Machine Learning Best Practices in Financial Services

An overview of security and machine learning governance practices using Amazon SageMaker

July 2020

Notices

Customers are responsible for making their own independent assessment of the information in this document. The information included in this document is for illustrative purposes only. Nothing in this document is intended to provide you legal, compliance, or regulatory guidance. You should review the laws that apply to you. This document: (a) is for informational and illustrative purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

Contents

Abstract

This whitepaper outlines security and model governance considerations for financial institutions using machine learning applications. It illustrates how you can create a secure machine learning environment on AWS and use best practices in model governance based on your organization's risk tolerance, integration with existing governance, and regulatory expectations.

Are you Well-Architected?

The AWS Well-Architected Framework helps you understand the pros and cons of the decisions you make when building systems on AWS. Using the Framework allows you to learn architectural best practices for designing and operating reliable, secure, efficient, and cost-effective systems in the cloud.

In the Machine Learning Lens, we focus on how to design, deploy, and architect your machine learning workloads in the AWS Cloud. This lens adds to the best practices described in the Well-Architected Framework.

Introduction

Financial institutions are increasingly building machine learning (ML) models to transform their businesses. These institutions apply ML across a broad range of use cases including:

- fraud detection
- market surveillance
- due diligence
- portfolio optimization
- customer experience enhancement
- optimized trade execution
- quantitative strategy development
- algorithmic trading

An ML model's ability to learn helps drive more accurate predictions. But care must be taken to help ensure such models are being used in compliance with regulatory requirements and are well governed and secure.

There are a number of publications that the financial services industry can use for guidance. For example,

- The Guidance on Model Risk Management was developed jointly by the Federal Reserve and the OCC. This reference discusses the need for effective model governance, including guidance noted in Federal Reserve SR 11-7, OCC 2011-12.
- The European Central Bank (ECB) guide to internal models illustrates how you can implement a secure and well-governed machine learning environment.

The ML lifecycle: How to provision, govern, and operationalize ML workloads

This section presents a high-level overview of the various components that are part of an end-to-end machine learning pipeline. While this is not intended to be an exhaustive guide, it helps frame our discussion of security, governance, and operationalization of machine learning workflows for financial services. Figure 1 illustrates the machine learning lifecycle graphically.

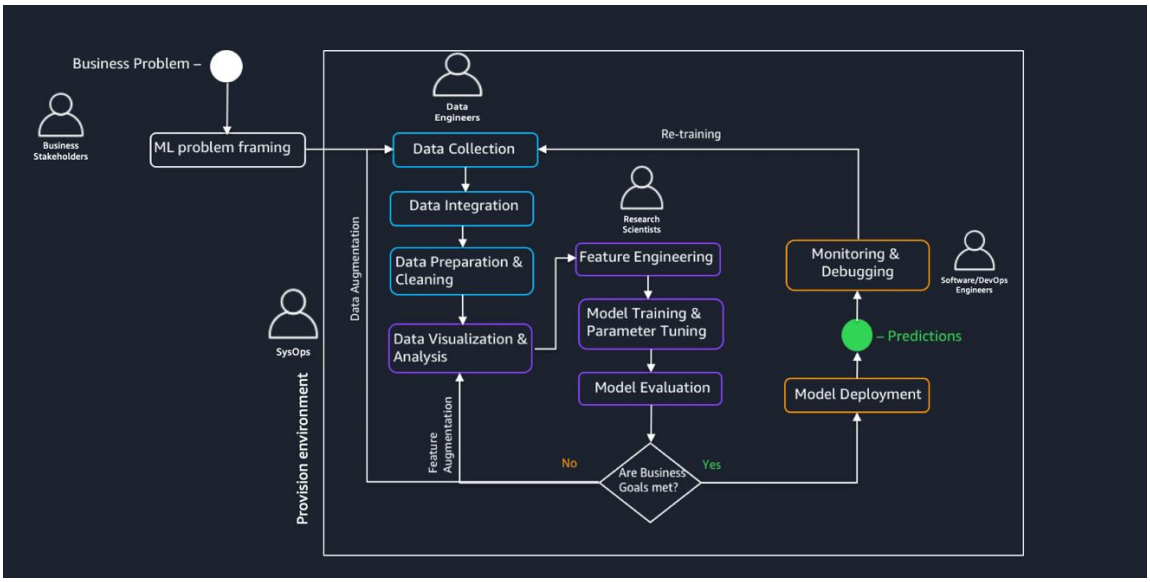


Figure 1 – Typical ML lifecycle

A typical ML workflow involves multiple stakeholders. To successfully govern and operationalize this workflow requires collaboration across these teams. Stakeholders usually frame the high-level business problem. Once identified, you must transform the business problem and frame it into a machine learning problem, with well-defined inputs and outputs. This process typically involves business stakeholders engaging with their data science center of excellence team, along with IT, and risk and compliance partners, to understand the complexity of implementing the use case.

At this point, you must also gather any relevant success metrics or key performance indicators (KPIs), both from the perspective of the model as well as any non-functional requirements in a production solution, such as any regulatory or compliance needs that may dictate the validity of the solution. For example, there may be regulations that impose certain requirements on explaining the reasons for decisions that are made .This may dictate the types of algorithms that the data science teams can explore. This is typically driven by regulatory or compliance experts, as well as business stakeholders.

The next stage is collecting the input data from various sources. This role is often performed by engineering teams familiar with big data tools for data ingestion, extraction, transformation, and loading (ETL). It is important to ensure that the data is versioned and the lineage of the data tracked for auditing and compliance, detailed in Establish ML Governance.

Once the data is collected, data scientists typically explore an appropriately selected subset of the data to understand the schema of the dataset, clean the dataset to remove or impute any missing values, identify necessary transformations (such as removal of outliers, scaling input-independent variables, dimensionality reduction, encoding categorical variables etc.) that may be required to transform the data into a numerical format, suitable for building models. This process, known as “feature engineering” typically consumes the majority of a data scientist’s time, but is a crucial step to identifying key features that are predictive, and contain signals that correlate with the dependent variable.

Once a set of features has been selected, data scientists typically decide which framework they want to use and which models they want to try. Then they start the training iterations across models. Modeling performance against the identified KPIs can be improved by tuning hyperparameters (parameters whose values need to be defined at the outset prior to training) or by incorporating more labeled training data to help the model see more instances and thus learn more. Once the models are trained, the model artifacts and any corresponding code must be properly versioned and stored in a centralized code repository or in an artifact management system. Note that this stage of the process is experimental and data scientists may often go back to feature engineering or even the data collection stage if the model performance is consistently poor. Data scientists also use a number of methods to identify what features are important in driving model performance.

Next, DevOps engineers will typically pick up trained models and prepare them for deployment into production. At this point, models need to be tested on integration with existing pipelines and business workflows, back tested on historical data, performance tested for producing inferences at scale, and quality-tested against any historical models. All the dependencies, model artifacts and any related metadata need to be managed and stored. At this stage, there is typically a manual ("human-in-the-loop") process to ensure the model meets the business goals, as well as any regulatory or compliance requirements. Once the model is deployed for production, the model is regularly monitored for performance degradation, data drift, or concept drift in order to trigger any retraining or deprecating the model, which may be required as the needs of the business evolve. This is often a best practice but is also recommended by regulations that require banking organizations to periodically review their models.

Figure 2 shows the different technical and business stakeholders that are typically engaged throughout the machine learning lifecycle.

