrieve them efficiently.

Technical Components of CDAMLA

We will explore the CDAMLA process with the exemplar application as the basis. The first step, typically performed by Data Engineers, is comprehending and engineering the data which in this case comprises of:

- Oil price data from the U.S Energy Information administration. *(https://www.eia.gov/dnav/pet/hist/LeafHandler.ashx?n=PET&s=rbrte&f=D)*
- Stock Price Data from Yahoo Finance: Royal Dutch Shell share price dataset query: (https://uk.finance.yahoo.com/quote/RDSB.L/history?period1=946684800&period2=1499122800&interval=1d&filter=history&frequency=1d)

Data Engineers (and Data Scientists) perform Exploratory Data Analysis (EDA) to understand the content characteristics of the data and try to identify broad patterns, outliers, anomalies, transformation, potential feature elements etc. The data required to train useful ML models, will, most likely, not be structured the way a Data Scientist needs them (transformed, cleansed, denormalized, unified) and therefore it highlights the need for the first technical component i.e. *Discoverable, Accessible data*.

Discoverable, Accessible Data

The most common source of data will be internal enterprise systems. However, there is great value in bringing other data sources from outside the organization. A few common patterns for collecting and making data available are a data lake architecture, a more traditional data

warehouse, a collection of real-time data streams, or more recently, a decentralized data mesh architecture.

Regardless of the architecture chosen, data must be easily discoverable and accessible to quickly build useful models and to engineer new features on top of the input data, to improve the model's performance.

In the exemplar application, after doing the initial Exploratory Data Analysis, multiple files will be de-normalized into a single file, and data points that are not relevant or that could introduce unwanted noise into the model are cleaned up. We then store the output in a cloud storage system like Amazon S3 (or Google Cloud Storage or Azure Storage Account).

Using this file to represent a snapshot of the input training data, a simple approach to version the input dataset based on the folder structure and file naming conventions is adopted. Data versioning is an extensive topic, as it can change in two different axes: structural changes to its schema, and the actual sampling of data over time.

In real world projects, it is common to have complex data pipelines to move data from multiple sources into a place and a form where it can be accessed and used easily by Data Scientists who then create Model pipelines for the purposes of training ML models.

Data and ML Pipelines

Data Pipelines are processes that takes input data through a series of transformation stages, producing data as output. Both the input and output data can be fetched and stored in different locations, such as a database, a stream, a file, etc. The transformation stages are usually defined in code, although some ETL tools allow you to represent them in a graphical form. They can be executed either as a batch job, or as a long-running streaming application. For the purposes of CDAMLA, we treat a data pipeline as an artifact, which can be version controlled, tested, and deployed to a target execution environment.

Machine Learning (ML) Pipelines also called "model training pipelines", are the processes that takes data and code as input and produce a trained ML model as the output. This process usually involves data cleansing and pre-processing, feature engineering, model and algorithm selection, model optimization and evaluation. While developing this process encompasses a major part of a Data Scientist's workflow, for the purposes of CDAMLA, we treat the ML pipeline as the final automated implementation of the chosen model training process.

Reproducible Model Training

Once data is available, the process moves into the iterative data science workflow of model building. This usually involves splitting the data into a training set and a validation set, trying different combinations of algorithms, and tuning their parameters and hyper-parameters. That produces a model that can be evaluated against the validation set, to assess the quality of its

predictions. The step-by-step of this model training process becomes the machine learning pipeline.

The figure below shows how to structure the ML pipeline for the price prediction problem, highlighting the different source code, data, and model components. The input data, the intermediate training and validation data sets, and the output model can potentially be large files, which we don't want to store in the source control repository. Also, the stages of the pipeline are usually in constant change, which makes it hard to reproduce them outside of the Data Scientist's local environment.

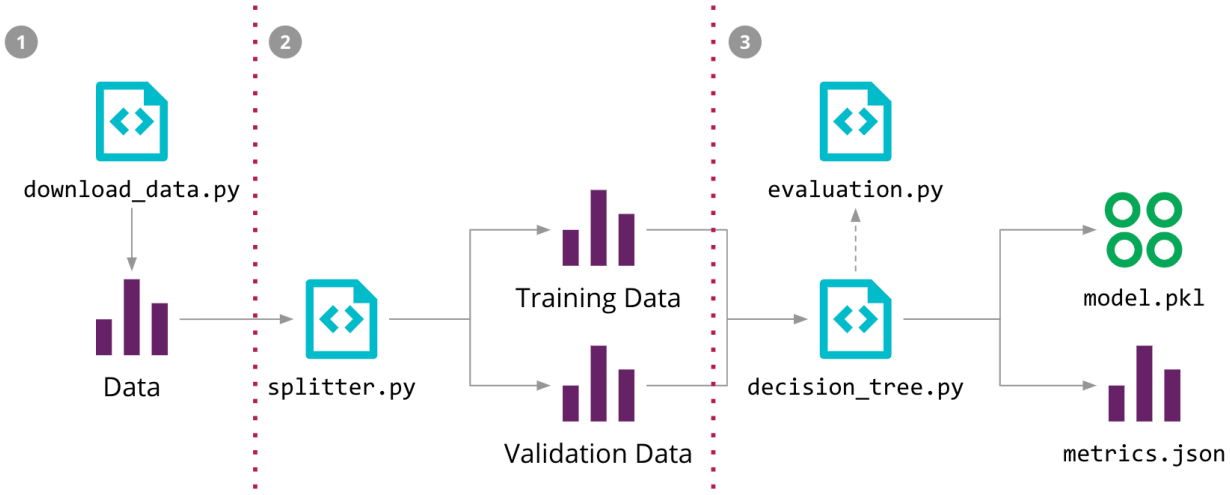


Figure 5: Machine Learning pipeline for our Price prediction problem, and the 3 steps to automate it with DVC

Formalizing the model training process in code is done via the tool DVC (Data Science Version Control) as it provides similar semantics to Git, but also solves a few ML-specific problems:

- DVC has multiple backend plugins to fetch and store large files on an external storage outside of the source control repository;
- DVC can keep track of those files' versions, allowing us to retrain our models when the data changes;
- DVC keeps track of the dependency graph and commands used to execute the ML pipeline, allowing the process to be reproduced in other environments;
- DVC can integrate with Git branches to allow multiple experiments to co-exist;

The initial ML pipeline in the figure above is configured with three `dvc run` commands (`-d` specify dependencies, `-o` specify outputs, `-f` is the filename to record that step, and `-M` is the resulting metrics):

```
dvc run -f input.dvc \ ❶
  -d src/download_data.py -o data/raw/price-data-2016.csv python
src/download_data.py
dvc run -f split.dvc \ ❷
  -d data/raw/store47-2016.csv -d src/splitter.py \
  -o data/splitter/train.csv -o data/splitter/validation.csv python
src/splitter.py
```

```
dvc run ❸
  -d data/splitter/train.csv -d data/splitter/validation.csv -d
src/decision_tree.py \
  -o data/decision_tree/model.pkl -M results/metrics.json python
src/decision_tree.py
```

Each run will create a corresponding file, that can be committed to version control, and that allows other people to reproduce the entire ML pipeline, by executing the `dvc repro` command. Once we find a suitable model, we will treat it as an artifact that needs to be versioned and deployed to production. With DVC, we can use `dvc push` and `dvc pull` commands to publish and fetch it from external storage.

There are other open source tools that can be used to solve these problems: Pachyderm uses containers to execute the different steps of the pipeline and also solves the data versioning and data provenance issues by tracking data commits and optimizing the pipeline execution based on that. MLflow Projects defines a file format to specify the environment and the steps of the pipeline and provides both an API and a CLI tool to run the project locally or remotely. DVC is chosen here because it is a simple CLI tool that solves this part of the problem very well.

Model Serving Patterns

Once a suitable model is found, we need to decide how it will be served and used in production. We have seen a few patterns to achieve that:

- **Embedded model:** this is the simpler approach, where you treat the model artifact as a dependency that is built and packaged within the consuming application. From this point forward, you can treat the application artifact and version as being a combination of the application code and the chosen model.
- **Model deployed as a separate service (MLaaS):** in this approach, the model is wrapped in a service that can be deployed independently of the consuming applications. This allows updates to the model to be released independently, but it can also introduce latency at inference time, as there will be some sort of remote invocation required for each prediction.
- **Model published as data:** in this approach, the model is also treated and published independently, but the consuming application will ingest it as data at runtime. We have seen this used in streaming/real-time scenarios where the application can subscribe to events that are published whenever a new model version is released and ingest them into memory while continuing to predict using the previous version. Software release patterns such as Blue Green Deployment or Canary Releases can also be applied in this scenario.

In the Oil Price Prediction exemplar application, the simpler approach of *embedding the model* is used, given that the consuming application is also written in Python (Flask). The model is exported as a serialized object (pickle file) and pushed to storage by DVC. When building the application, the model is pulled and embedded inside the same Docker container. From that point onwards, the Docker image becomes the application + model artifact that gets versioned and deployed to production.

There are other tool options to implement the embedded model pattern, besides serializing the model object with pickle. MLeap provides a common serialization format for exporting/importing Spark, scikit-learn, and Tensorflow models. There are also language-agnostic exchange formats to share models, such as PMML, PFA, and ONNX. Some of these serialization options are also applicable for implementing the "model as data" pattern. Another approach is to use a tool like H2O to export the model as a POJO in a JAR Java library, which you can then add as a dependency in your application. The benefit of this approach is that you can train the models in a language familiar to Data Scientists, such as Python or R, and export the model as a compiled binary that runs in a different target environment (JVM), which can be faster at inference time.

To implement the "model as service" pattern, many of the cloud providers have tools and SDKs to wrap your model for deployment into their MLaaS (Machine Learning as a Service) platforms, such as AWS Sagemaker.

Azure Machine Learning & Google AI Platform are other providers. Another option is to use a tool like Kubeflow, which is a project designed to deploy ML workflows on Kubernetes, although it tries to solve more than just the model serving part of the problem. MLflow Models is trying to provide a standard way to package models in different flavors, to be used by different downstream tools, some in the "model as a service", some in the "embedded model" pattern.

Regardless of the model serving pattern used, there is an implicit contract between the model and its consumers. The model will usually expect input data in a certain shape. If Data Scientists change that contract to require new input or add new features, it causes integration issues and will break the applications using it.

This is a current area of development. Various tools vendors are working to simplify this task. There is no clear standard (open or proprietary) that can be considered a clear winner. Organizations will need to evaluate the right option for their needs.

Testing and Quality in Machine Learning

There are different types of testing that need to be introduced in the ML workflow. While some aspects are inherently non-deterministic and hard to automate, there are many types of automated tests that can add value and improve the overall quality of the ML system:

- **Validating data:** Tests to validate input data against the expected schema, or to validate our assumptions about its valid values — e.g. they fall within expected ranges or are not null. For engineered features, Unit tests to check they are calculated correctly — e.g. numeric features are scaled or normalized, one-hot encoded vectors contain all zeroes and a single 1, or missing values are replaced appropriately.
- **Validating component integration:** Use an approach similar to testing the integration between different services i.e., Contract Tests to validate that the expected model interface is compatible with the consuming application. Another type of testing that is relevant when the model is productionized in a different format, is to make sure that the exported model still

produces the same results. This can be achieved by running the original and the productionized models against the same validation dataset, and comparing the results are the same.

- **Validating the model quality:** While ML model performance is non-deterministic, Data Scientists usually collect and monitor a number of metrics to evaluate a model's performance, such as error rates, accuracy, AUC, ROC, confusion matrix, precision, recall, etc. They are also useful during parameter and hyper-parameter optimization. As a simple quality gate, these metrics can be used to introduce Threshold Tests or Ratcheting in ML pipelines, to ensure that new models don't degrade against a known performance baseline.
- **Validating model bias and fairness:** while we might get good performance on the overall test and validation datasets, it is important to check how the model performs against baselines for specific data slices. For instance, you might have inherent bias in the training data where there are many more data points for a given value of a feature (e.g. race, gender, or region) compared to the actual distribution in the real world, so it's important to check performance across different slices of the data. A tool like Facets can help you visualize those slices and the distribution of values across the features in your datasets.

In the exemplar application, the evaluation metric defined is a normalized error rate. A simple PyUnit threshold test that breaks if the error rate goes above 80% is created, and this test can be executed prior to publishing a new model version, to demonstrate how we can prevent a bad model from getting promoted.

While these are examples of tests that are easier to automate, assessing a model's quality holistically is harder. Over time, if we always compute metrics against the same dataset, we can start overfitting. And when you have other models already live, you need to ensure that the new model version will not degrade against unseen data. Therefore, managing and curating test data becomes more important.

When models are distributed or exported to be used by a different application, you can also find issues where the engineered features are calculated differently between training and serving time. An approach to help to catch these types of problems is to *distribute a holdout dataset along with the model artifact* and allow the consuming application team to reassess the model's performance against the holdout dataset after it is integrated. This would be the equivalent of a broad integration test in traditional software development.

Other types of tests can be also considered, but we think it's also important to add some manual stages into the deployment pipeline, to display information about the model and allow humans to decide if they should be promoted or not. This allows you to model a Machine Learning governance process and introduce checks for model bias, model fairness, or gather explain-ability information for humans to understand how the model is behaving.

With more types of testing, it will make you rethink the shape of your Test Pyramid: you can consider separate pyramids for each type of artifact (code, model, and data), but also how to combine them, as shown in the figure below. Overall, testing and quality for ML systems is more complex than traditional applications.