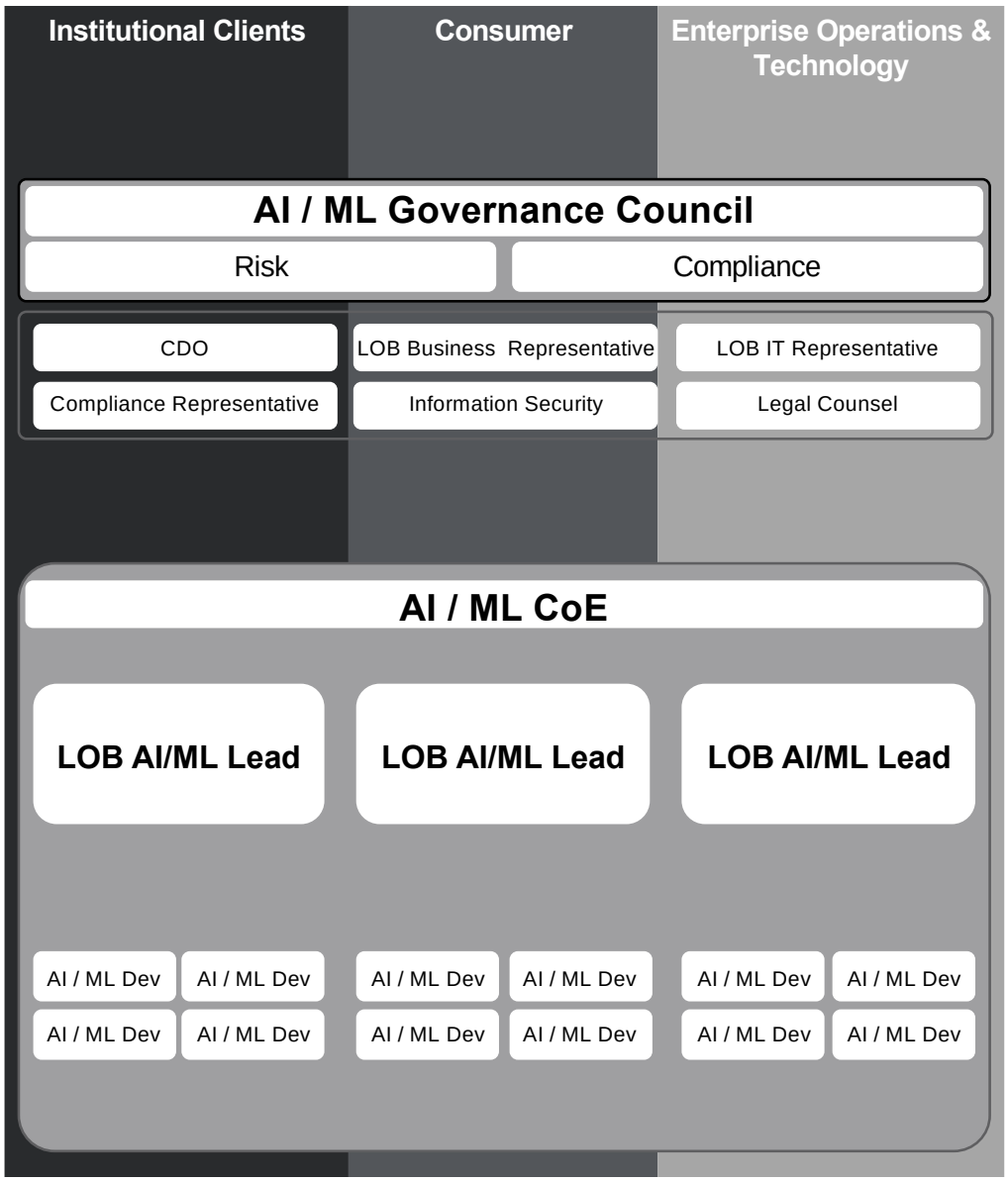


Figure 2 – Typical stakeholders and their responsibilities in an ML lifecycle

As this shows, it is necessary to ensure the proper layers of access controls, security, and governance and monitoring at each stage of the ML lifecycle.

The rest of this whitepaper provides an example or illustration of how you can use Amazon SageMaker, along with other AWS tools throughout the ML lifecycle. More specifically, based on our customers' experience running regulated predictive models, we cover the following topics:

A. <u>Provision a secure ML environment:</u>

- **Compute and network isolation:** How to deploy SageMaker in a customer's private network, with no internet connectivity.

- **Authentication and authorization:** How to authenticate users in a controlled fashion, and authorize these users based on their IAM permissions, with no multi-tenancy.
- **Data protection:** How to encrypt data in transit and at-rest with customer provided encryption keys.
- **Auditability:** How to audit, prevent, or detect who did what at any given point in time to ensure that malicious activities are blocked or identified early.

B. Establish ML governance:

- **Traceability:** Methods to trace ML model lineage from data preparation, model development, and training iterations, as well as how to audit who did what at any given point in time.
- **Explainability and interpretability:** Methods that may help explain and interpret the trained model and obtain feature importance.
- **Model monitoring:** How to monitor your model in production to protect against data drift, and automatically react to rules that you define.
- **Reproducibility:** How to reproduce the ML model based on model lineage and the stored artifacts.

C. Operationalize ML workloads:

- **Model development workload:** How to build both automated and manual review processes in the dev environment.
- **Preproduction workload:** How to build automated CI/CD pipelines using the AWS Code* suite and AWS Step Functions.
- **Production and continuous monitoring workload:** How to combine continuous deployment and automated model monitoring.
- **Tracking and alerting:** How to track model metrics (operational and statistical) and alert appropriate users if something goes wrong.

A. Provision a secure ML environment

For your financial institution, the security of a machine learning environment is paramount. You must protect against unauthorized access, privilege escalation, and data exfiltration. There are four common considerations when you set up a secure machine learning environment:

- compute and network isolation
- authentication and authorization
- data encryption
- auditability

These considerations are detailed in the following sections.

Compute and network isolation

A well-governed and secure machine learning workflow begins with establishing a private and isolated compute and network environment. The virtual private cloud (VPC) that hosts SageMaker and its associated components, such as Jupyter notebooks, training instances, and hosting instances, should be deployed in a private network with no internet connectivity. Furthermore, these SageMaker resources can be associated with your customer VPC environment, allowing you to apply network level controls, such as security groups to govern access to SageMaker resources and control ingress and egress of data into and out of the environment. Connectivity between SageMaker and other AWS services, such as Amazon Simple Storage Service (Amazon S3), should be established using VPC endpoints or even AWS PrivateLink. Additionally, when creating a VPC endpoint, we recommend attaching an endpoint policy to further control access to specific resources for which you are connecting. Figure 3 below illustrates the recommended deployment for a SageMaker notebook instance.

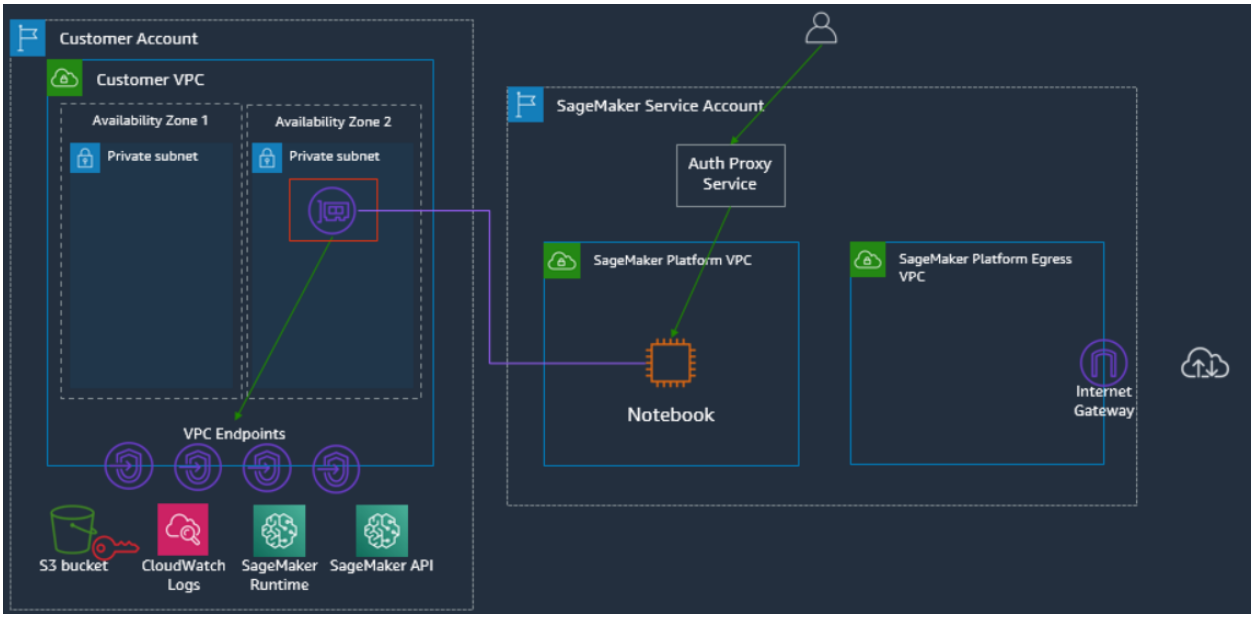


Figure 3 – Recommended deployment architecture for Amazon SageMaker notebook instance