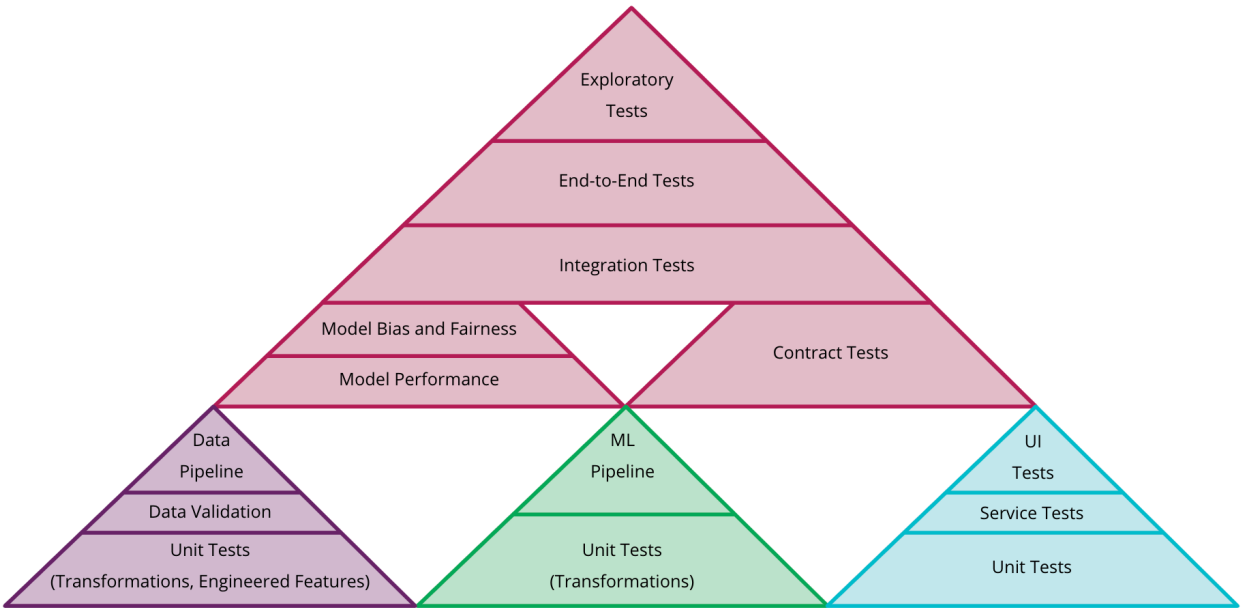


Figure 6: Combining different test pyramids for data, model, and code in CDAMLA

Experiments Tracking

In order to support the governance process, it is important to capture and display information that will allow humans to decide if and which model should be promoted to production. As the Data Science process is very research-oriented, it is common to have multiple experiments being tried in parallel, and many of them will not make it to production.

This experimentation approach during the research phase is different from a more traditional software development process, as we expect that the code for many of these experiments will be thrown away, and only a few of them will be deemed worthy of making it to production. For that reason, we will need to define an approach to track them.

The approach of using different Git branches to track the different experiments in source control is the simplest though it goes against the preferred practice of Continuous Integration on a single trunk. DVC can fetch and display metrics from experiments running in different branches or tags, making it easy to navigate between them.

Some of the drawbacks of developing in feature branches for traditional software development are that

1. it can cause merge pain if the branches are long-lived
2. it can prevent teams from refactoring more aggressively as the changes can impact broader areas of the codebase
3. it hinders the practice of Continuous Integration (CI) as it forces you to setup multiple jobs for each branch and proper integration is delayed until the code is merged back into the mainline.

For ML experimentation the expectation is that most of the branches will never be integrated, and the variation in the code between experiments is usually not significant. From a CI automation perspective, training multiple models for each experiment is required, and gathering metrics on them will inform data scientists which model can move to the next stage of the deployment pipeline.

Besides DVC, another tool to help with experiment tracking is MLflow Tracking. It can be deployed as a hosted service, and provides an API and a web interface to visualize the multiple experiment runs, along with their parameters and performance metrics, as shown in the figure below.

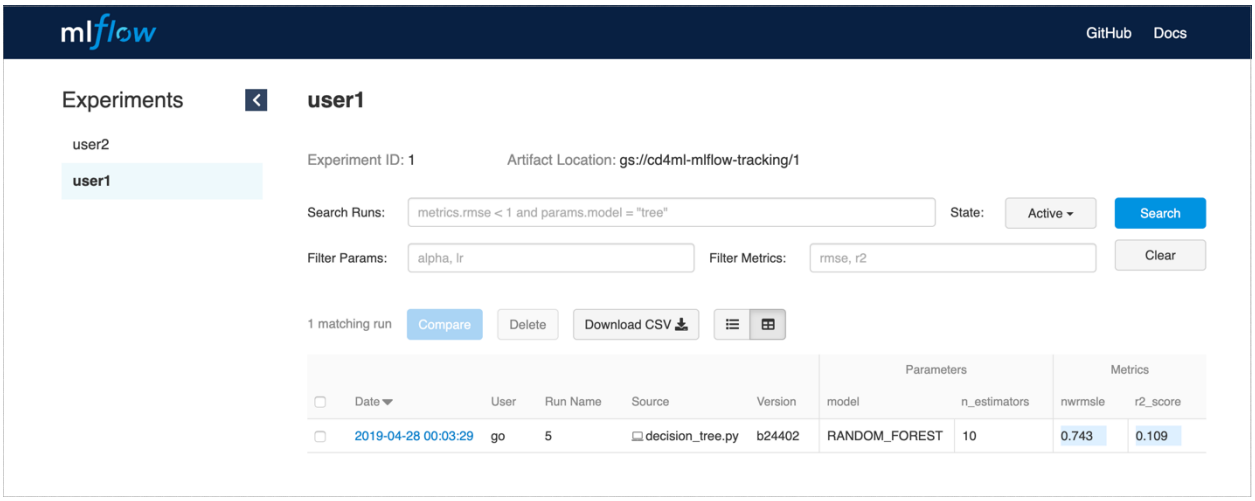


Figure 7: MLflow Tracking web UI to show experiment runs, parameters, and metrics

To support this experimentation process, it is also important to highlight the benefits of having elastic infrastructure, as multiple environments might be needed to be available — and sometimes with specialized hardware — for training.

Cloud-based infrastructure is a natural fit for this, and many of the public cloud providers are building services and solutions to support various aspects of this process.

Model Deployment

In the example application, the method is to experiment to build a single model, which is embedded and deployed with the application. In the real world, deployment might pose more complex scenarios like:

- **Multiple models:** More than one model could be performing the same task. For example, training models to predict Oil Prices by seasons/fiscal quarters. In this case, deploying the models as a separate service might be better for consuming applications to get predictions with a single API call. This can later be evolved to optimally identifying the number of models needed behind that published interface
- **Shadow models:** This pattern is useful when considering the replacement of a model in production. You can deploy the new model side-by-side with the current one, as a *shadow*

model, and send the same production traffic to gather data on how the shadow model performs before promoting it

- **Competing models:** a slightly more complex scenario is when you are trying multiple versions of the model in production — like an A/B test — to find out which one is better. The added complexity here comes from the infrastructure and routing rules required to ensure the traffic is being redirected to the right models, and that you need to gather enough data to make statistically significant decisions, which can take some time. Another popular approach for evaluating multiple competing models is Multi-Armed Bandits, which also requires you to define a way to calculate and monitor the reward associated with using each model. Applying this to ML is an active area of research, and we are starting to see some tools and services appear, such as Seldon core and Azure Personalizer .
- **Online learning models:** unlike the models we discussed so far, that are trained offline and used online to serve predictions, online learning models use algorithms and techniques that can continuously improve its performance with the arrival of new data. They are constantly learning in production. This poses extra complexities, as versioning the model as a static artifact won't yield the same results if it is not fed the same data. You will need to version not only the training data, but also the production data that will impact the model's performance.

Once again, to support more complex deployment scenarios, you will benefit from using elastic infrastructure. Besides making it possible to run those multiple models in production, it also allows you to improve your system's reliability and scalability by spinning up more infrastructure when needed.

Continuous Delivery Orchestration

Deployment Pipelines
In this article we have talked about data pipelines and ML pipelines, but there is another type of pipeline to understand: the Deployment Pipeline. A deployment pipeline automates the process for getting software from version control into production, including all the stages, approvals, testing, and deployment to different environments.

In CDAMLA, we can model automated and manual ML governance stages into our deployment pipeline, to help detect model bias, fairness, or to introduce explainability for humans to decide if the model should further progress towards production or not.

With all of the main building blocks in place, there is a need to tie everything together, and this is where our Continuous Delivery orchestration tools come into place. There are many options of tools in this space, with most of them providing means to configure and execute Deployment Pipelines for building and releasing software to production. In CDAMLA, we have extra requirements to orchestrate: the provisioning of infrastructure and the execution of the Machine Learning Pipelines to train and capture metrics from multiple model experiments; the build, test, and deployment process for our Data Pipelines; the different types of testing and validation to decide which models to promote; the provisioning of infrastructure and deployment of our models to production.

Jenkins (with Blue Ocean) is used as the Continuous Delivery tool, as it was built with the concept of pipelines as a first-class concern. It also allows for the configuration of complex workflows and dependencies by combining different pipelines, their triggers, and defining both manual or automated promotion steps between pipeline stages.
In our simplified example, we haven't built any complex Data Pipelines or infrastructure provisioning yet, but we demonstrate how to combine two Jenkins pipelines. as shown in the Figure below:

- *Machine Learning Pipeline:* to perform model training and evaluation within the Jenkins agent, as well as executing the basic threshold test to decide if the model can be promoted or not. If the model is good, we perform a `dvc push` command to publish it as an artifact.
- *Application Deployment Pipeline:* to build and test the application code, to fetch the promoted model from the upstream pipeline using `dvc pull`, to package a new combined artifact that contains the model and the application as a Docker image, and to deploy them to a Kubernetes production cluster.

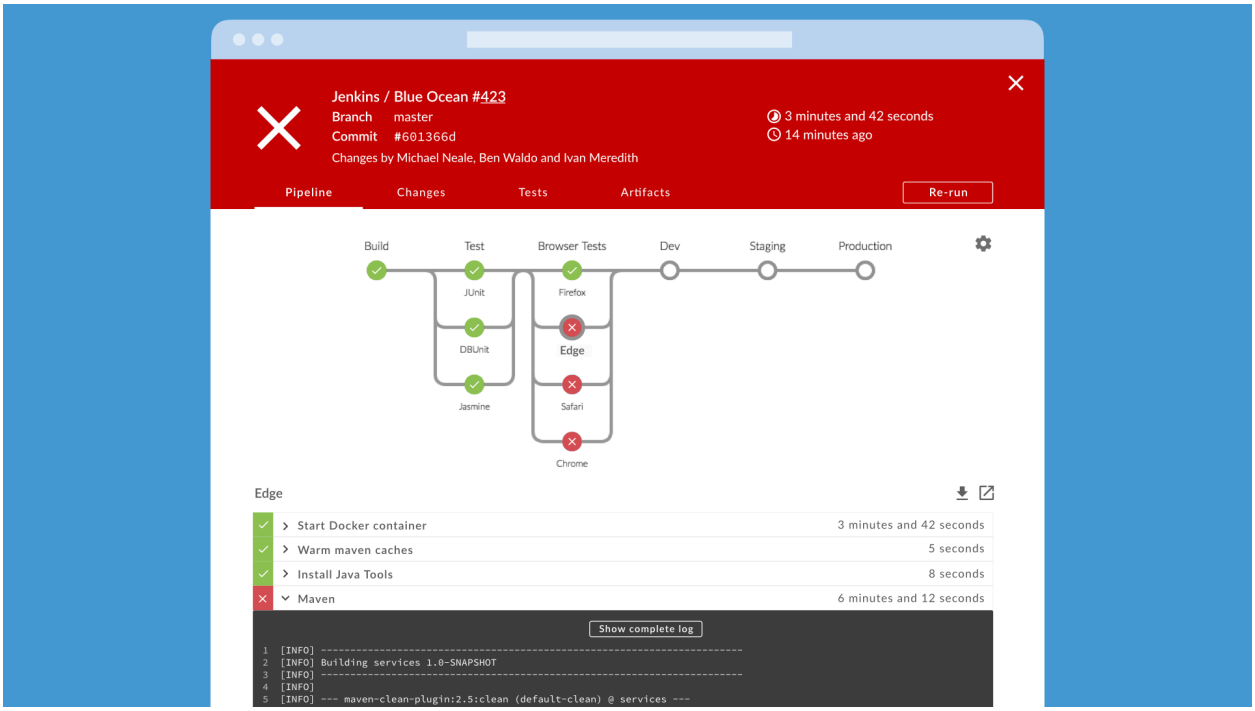


Figure 8: Combining Machine Learning Pipeline and Application Deployment Pipeline in GoCD

Over time, the ML pipeline can be extended to execute multiple experiments in parallel (a feature supported by Jenkins fan-out/fan-in model), and to define your model governance process to check for bias, fairness, correctness, and other types of gates to make informed decisions about which model is promoted and deployed to production.

Finally, another aspect of your Continuous Delivery orchestration is to define a process for rollbacks, in case a deployed model turns out to perform poorly or incorrectly in production. This adds another safety net into the overall process.

Model Monitoring and Observability

Now that the model is live, we need to understand how it performs in production and close the data feedback loop. Here we can reuse all the monitoring and observability infrastructure that might already be in place for your applications and services.