Similarly, for training jobs and hosted models, SageMaker EC2 instances will communicate with Amazon S3 through your VPC when retrieving training data from Amazon S3. SageMaker does this by creating an elastic network interface (ENI) in your specified VPC and attaching it to the Amazon EC2 infrastructure in the service account. Using this pattern, the service gives you control over the network-level access of the services you run on SageMaker. Equally, when the SageMaker instances retrieve your trained model for hosting, they will communicate with Amazon S3 through your VPC. In such manner, you can maintain governance and control of the network communication of your SageMaker resources. For details on setting up a private network environment as illustrated above, refer to the Amazon SageMaker Workshop for Secure Networking. You can also refer to Vanguard's 2019 re:Invent presentation on Security for ML environments with Amazon SageMaker.

Authentication and authorization

After an isolated and private network environment is created, the next step is to ensure that only authorized users can access the appropriate AWS services. AWS Identity and Access Management (IAM) can help you create preventive controls for many aspects of your machine learning environment, including access to SageMaker resources, access to your data in Amazon S3, and access to API endpoints. You can access AWS using a RESTful API, and every API call is authorized by IAM. You grant explicit permissions through IAM policy documents, which specify the principal (who), the actions (API calls), and the resources (such as S3 objects) that are allowed, as well as the conditions under which the access is granted.

Accordingly, access to a SageMaker Jupyter notebook instance is also governed by IAM. We recommend that each data scientist be provided their own SageMaker notebook instance and only have access to that particular notebook. Root access to the notebook instance should be disabled. In order to open a

Jupyter Notebook instance, users will need access to the *CreatePresignedNotebookInstanceUrl* API call. This API call creates a presigned URL which is used to obtain Web UI access to the Jupyter notebook server. To secure this interface you can use IAM policy statements to wrap conditions around calling the API, such as who can invoke the API and from what IP address.

While each organization will have different authentication and access requirements, you should configure permissions in line with IAM Best Practices and your own internal policies and controls by granting least privilege access, or granting only the permissions required to perform a particular task. Role based access control (RBAC) is an approach used commonly by customers in financial services for ensuring only authorized parties have access to desired system controls. Creating roles based on job function policies, and using AWS Config to monitor and periodically audit IAM policies attached to users is a recommended best practice for viewing configuration changes over time. For details about how Millennium Management enforces compliance using IAM, read Millennium Management: Secure machine learning using Amazon SageMaker. For more information about security best practices for financial services on AWS, read Financial Services Industry Lens.

Data encryption

As machine learning environments can contain sensitive data and intellectual property, the third consideration for a secure ML environment is data encryption. We recommend that you enable data encryption both at-rest and in-transit. For at-rest encryption, such as data stored in Amazon S3 and the data stored in Amazon Elastic Block Store (Amazon EBS) volumes within the SageMaker notebook, most financial services customers require the use of customer managed customer master keys (CMK). For details around CMK on AWS KMS, see AWS Key Management Service concepts. Additionally, we recommend enabling Amazon S3 default encryption so that all new objects are encrypted when they are stored in the Amazon S3 bucket. We also recommend using Amazon S3 bucket policies to prevent unencrypted objects from being uploaded. For data in transit, all inter-network data in transit supports TLS 1.2 encryption. Finally, for data transmitted between instances during distributed training, you can enable inter-container traffic encryption. However, note that this can increase the training time, particularly for distributed deep learning algorithms.

Auditability

The fourth consideration for a well governed and secure ML environment is having a robust and transparent audit trail that logs all access and changes to the data and models such as a change in the model configuration or the hyper-parameters. AWS CloudTrail is one service that will log, continuously monitor, and retain account activity related to actions across your AWS infrastructure. CloudTrail logs every AWS API call and provides an event history of your AWS account activity, including actions taken through the AWS Management Console, AWS SDKs, command line tools, and other AWS services. Another service, AWS Config allows you to continuously monitor and record configuration changes of your AWS resources.

More broadly, in addition to the logging and audit capabilities, AWS recommends a defense in depth approach to security, applying security at every level of your application and environment. Here, AWS

CloudTrail and AWS Config can be used as detective controls responsible for identifying potential security threats or incidents. As the detective controls identify potential threats, you can set up a corrective control to respond to and mitigate the potential impact of security incidents. Amazon CloudWatch is a monitoring service for AWS resources which can trigger CloudWatch Events to automate security responses. For details on setting up detective and corrective controls, refer to the Amazon SageMaker Security Workshop.

B. Establish ML governance

ML governance includes four key aspects:

- traceability and auditability
- explainability
- real-time model monitoring
- reproducibility

The Financial Services Industry has various compliance and regulatory obligations. You should review and understand these obligations with your legal counsel, compliance personnel, and other stakeholders involved in the ML process. As an example, if Jane Smith was denied a bank loan, the lender may be required to explain how that decision was made in order to comply with regulatory requirements. If the lender is using ML as part of the loan review process, the prediction made by the ML model may need to be interpreted or explained in order to meet these requirements. In general, a ML model's interpretability or explainability refers to the ability for people to understand and explain the processes that the model uses to arrive at its predictions. It is also important to note that many ML models make predictions of a likely answer, and not the answer itself. It may be appropriate to have additional trained human review of predictions made by such models before any action is taken. The model may also need to be monitored, so that if the underlying data based on which this prediction was made changes, the model will be periodically retrained on new data. Finally, the ML model may need to be reproducible, such that if the steps leading to the model's output were retraced, the model outputs should not change. In the section below, we will provide examples of things you may need to consider when you are establishing a ML governance regime and showcases how you can operationalize your selected governance regime.

Traceability

Effective model governance requires a detailed understanding of the data and data transformations used in the modeling process, in addition to continuous tracking of all model development iterations. It is important to know: which dataset was used, what types of transformations it went through, what type of model was built, and its hyperparameters. Additionally, any post-processing steps such as batch jobs to remove biases from predictions during batch inference need to be tracked. If a model is promoted to online inference, model performance and model predictions need to be tracked.

Within the machine learning workflow, there are various steps where transformations are applied to the raw data, or to the dataset that is part of a data lake. This starts with data exploration, where missing, duplicate, or inconsistent data points are identified and treated. It continues with data pre-processing and feature engineering that help with new feature creation, normalization, and standardization. Finally, just prior to start iterating with different frameworks, ML models or parameterization, it ends with the preprocessed data split into three sets: training, testing, and validation. Depending on whether the overall dataset is imbalanced, additional considerations such as stratified sampling, oversampling, or under-sampling may be required. Well-governed machine learning workflows provide clear lineage of all transformations applied to the data as well as the post-transformation data output. The following figure shows some of key building blocks for tracking data and model lineage.

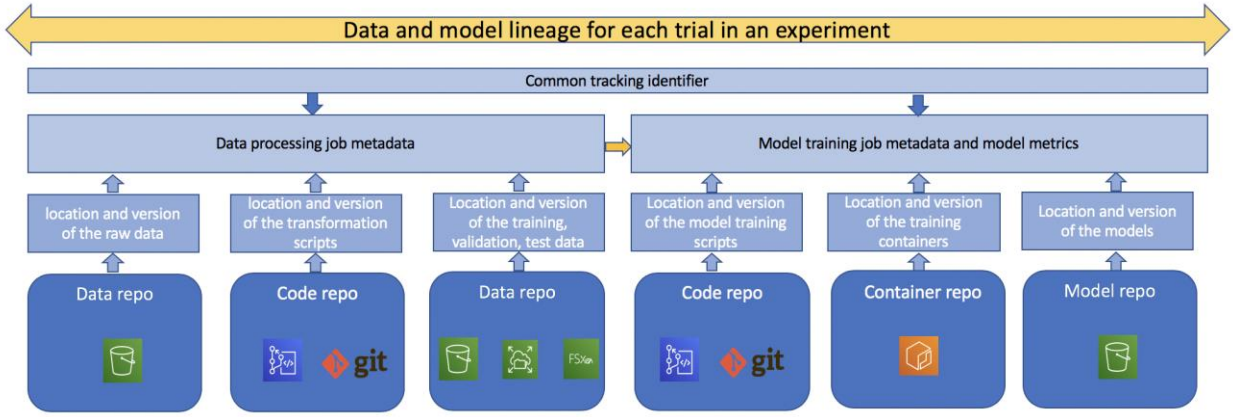


Figure 4 – Data and model lineage

With Amazon SageMaker Experiments, you can track ML artifacts and have clear lineage across datasets, algorithms, hyperparameters, metrics, in addition to any other ML workflow steps, such as data pre-processing or post-training model evaluation. By using Git repository, you can add model versioning to the experiments, because you can pull the latest commit, the user IP, the commit ID, and the timestamp of each model version. If necessary, you can also enforce that all library dependencies are downloaded and versioned in Amazon S3, in order to hold an immutable snapshot and thus ensure backward compatibility at future date. In addition, with SageMaker Experiments, you can search and browse your current and past experiments, compare experiments, and identify best performing models in a secure fashion, because all lineage metadata is stored in an encrypted fashion.

The process to track all artifacts is straightforward. You have to first create an experiment, as shown in the code snippet below, to track all the ML workflow components, and the iterations within them. SageMaker Experiments should be seen as a way to organize your data science work. You can create experiments to organize all of your model development work for:

- a business use case you are addressing (for example, create an experiment named "CreditCardDefault" as shown in the example that follows)

- a data science team that owns the experiment (for example, create an experiment named "CreditCardTeam-ProjectName")
- a specific data science and ML project. Think of it as a "folder" for organizing your "files".

The following is an example script for creating an SageMaker experiment:

```
cc_experiment = Experiment.create(
    experiment_name=f"CreditCardDefault-{int(time.time())}",
    description="Predict credit card default from payments data",
    sagemaker_boto_client=sm)
print(cc_experiment)
```

You can then start tracking parameters from the pre-processing/feature engineering steps, such as the training data location, the validation data location, as well as, the pre-processing parameters such as the train/test split ratio. On the latter, we could also add normalization mean, normalization standard deviation, and any other indicator of what feature engineering steps like scaling, one-hot encoding, etc. the data went through.

The following is an example script for tracking parameters used in the pre-processing pipeline.

```
with Tracker.create(display_name="Preprocessing", sagemaker_boto_client=sm) as
tracker:
    tracker.log_parameters({
        "train_test_split_ratio": 0.2
    })
    # we can log the s3 uri to the dataset we just uploaded
    tracker.log_input(name="ccdefault-raw-dataset", media_type="s3/uri",
value=raw_data_location)
    tracker.log_input(name="ccdefault-train-dataset", media_type="s3/uri",
value=train_data_location)
    tracker.log_input(name="ccdefault-test-dataset", media_type="s3/uri",
value=test_data_location)
```

Subsequently, you can start tracking each training job by creating a trial. In order to enrich the trial with the parameters from the pre-processing job, you have to also create a trial component on the tracker you instantiated, and add it to the trial. In the code snippet below, we named the trial **cc-fraud-training-job-** along with a timestamp, added as TrialComponent the "Preprocessing" job, and thus its parameters. On each training trial, SageMaker Experiments will track the Docker image of the model that was used, the output bucket of the model artifact location, and the hyperparameters as shown during the following model fit.

```
account = sess.boto_session.client('sts').get_caller_identity()['Account']
```

```
image = '{}.dkr.ecr.{}.amazonaws.com/sagemaker-xgboost:latest'.format(account,
region)
output_bucket = 'output-bucket-name'
preprocessing_trial_component = tracker.trial_component

trial_name = f"cc-fraud-training-job-{int(time.time())}"
cc_trial = Trial.create(
        trial_name=trial_name,
        experiment_name=cc_experiment.experiment_name,
        sagemaker_boto_client=sm
    )

cc_trial.add_trial_component(preprocessing_trial_component)
cc_training_job_name = "cc-training-job-{}".format(int(time.time()))

xgb = sagemaker.estimator.Estimator(image,
                                    role,
                                    train_instance_count=1,
                                    train_instance_type='ml.m4.xlarge',
                                    train_max_run=86400,

output_path='s3://{}/{}/models'.format(output_bucket, prefix),
                                    sagemaker_session=sess,
                                    train_use_spot_instances=True,
                                    train_max_wait=86400,
                                    subnets = ['subnet-0f9914042f9a20cad'],
                                    security_group_ids = ['sg-089715a9429257862'],
                                    train_volume_kms_key=cmk_id,
                                    encrypt_inter_container_traffic=False) # set to
true for distributed training

xgb.set_hyperparameters(max_depth=5,
                        eta=0.2,
                        gamma=4,
                        min_child_weight=6,
                        subsample=0.8,
                        verbosity=0,
                        objective='binary:logistic',
                        num_round=100)

xgb.fit(inputs = {'training':'s3://' + train_data_location},
       job_name=cc_training_job_name,
        experiment_config={
            "TrialName": cc_trial.trial_name,
```

```
            "TrialComponentDisplayName": "Training",
        },
        wait=True,
    )
time.sleep(2)
```

In order to review all the components and parameters in a trial, SageMaker Experiments lets you search based on the trial name, as shown in the following code example.

```
# Present the Model Lineage as a dataframe
sagemaker.session import Session
sess = boto3.Session()
lineage_table = ExperimentAnalytics(
    sagemaker_session=Session(sess, sm),
    search_expression={
        "Filters":[{
            "Name": "Parents.TrialName",
            "Operator": "Equals",
            "Value": trial_name
        }]
    },
    sort_by="CreationTime",
    sort_order="Ascending",
)
lineagedf= lineage_table.dataframe()

lineagedf
```

You can trace the details of each component of the feature engineering and training process, and showcase them as the following data frame. Once all workflow artifacts are saved in Amazon S3, there should also be Amazon S3 object-lock, and versioning enabled in order to be able to avoid any overwrite and thus traceability loss.