Tools for log aggregation and metrics collection are usually used to capture data from a live system such as business KPIs, software reliability and performance metrics, debugging information for troubleshooting, and other indicators that could trigger alerts when something goes out of the ordinary. We can also leverage these same tools to capture data to understand how our model is behaving such as:

- **Model inputs:** what data is being fed to the models, giving visibility into any training-serving skew. Model outputs: what predictions and recommendations are the models making from these inputs, to understand how the model is performing with real data.
- **Model interpretability outputs:** metrics such as model coefficients, ELI5, or LIME outputs that allow further investigation to understand how the models are making predictions to identify potential overfit or bias that was not found during training.
- **Model outputs and decisions:** what predictions our models are making given the production input data, and also which decisions are being made with those predictions. Sometimes the application might choose to ignore the model and make a decision based on pre-defined rules (or to avoid future bias).
- **User action and rewards:** based on further user action, we can capture reward metrics to understand if the model is having the desired effect. For example, if we display product recommendations, we can track when the user decides to purchase the recommended product as a reward.
- **Model fairness:** analysing input data and output predictions against known features that could bias, such as race, gender, age, income groups, etc.

In our example, we are using the EFK stack for monitoring and observability, composed of three main tools:

- Elasticsearch: an open source search engine.
- FluentD: an open source data collector for unified logging layer.
- Kibana: an open source web UI that makes it easy to explore and visualize the data indexed by Elasticsearch.

We can instrument our application code to log the model inputs and predictions as an event in FluentD:

predict_with_logging.py…

```
  df = pd.DataFrame(data=data, index=['row1'])
  df = decision_tree.encode_categorical_columns(df)
  pred = model.predict(df)
  logger = sender.FluentSender(TENANT, host=FLUENTD_HOST,
port=int(FLUENTD_PORT))
  log_payload = {'prediction': pred[0], **data}
  logger.emit('prediction', log_payload)
```

This event is then forwarded and indexed in ElasticSearch, and we can use Kibana to query and analyze it through a web interface, as shown in Figure 9.

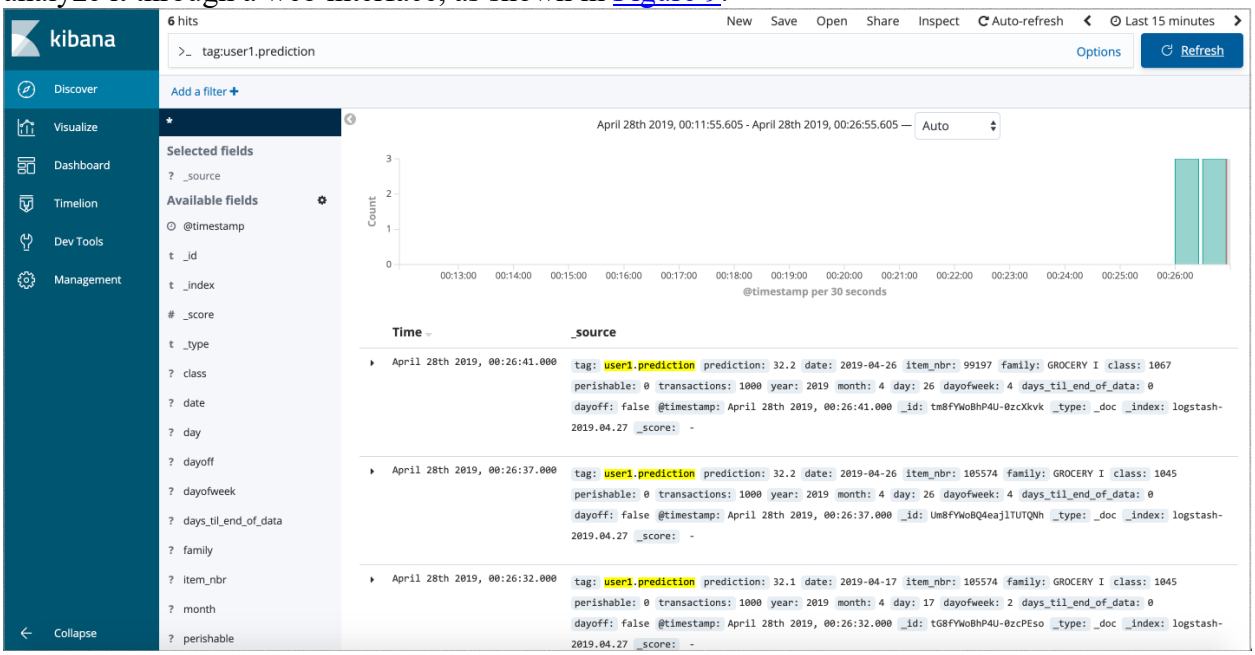


Figure 9: Analysing our model's predictions against real input data in Kibana

There are other popular monitoring and observability tools, such as the ELK stack (a variation that uses Logstash instead of FluentD for log ingestion and forwarding), Splunk, among others. Collecting monitoring and observability data becomes even more important when you have multiple models deployed in production. For example, you might have a shadow model to assess, you might be performing split tests, or running multi-arm bandit experiments with multiple models.
This is also relevant if you are training or running federated models at the edge — e.g. on a user's mobile device — or if you are deploying online learning models that diverge over time as they learn from new data in production.

By capturing this data, you can close the data feedback loop. This is achieved either by collecting more real data (e.g. in a pricing engine or a recommendation system) or by adding a human in the loop to analyse the new data captured from production, and curate it to create new training datasets for new and improved models. Closing this feedback loop is one of the main advantages of CDAMLA, as it allows us to adapt our models based on learnings taken from real production data, creating a process of continuous improvement.

The End-to-End CDAMLA Process

By tackling each technical challenge incrementally, and using a variety of tools and technologies, we managed to create the end-to-end process shown in Figure 10 that manages the promotion of artifacts across all three axis: code, model, and data.

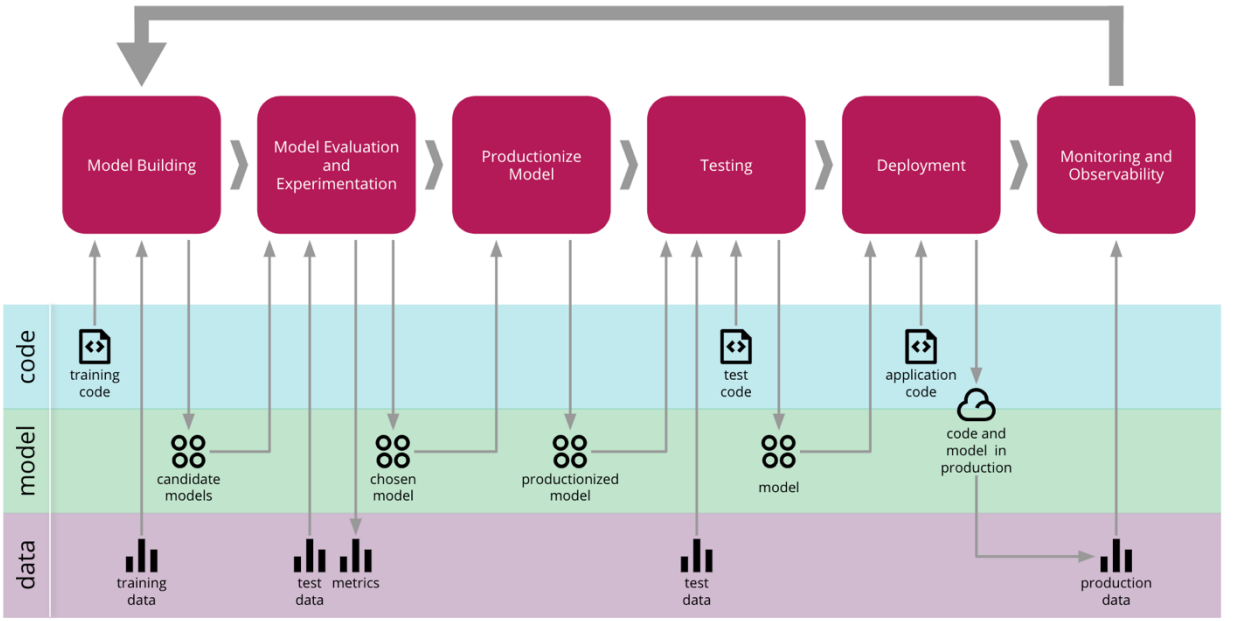


Figure 10: Continuous Delivery for Machine Learning end-to-end process

At the base, what is needed is an easy way to manage, discover, access, and version data. It is then possible to automate the model building and training process to make it reproducible. This allows for experimentation and training of multiple models, which brings a need to measure and track those experiments. Once a suitable model is found, a decision can be made on how it will be productionized and served. Because the model is evolving, ensure that it won't break any contract with its consumers is key, therefore the need to test it before deploying to production is critical. Once in production, monitoring and observability infrastructure can be used to gather new data that can be analyzed and used to create new training data sets, closing the feedback loop of continuous improvement.

A Continuous Delivery orchestration tool coordinates the end-to-end CDAMLA process, provisions the desired infrastructure on-demand, and governs how models and applications are deployed to production.

The example application and code used here is available on here to invite improvements: https://github.com/keshava/CDAMLA

Data Versioning

In Continuous Delivery, every code commit is treated as a release candidate, which triggers a new execution of the deployment pipeline. Assuming that commit passes through all the pipeline stages, it can be deployed to production. When talking about CDAMLA, one of the regular questions is *"how can I trigger a pipeline when the data changes?"*
In our example, the Machine Learning pipeline in Figure 5 starts with the `download_data.py` file, which is responsible for downloading the training dataset from a

shared location. If we change the contents of the dataset in the shared location, it won't immediately trigger the pipeline, as the code has not changed and DVC won't be able to detect it. To version the data we would have to create a new file or change the file name, which in turn would require us to update the `download_data.py` script with the new path and therefore create a new code commit.

An improvement to this approach would be to allow DVC to track the file content for us, by replacing our hand-written download script with:

```
dvc add data/raw/store47-2016.csv
```

This will slightly change the first step of our Machine Learning pipeline, as shown in Figure 11 below.

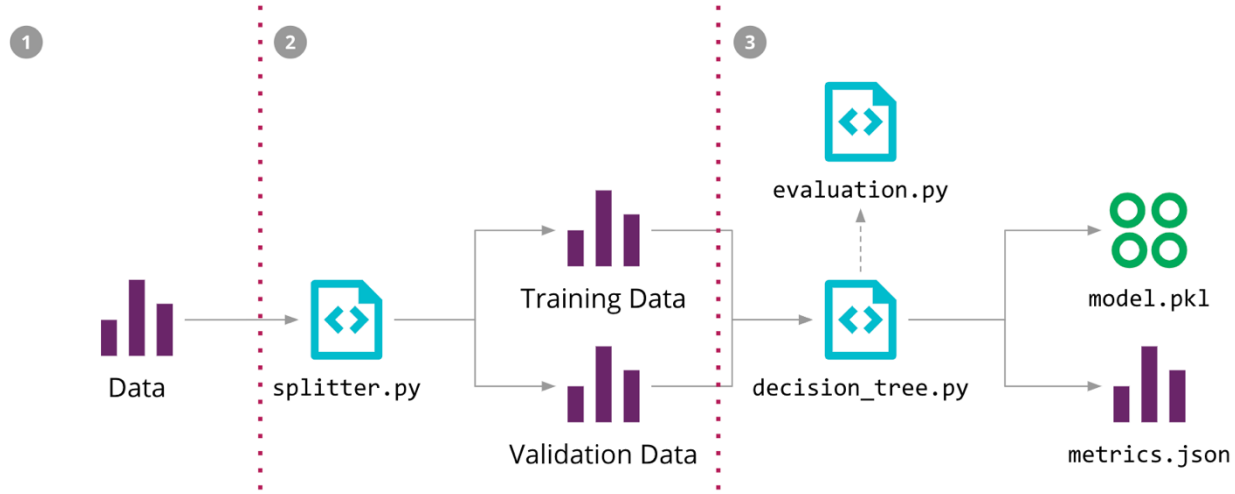


Figure 11: updating the first step to allow DVC to track the data versions and simplifying the ML Pipeline

This creates a metadata file that tracks a checksum of the file's content which we can commit to Git. Now when the file contents change, the checksum will change and DVC will update that metadata file, which will be the commit we need to trigger a pipeline execution.

While this allows us to retrain our model when the data changes, it doesn't tell the entire story about data versioning. One aspect is data history: ideally you would want to keep an entire history of all data changes, but that's not always feasible depending on how frequently the data changes. Another aspect is data provenance: knowing what processing step caused the data to change, and how that propagates across different data sets. There is also a question around tracking and evolving the data schema over time, and whether those changes are backwards and forwards compatible.

In a streaming world, these aspects of data versioning become even more complicated to reason about, so this is an area where practices, tools and techniques will continue to evolve.

Data Pipelines

Versioning, testing, deploying, and monitoring data pipelines is a complex task. Some tooling options are better than others to enable CDAMLA. Many ETL tools that require you to define transformation and processing steps through a GUI, are usually not easy to version control, test, or deploy to hybrid environments. Some of them can generate code that you can treat as an artifact and put through a deployment pipeline.

Open Source tools that allow us to define the Data Pipelines in code, is easier to version control, test, and deploy. For example, if when using Spark, having the data pipeline written in Scala, allows testing using ScalaTest or spark-testing-base and packaging the job as a JAR artifact that can be versioned and deployed on a Deployment Pipeline in Jenkins/GoCD.

As Data Pipelines are usually either running as a batch job or as a long-running streaming application, we didn't include them in the end-to-end CDAMLA process diagram in figure 10, but they are also another potential source of integration issues if they change the output that either your model or your application expect. Therefore, having integration and data Contract Tests as part of Deployment Pipelines to catch those mistakes is key.

Another type of test that is relevant for Data Pipelines is a data quality check, but this can become another extensive topic of discussion, and is probably better to be covered separately.

Platform Thinking

As you might have noticed, various tools and technologies were used to implement CDAMLA. If you have multiple teams trying to do this, they might end up reinventing things or duplicating efforts. This is where platform thinking can be useful. Not by centralizing all the efforts in a single team that becomes a bottleneck, but by focusing the platform engineering effort into building domain-agnostic tools that hide the underlying complexity and speed up teams trying to adopt it.