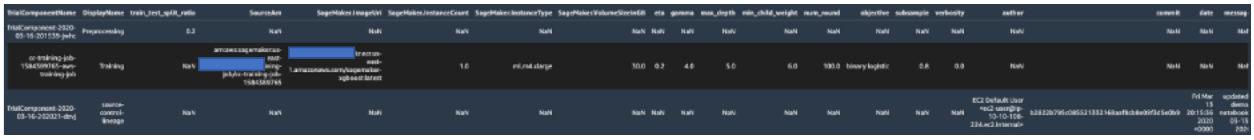
	TrialComponentName	DisplayName	train_test_split_ratio	SourceArn	SageMaker.ImageUri	SageMaker.InstanceCount	SageMaker.InstanceType	SageMaker.VolumeSizeInGB	eta	gamma	max_depth	min_child_weight	num_round	objective
0	TrialComponent-2020-03-16-201539-jwhc	Preprocessing	0.2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	cc-training-job-1584389765-aws-training-job	Training	NaN	arn:aws:sagemaker:us-east-1:[redacted]:training-job/cc-training-job-1584389765	[redacted].dkr.ecr.us-east-1.amazonaws.com/sagemaker-xgboost:latest	1.0	ml.m4.xlarge	30.0	0.2	4.0	5.0	6.0	100.0	binary:logistic

Similarly, you can use model tracking capability to verify that a specific dataset was used in a training job for an audit or to verify compliance. To achieve that, you search for the URL that represents its location in Amazon S3. Model tracking capability returns the training jobs that used the dataset that you specify. If your search doesn't return the dataset (the result is empty), the dataset wasn't used in a training job. An empty result confirms, for example, that a holdout dataset wasn't used.

```
def get_codecommit(commit_id):
    codecommitclient = boto3.client('codecommit')

    reponame =
codecommitclient.list_repositories()['repositories'][0]['repositoryName']

    return codecommitclient.get_commit(repositoryName=reponame,
    commitId=commit_id
    )

# Below you will need to navigate to CodeCommit to obtain the corresponding commit
IDs if you choose to commit your code.
# If you only commit your code once, then use the same repo name and CommitIDs for
sclineage and endpointlineage.

sclineage = get_codecommit('1796a0a6972c8df97fd2d279557a6f94cfe91eae') # Enter your
CommitID here

endpointlineage = get_codecommit('a6df1799849f52295864754a8f1e30e604fefd00') #
Enter your CommitID here

with Tracker.create(display_name="source-control-lineage",
sagemaker_boto_client=sm) as sourcetracker:
    sourcetracker.log_parameters({
        "commit": sclineage['commit']['commitId'],
        "author":sclineage['commit']['author']['email'],
        "date":sclineage['commit']['author']['date'],
        "message":sclineage['commit']['message'].replace('-', '_').split('\n')[0]
    })


with Tracker.create(display_name="prod-endpoint-lineage", sagemaker_boto_client=sm)
as endtracker:
    endtracker.log_parameters({
        "commit": endpointlineage['commit']['commitId'],
        "author":endpointlineage['commit']['author']['email'],
        "date":endpointlineage['commit']['author']['date'],
        "message":endpointlineage['commit']['message'].replace('-',
'_').split('\n')[0]
    })
    endtracker.log_input(name="endpoint-name", value='cc-training-job-1583551877')

cc_trial.add_trial_component(sourcetracker.trial_component)
cc_trial.add_trial_component(endtracker.trial_component)
```

```
# Present the Model Lineage as a dataframe
from sagemaker.session import Session
sess = boto3.Session()
lineage_table = ExperimentAnalytics(
    sagemaker_session=Session(sess, sm),
    search_expression={
        "Filters":[{
            "Name": "Parents.TrialName",
            "Operator": "Equals",
            "Value": trial_name
        }]
    },
    sort_by="CreationTime",
    sort_order="Ascending",
)
lineagedf= lineage_table.dataframe()

lineagedf
```

To trace a model's lineage, you can trace the model's name from the relevant endpoint, or use one of the trial components' name. You can then use either of these names to search for the training, its parameters, the pre-processing job, and the model version that were part of the deployed endpoint with the TrialComponentName API as shown on the last part of the above code snippet. Below, you can see the data frame with all the different components of the lineage. Important to note that the last row showcases the commit ID, the timestamp, the author, and the commit message of the model version that was deployed.



To help simplify the reproducibility process, Amazon SageMaker Studio, now provides notebook sharing functionality so that each team member can easily share their work, and let their teammates reproduce their model quickly.

Explainability and interpretability

Some models, like logistic regression and decision trees, are fairly simple and straightforward and it is easier to interpret how the model came up with its output. But as more features are added or more complicated machine learning models are used, interpretability becomes more difficult. When using an algorithm's predictions in connection with financial services decisions, it's important to know which features the model did or did not take into account, and in some cases, it may be important to have appropriate human review of such predictions before taking action. Within financial services, model

users may need to understand each model's limitations, intent, and output, including understanding the factors that contribute to the model's outcomes.

There are a variety of different methods for helping explain machine learning models: partial dependency plots, building surrogate models on either the entire dataset (global) or a subset of the data (local), Quantitative input influence, LIME (Local Interpretable Model-Agnostic Explanation) and SHAP (SHapley Additive exPlanations) to name a few. Explainability in neural network models continues to be an active topic of research, and it is important to stay educated on this important topic.

While explainability in machine learning remains an active research topic, SHAP has emerged as a popular unifying method for extracting feature importance and is detailed here within the context of our dataset. The goal of SHAP is to explain the prediction by computing the contribution of each feature to the prediction by producing two models: one with the feature included and another with the feature withheld. The difference in predictions on a given sample is related to the importance of the feature. SHAP values extend this idea beyond linear models or locally independent features by accounting for the interdependency of features by averaging over all permutations of the order in which features are added to a model.

SageMaker enables you to obtain SHAP values for popular models such as LightGBM, XGBoost or simple deep learning models. We first need to ensure that the SHAP library is pre-installed in our local Jupyter working environment. This library can be provisioned for data scientists from the outset via lifecycle configuration scripts. If the development VPC, which the data scientists are using for model training and development is configured to not have internet access, the library can be downloaded from a local PyPI server or using a pip mirror from a separate VPC which does have internet access enabled such as a shared services account.

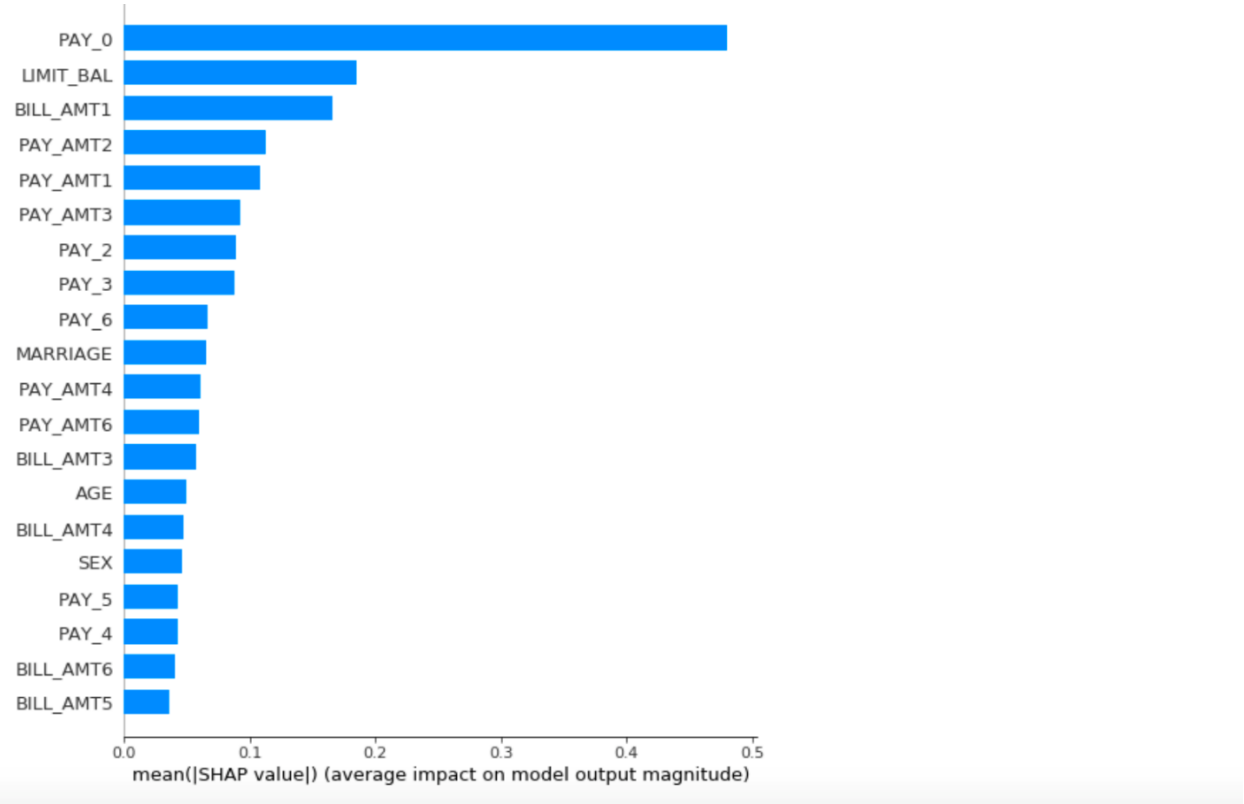
Once imported, the trained model object can be copied over from Amazon S3 into the local environment. The following function can call the SageMaker Experiments Trail and unpack the pickled model object. Here our training container stores the trained XGBoost model object as a pickle file called `xgboost.pkl`.

```
def download_artifacts(job_name, local_fname):
    ''' Given a trial name in a SageMaker Experiment, extract the model
file and download it locally'''
    sm_client = boto3.Session().client('sagemaker')
    response =
sm_client.describe_trial_component(TrialComponentName=job_name)
    model_artifacts_full_path =
response['OutputArtifacts']['SageMaker.ModelArtifact']['Value']

    p = re.compile('(?<=s3://).*?/')
    s = p.search(model_artifacts_full_path)
    object_name_start = s.span()[1]
    object_name = model_artifacts_full_path[object_name_start:]
    bucket_name = s.group()[:-1]
```

```
    s3 = boto3.client('s3')
    s3.download_file(bucket_name, object_name, local_fname)

def unpack_model_file(fn):
    # Unpack model file
    _tar = tarfile.open(fn, 'r:gz')
    _tar.extractall()
    _fil = open('xgboost-model', 'rb')
    _model = joblib.load(_fil)
    print(_model)

    return _model
```

Once the model object is extracted, we can use the SHAP library to compute and plot the feature importance on the entire training dataset or a subset of the dataset. While the former approach obtains global feature importance across the entire dataset, local importance can also be derived that can then be published as part of a detailed report for risk management and model governance. For example, if the model predicts that a customer is likely to default on their credit card payments, you may need to reverse the model's decision by identifying which features led to the decision, and take corrective action if necessary.

```
shap.summary_plot(shap_values, traindata.drop(columns =['Label']))
```



Note that for tree-based models such as Gradient Boosted trees, the relative importance of a feature can also be obtained by attributing whether a particular split by that feature contributes to the predictive accuracy of the model, weighted by the number of observations split on that node. This Gini impurity metric, averaged across all the trees can also be used as an initial metric for feature importance. It is not uncommon to report multiple different approaches to calculating feature importance. Since this metric can be natively obtained from XGBoost APIs, and does not generalize to other non-tree-based models unlike SHAP values, we do not discuss it further here.

In addition to manually using SHAP values for identifying feature importance *post*-training, SageMaker recently launched Amazon SageMaker Debugger, a fully managed service for debugging models during training by visualizing tensors for deep learning models, as well as evaluating built-in and custom rules to check training jobs and helping catch errors. SageMaker Debugger enables customers to log feature importance and SHAP values during model training iterations, that can then be visualized later in Amazon SageMaker Studio. For more information about SageMaker Debugger, read How Debugger Works. Finally, we reiterate that while we discuss SHAP here from an implementation perspective, the field of explainability is very nascent and under development, and the efficacy of models such as SHAP for explainability is still subject to debate.

In addition to explainability, considerations of fairness and safety may also be important, which we briefly discuss below. Fairness is a very complex and broad ranging topic and a full discussion is outside the scope of this paper; in the context of this paper we discuss fairness as the evaluation for bias and the understanding of any bias in a model. Considerable effort needs to be put in place to ensure that the

training data is fair and representative, and each dataset that is used for training should be reviewed for bias. Companies implement model bias policies, bias checking methods and monitor results on a regular basis. There are different types of bias. Here are some examples:

- **Sampling bias**—Occurs when the training data does not accurately represent real-world scenarios. Sample bias can be reduced or eliminated by training your model on all potential scenarios.
- **Exclusion bias**—Happens as a result of excluding some features from your dataset usually under the umbrella of cleaning our data. This happens as a result of deletion of features based on the developer's understanding of the data. Exclusion bias can be reduced or eliminated by appropriate research prior to elimination of features and also by procuring business Subject Matter Expert (SME) consensus on features that are planned to be eliminated.
- **Cultural or stereotype bias**—Bias related to issues including appearances, social class, status, and gender. Certain types of bias may be reduced by understanding and avoiding the cultural and stereotype biased results. In the previous example, this could include ignoring statistical relationships between occupation and gender. Key mitigating factors include using diverse teams who are educated on and aware of these issues, and training the model with appropriate datasets that minimize cultural and stereotype bias
- **Measurement bias**—This happens when there's an issue with the device used to observe or measure. Systematic value distortion occurs due to an issue with the device. Measurement bias can be reduced or eliminated by leveraging multiple devices to avoid device distortions and incorporating data checks by humans or Amazon Mechanical Turk to compare the device outputs.

You should work with your risk, legal and compliance teams to assess legal, ethical, regulatory, and compliance requirements for, and implications of, building and using ML systems. This may include vetting legal rights to use data and/or models in the system, as well as determining applicability of laws covering issues such as privacy, biometrics, anti-discrimination, and other considerations specific to financial services use cases.

Model monitoring

Model monitoring is the next phase of a data science project's lifecycle after a machine learning model is deployed to production. Model monitoring measures model performance against the actual outcomes, and helps ensure the deployed model works as expected in a real-world environment. There may be regulatory guidelines around model monitoring. Model performance data is often used in the audit report, or as an early warning signal so that proper contingency plans can be executed. It can also be used as guidelines for model re-training or model decommissioning.

One might ask why model monitoring is required if it is already verified and validated in a testing environment. That's because machine learning model performance could degrade when the statistical nature of data such as mean, average and correlation changes gradually, or the statistical relationship

between the independent and dependent variables changes over time. These are called data drift and concept drift respectively. Drifts could happen from various sources and there are different approaches for drift detection. One way to detect data drift is to compare the schemas and the computed statistical distributions between training and new data. Changes such as mean and standard deviation shift or changes in data field types could be indicators for data drift. Prediction distribution can also be monitored and computed over time as a way to detect model drift. In addition, if ground truth labels can be collected for new data, then comparing prediction results with the ground truth is another way to monitor model performance degradation.

Detecting data drift with SageMaker

Detecting data drift is a challenging engineering and science task. You would need to capture data received by the models, run all kinds of statistical analysis to compare that data to the training set, define rules to detect drift, and send alerts if it happens. And every time you release a new model, these steps need to be repeated. The SageMaker Model Monitor helps simplify data drift detection with the following steps/capabilities:

1. **Capture incoming request data and model output** – SageMaker Model Monitor captures incoming request data and prediction output. The captured data is stored in your Amazon S3 bucket, and it is enriched with metadata such as content type and time stamp. As a user, you have to ability to configure how much data is captured.

```
from sagemaker.model_monitor import DataCaptureConfig
endpoint_name = 'DEMO-xgb-churn-pred-model-monitor-' + strftime("%Y-%m-%d-%H-%M-
%S", gmtime())
print("EndpointName={}".format(endpoint_name))

data_capture_config = DataCaptureConfig(
                        enable_capture=True,
                        sampling_percentage=100,
                        destination_s3_uri=s3_capture_upload_path)

predictor = model.deploy(initial_instance_count=1,
                instance_type='ml.m4.xlarge',
                endpoint_name=endpoint_name,
                data_capture_config=data_capture_config)
```

2. **Create a monitoring baseline** – SageMaker Model Monitor can create baseline using the data that was used for training the model. It will infer a schema for the input data such as type and completeness of each feature, and compute feature statistics as part of the baseline computation.

```
from sagemaker.model_monitor import DefaultModelMonitor
```

```
from sagemaker.model_monitor.dataset_format import DatasetFormat

my_default_monitor = DefaultModelMonitor(
    role=role,
    instance_count=1,
    instance_type='ml.m5.xlarge',
    volume_size_in_gb=20,
    max_runtime_in_seconds=3600,
)

my_default_monitor.suggest_baseline(
    baseline_dataset=baseline_data_uri+'/training-dataset-with-header.csv',
    dataset_format=DatasetFormat.csv(header=True),
    output_s3_uri=baseline_results_uri,
    wait=True
)
```

3. **Monitor an endpoint** – You can establish a monitoring schedule using SageMaker Model Monitor to inspect the collected data against the baseline. There are a number of built-in rules such as violations checks applied to the results, and the results are pushed to Amazon S3.