Applying platform thinking to CDAMLA is why we see a growing interest in Machine Learning platforms and other products that are trying to provide a single solution to manage the end-to-end Machine Learning lifecycle. Many of the major technology giants have developed their own in-house tools, but we believe this is an active area of research and development, and expect new tools and vendors to appear, offering solutions that can be more widely adopted.

Evolving Intelligent Systems without Bias

As soon as your first Machine Learning system is deployed to production, it will start making predictions and be used against unseen data. It might even replace a previous rules-based system you had. It is important to realise that the training data and model validation you performed was based on historical data, which might include inherent bias based on how the previous system behaved. Moreover, any impact your ML system might have on your users going forward, will also affect your future training data.

Let's consider two examples to understand the impact. First, let's consider the price forecasting solution we explored throughout this article. Let's assume there is an application that takes the predicted price to decide the exact quantity of crude to be ordered and offered to customers. If the predicted demand is lower than the actual demand, you will not have enough crude to sell and therefore the transaction volume for that product decreases. If you only use these new transactions as training data for improving your model, over time your demand forecast predictions will degrade.

For the second example, imagine that you are building an anomaly detection model to decide if a customer's credit card transaction is fraudulent or not. If your application takes the model decision to block them, over time you will only have "true labels" for the transactions allowed by the model and less fraudulent ones to train on. The model's performance will also degrade because the training data becomes biased towards "good" transactions.

There is no simple solution to this problem. On our first example, retailers also consider out-of-stock situations and order more items than forecasted to cover a potential shortage. For the fraud detection scenario, we can ignore or override the model's classification sometimes, using some probability distribution. It is also important to realise that many datasets are temporal, i.e. their distribution changes over time. Many validation approaches that perform a random split of data assume they are i.i.d. (independent and identically distributed), but that is not true once you consider the effect of time.

Therefore, it is important to not just capture the input/output of a model, but also the ultimate decision taken by the consuming application to either use or override the model's output. This allows you to annotate the data to avoid this bias in future training rounds. Managing training data and having systems to allow humans to curate them is another key component that will be required when you face these issues.

Evolving an intelligent system to choose and improve ML models over time can also be seen as a meta-learning problem. Many of the state of the art research in this area is focused on these types of problems. For example, the usage of reinforcement learning techniques, such as multi-arm bandits, or online learning in production. We expect that our experience and knowledge on how to best build, deploy, and monitor these types of ML systems will continue to evolve.

SageMaker based CDAMLA

The following steps are involved in deploying a model using SageMaker -

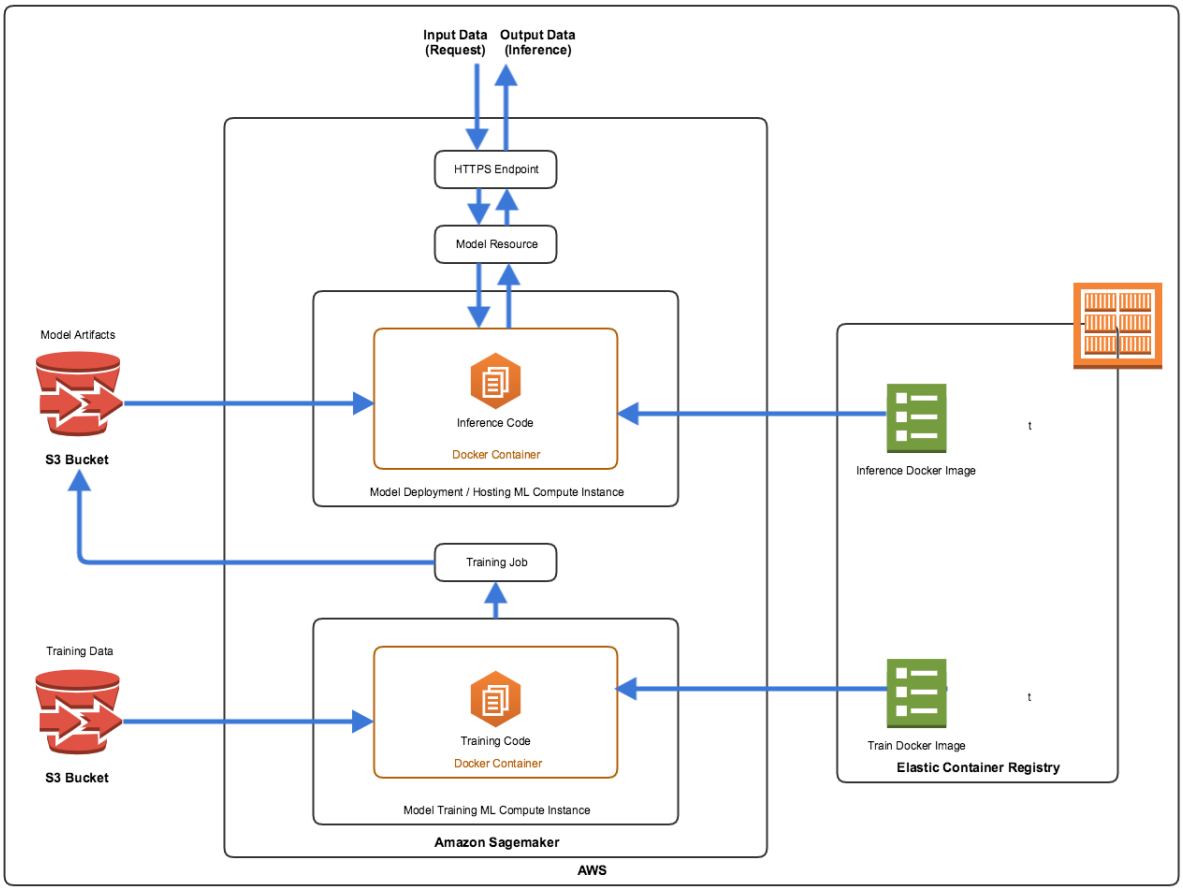
1. Build the docker image for training and upload to ECR (Elastic Container Registry) This image holds the training code and dependencies

2. Create and start SageMaker training job (SageMaker CreateTrainingJob API)

3. Build the docker image for serving (inferencing) and upload to ECR (Elastic Container Registry) This image holds your inference code and dependencies.

4. Create SageMaker Model (SageMaker CreateModel API)

5. Create/update SageMaker endpoint that hosts the model (SageMaker CreateEndpoint API)



Sagemaker Architecture BYOM (Bring Your Own Model) Approach

SageMaker algorithms are packaged as Docker images. This provides the flexibility to use almost any algorithm code with SageMaker, regardless of implementation language, dependent libraries, frameworks, and so on. Custom training algorithms can be used and as well as custom inference code. The training algorithm and inference code are packaged in Docker images, and these images are used to train a model and deploy it with Amazon SageMaker.