The results contain statistics and schema information on the data received during the latest time frame, as well as any violations that were detected. SageMaker Model Monitor also emits per-feature metrics to Amazon CloudWatch, which can be used to set up alerts. The summary metrics are also available in the SageMaker Studio for easy access. The reports and alerts can help identify data quality issues quickly, so necessary remediation steps can be taken to address any issues.

Detecting model drift with SageMaker

Data drift detection is the first line of defense against model performance degradation. If ground truth labels can be collected for new data, then SageMaker Batch Transform can be used to run batch inference periodically for computing model metrics. The following steps list how to use SageMaker Batch Transform to run and compute the model metrics:

1. **Prepare data**—Prepare model feature vectors using the labeled dataset and remove the actual labels from the inputs, and save the request files in an Amazon S3 bucket.
2. **Run batch transform**—Run SageMaker batch transform against the target model in SageMaker. The output file will be saved in an Amazon S3 bucket.
3. **Compute metrics**—Run model evaluation scripts using the output files and the original ground truth labels to compute the new metrics and compare against the original validation metrics. If the model performance metric changes cross a defined threshold, then remediation steps should be considered.

There are many different options to run the preceding steps. For example, AWS Lambda functions can be used to prepare the data, kick off the batch transform, and compute the model metrics, and thus a potential model drift.

Reproducibility

Reproducibility refers to the ability to re-create the same ML model at a later date, or by a different stakeholder while reducing the randomness involved in the process. Customers may need to be able to reproduce and validate the end-to-end development process, and provide the necessary artifacts and methodologies that were used to create the final model, in order to meet regulatory requirements. While a full discussion of model reproducibility is outside the scope of this paper, we focus on how to easily identify the source data used to develop the model, the actual model selection, its parameters, as well as the artifacts of the final model when leveraging SageMaker.

With SageMaker Experiments and your version control system (Git, CodeCommit, or Artifactory), as shown in the Traceability section, you can track all the lineage of the model version that was deployed, and thus re-run the same pre-processing pipeline, on the same data, and then use the same model with the same hyper-parameters to get almost the model. It is important to remember that due to the randomness entailed in the optimizer implementing stochastic gradient descent, exactly same processes on the same artifacts, might produce slightly different models which may have slightly different prediction outcomes. In our case, randomness is completely eliminated by all the steps except that of training the model, where although we use the exact same input, model type, and parameters, due to the seed sequence and the stochastic nature of the optimizer, we might notice minor differences in the optimal weights of the final model.

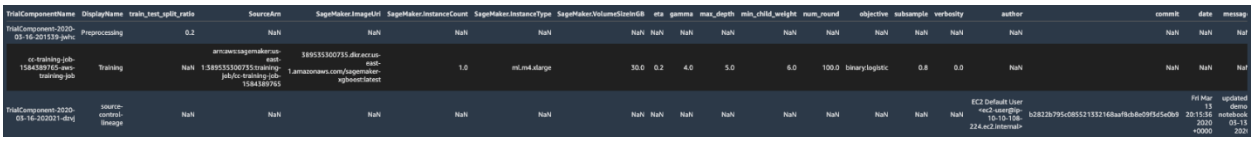
The following is an example script to keep track of commits from Git.

```
def get_codecommit(commit_id):
    codecommitclient = boto3.client('codecommit')

    reponame =
    codecommitclient.list_repositories()['repositories'][0]['repositoryName']

    return codecommitclient.get_commit(repositoryName=reponame,
    commitId=commit_id
    )

# Below you will need to navigate to CodeCommit to obtain the corresponding
commit IDs if you choose to commit your code.
# If you only commit your code once, then use the same repo name and CommitIDs
for sclineage and endpointlineage.

sclineage = get_codecommit('1796a0a6972c8df97fd2d279557a6f94cfe91eae') # Enter
your CommitID here
```

```
endpointlineage = get_codecommit('a6df1799849f52295864754a8f1e30e604fefd00') #
Enter your CommitID here

with Tracker.create(display_name="source-control-lineage",
sagemaker_boto_client=sm) as sourcetracker:
    sourcetracker.log_parameters({
        "commit": sclineage['commit']['commitId'],
        "author":sclineage['commit']['author']['email'],
        "date":sclineage['commit']['author']['date'],
        "message":sclineage['commit']['message'].replace('-',
'_').split('\n')[0]
    })


with Tracker.create(display_name="prod-endpoint-lineage",
sagemaker_boto_client=sm) as endtracker:
    endtracker.log_parameters({
        "commit": endpointlineage['commit']['commitId'],
        "author":endpointlineage['commit']['author']['email'],
        "date":endpointlineage['commit']['author']['date'],
        "message":endpointlineage['commit']['message'].replace('-',
'_').split('\n')[0]
    })
    endtracker.log_input(name="endpoint-name", value='cc-training-job-
1583551877')

cc_trial.add_trial_component(sourcetracker.trial_component)
cc_trial.add_trial_component(endtracker.trial_component)

# Present the Model Lineage as a dataframe
from sagemaker.session import Session
sess = boto3.Session()
lineage_table = ExperimentAnalytics(
    sagemaker_session=Session(sess, sm),
    search_expression={
        "Filters":[{
            "Name": "Parents.TrialName",
            "Operator": "Equals",
            "Value": trial_name
        }]
    },
    sort_by="CreationTime",
    sort_order="Ascending",
```

```
)
lineagedf= lineage_table.dataframe()

lineagedf
```

As shown in the previous code snippet, if you want to achieve model lineage and the ability to reproduce your models, you must enforce usage of the Tracker API for both the pre-processing parameters, and code versioning. In addition to the TrialComponent API, which will track the model, its parameters, and the datasets natively, customers have to ensure that the Tracker API is always logged for all other steps. Next, consider Git/CodeCommit best practices, with regards to branching, committing, and properly commenting to enable the Tracker to accurately include the author, the commitID, the date, and the message of the commit as shown in the bottom right section of the following image.



C. Operationalize ML workloads

In this final section, we discuss some best practices around operationalizing FSI ML workloads. We begin with a high-level discussion and then discuss a specific architecture that leverages AWS native tools and services. In addition to the process of deploying models, or what in traditional software deployments is referred to as CI/CD (continuous integration/deployment), deploying ML models into production for regulated industries may have additional implications from an implementation perspective.