- Training — When Sagemaker creates the training job, it launches the ML compute instance, runs the train docker image which creates the docker container in the ML compute instance, injects the training data from an S3 location into the container and uses the training code and training dataset to train the model. It saves the resulting model artifacts and other output in the S3 bucket specified for that purpose.

- Deployment — For model deployment, Sagemaker first creates the model resource using the S3 path where the model artifacts are stored and the Docker registry path for the

image that contains the inference code. It then creates an HTTPS endpoint using the endpoint configuration which specifies the production model variant and the ML compute instances to deploy to.

- Usage —The client application sends requests to the Sagemaker HTTPS endpoint to obtain inferences from a deployed model.

End to End Model Deployment with Sagemaker

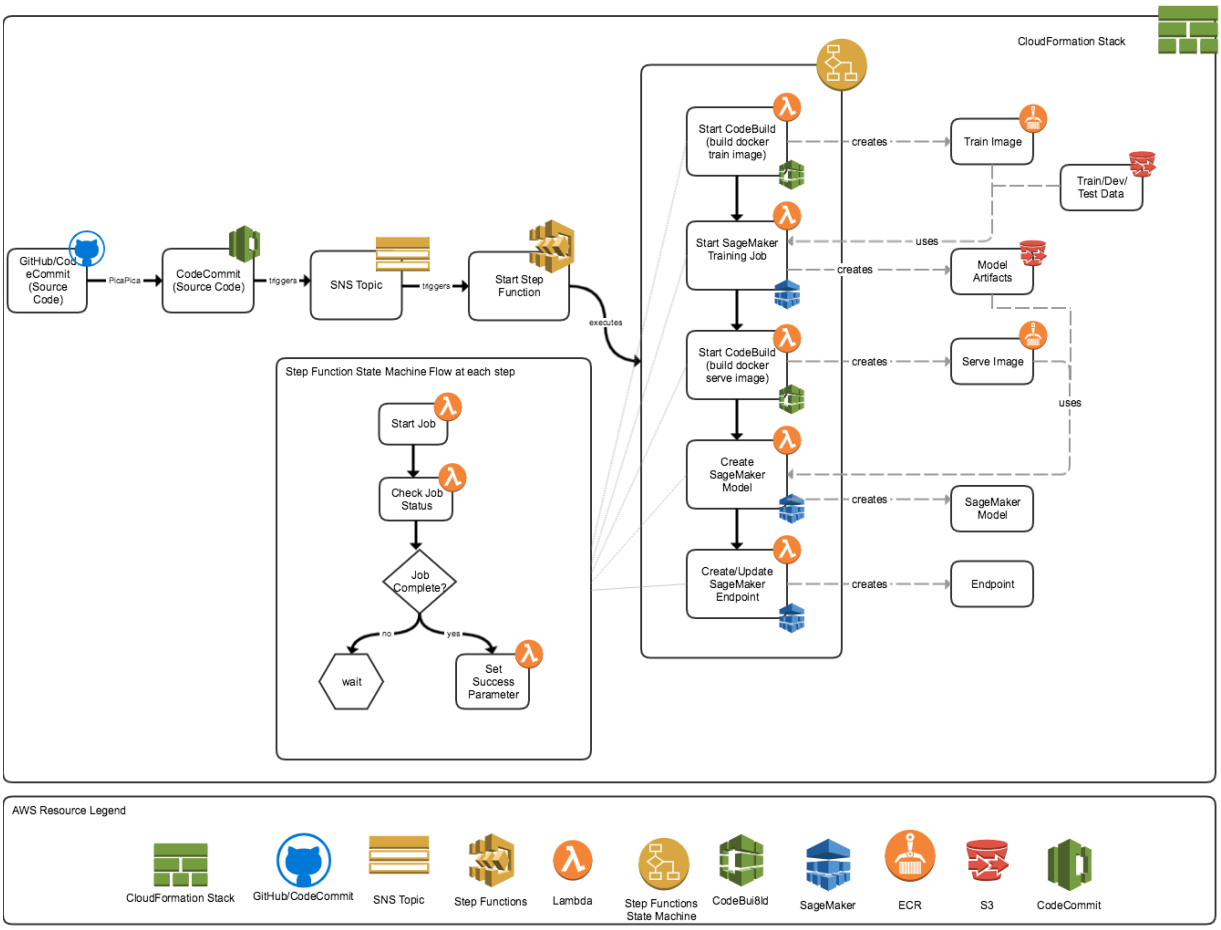
Here is the design for an end-to-end model deployment pipeline using Sagemaker and a host of other different AWS services with AWS Step Functions used as the main coordinator and workflow orchestrator.

- **AWS Step Functions** are used for workflow orchestration
- Step Functions provide the ability to design and run workflows that stitch together multiple AWS services such as AWS Lambda, Amazon ECS etc.
- Step Functions also translate workflows into a state machine diagram for an easy visual representation and monitoring

Below is a high-level overview of how the services work together -

- Step functions act like a state machine, beginning with an initial state and transforming the state using AWS Lambda functions — changing, branching or looping through states as needed
- AWS Lambda functions are used for starting model training, image building, checking on train and build status and so on
- AWS CodeBuild is used to build docker images and push them to an Elastic Container Registry (ECR) repository
- AWS Systems Manager Parameter Store provides a shared, centralized parameter store for our training and deployment jobs
- AWS Lambda functions query the parameters from this store
- AWS Simple Notification Service (SNS) is used for starting builds and for notifications
- AWS Simple Storage Service (S3) buckets are used to hold model training data and trained model artifacts
- A Github/CodeCommit repository publishes to an SNS topic when a code change is made. Notifications are also published to an SNS topic when the build has started, finished, and failed

The following diagram shows how the services work together:



Automatic Build and Deployment with AWS Sagemaker and Step Functions Architecture

AWS Step Functions Data Science SDK for Amazon SageMaker

The AWS Step Functions Data Science Software Development Kit (SDK) is an open-source library that allows the creation of workflows that pre-process data and then train and publish machine learning models using Amazon SageMaker and AWS Step Functions. You can create machine learning workflows in Python that orchestrate AWS infrastructure at scale, without having to provision and integrate the AWS services separately.

AWS Step Functions is a serverless orchestration service that allows you to build resilient workflows using AWS services such as Amazon SageMaker, AWS Glue, and AWS Lambda. Amazon SageMaker enables you to build, train and deploy machine learning models quickly. The new Data Science SDK, data and model pipelines can be built on AWS infrastructure using the preferred tools of data scientists - Python and Jupyter Notebooks.

The Data Science SDK ca be used to create and visualize end-to-end data science workflows that perform tasks such as data pre-processing on AWS Glue and model training, hyperparameter tuning, and endpoint creation on Amazon Sagemaker. You can reuse the workflows in production by exporting AWS CloudFormation templates.

The Data Science SDK is included in AWS Step Functions and is available in all regions where both AWS Step Functions and Amazon SageMaker are available. The SDK can be used in conjunction with other services such as AWS Glue and AWS Lambda in their supported regions.