The following diagram captures some of the high-level requirements that an enterprise ML platform might have to address guidelines around governance, auditing, logging, and reporting:

- A data lake for managing raw data as well as associated metadata
- A feature store for managing machine learning features as well as associated metadata (mapping from raw data to generated features such as one-hot encodings, scaling transformations etc.)
- A model and container registry containing trained model artifacts and associated metadata (hyperparameters, training times, dependencies etc.)
- A code repository (Git/CodeCommit/Artifactory etc.) for maintaining and versioning source code
- A pipeline registry to version and maintain training and deployment pipelines
- Logging tools for maintaining access logs
- Production monitoring and performance logs

- Tools for auditing and reporting

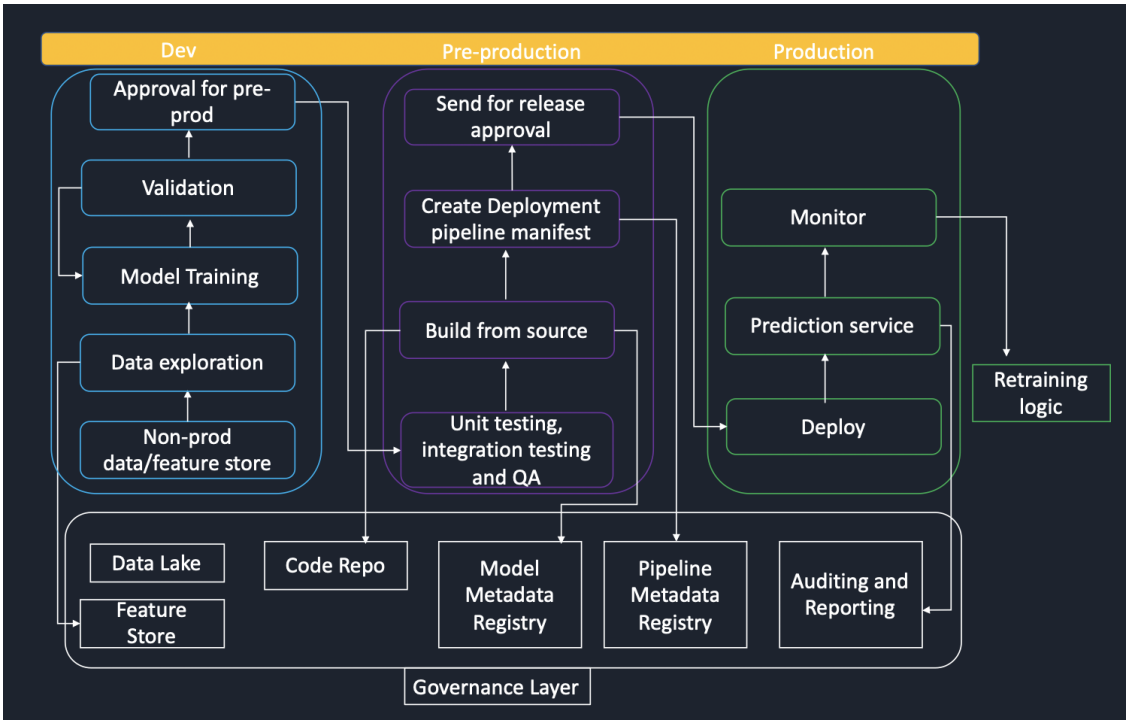


Figure 5 – ML DevOps workflow

Below we illustrate a specific implementation that leverages AWS native tools and services. While there are several scheduling and orchestration tools in the market such Airflow, Jenkins etc., for concreteness, we will focus the rest of the discussion on AWS Step Functions.

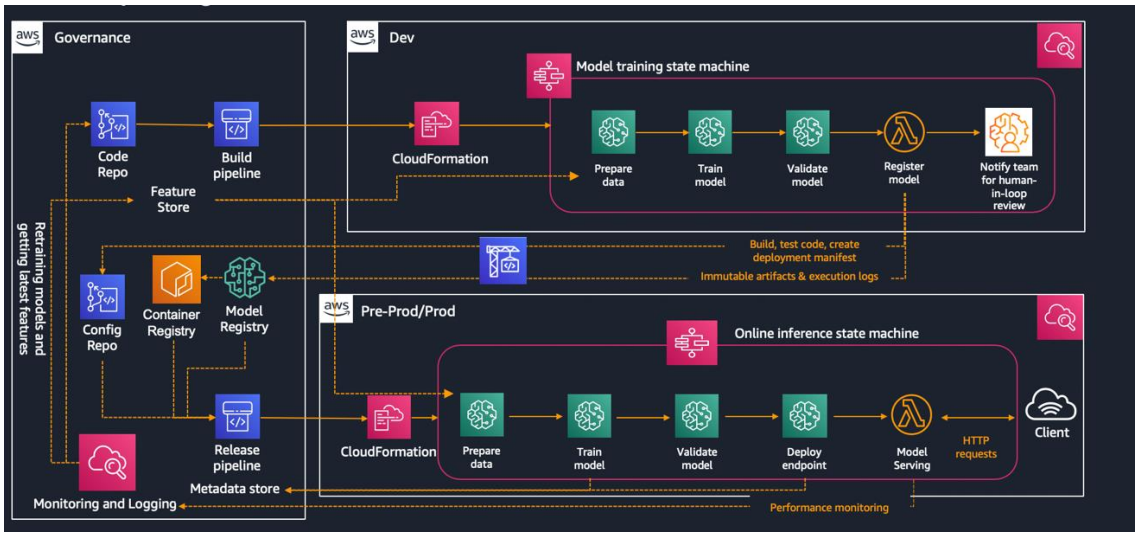


Figure 6 – ML DevOps with AWS native tools

Model development workload

Let's start with the dev environment. Here, data scientists can analyze and validate raw data, generate features, experiment with trained models, and validate these models in test scenarios. This can be done either with or without production data, depending on the specific nature of the use case. Although we have depicted the process as linear, this process is often highly cyclical and experimental.

To speed up experimentation, data scientists can use the AWS Step Functions Data Science SDK. This open source library allows you to author a training pipeline that can be triggered either when new features or data is pushed to the training data repository or when a new model is added. AWS Step Functions is a fully serverless orchestration tool for deploying pipelines. The Data Science SDK is a Python SDK that can be used for authoring state machines and provides convenient, easy-to-use APIs for machine learning pipeline orchestration. The SDK automatically calls an AWS Lambda function to deploy the model container, train the model on the data, create an endpoint or use offline testing through a batch transform job to test the model performance.

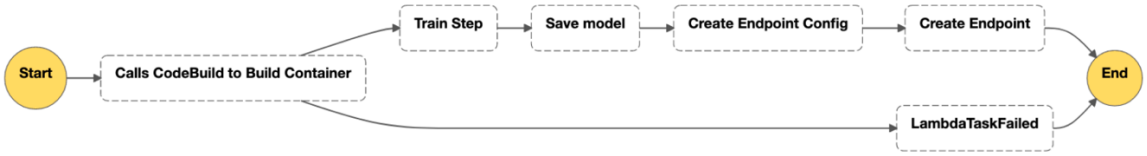


Figure 7: Dev environment endpoint creation workflow

The lines of code below show how you can use the Data Science SDK to launch a model training job and create a model from the training artifacts.

```
training_step = steps.TrainingStep(
    'Train Step',
    estimator=xgb,
    data={
        'train': sagemaker.s3_input(train_s3_file, content_type='csv'),
        'validation': sagemaker.s3_input(validation_s3_file, content_type='csv')},
    job_name=execution_input['JobName']
)
```

```
model_step = steps.ModelStep
    ('Save
    Model',
    model=training_step.get_expected_model(),
```

```
    model_name=execution_input['ModelName']).
    result_path='$.ModelStepResults'
)
```

As discussed in previous sections, SageMaker Experiments can be used to capture any metadata and model lineage related to training. Once our experiments are complete, many financial services companies might have a manual or human-in-the-loop process for validating and reviewing the trained model, ensuring that it meets their requirements around model explainability and interpretability, any model risk is properly documented, as well as an internal audit on whether the development process has been properly documented. This review may also need to be conducted by independent parties who are not responsible for the actual model development.

From a security and access management perspective, the development environment requires the most manual human interactions and may often be less restrictive since it doesn't involve production data. However, in cases where the development environment does require production data access, special attention is merited. One approach is to create micro-accounts, where only the teams and users who are authorized are granted access and strict boundaries are enforced. In addition, special approval workflows may need to be created before access can be granted to production data for model development.

Pre-production workload

Once the model has been adequately reviewed, the model is ready to be passed on to pre-production or production for deployment by a DevOps team. For simplicity, we have combined pre-production and production environments here. However, you might choose to separate them.

DevOps teams will conduct:

- Integration testing to ensure that the model can be incorporated into the existing business workflow, and may build any serving logic that may be required.
- QA testing such as performance testing, comparing model performance on historical production data (back-testing) versus on training data, or A/B testing against previously deployed models.
- Stress testing where models are tested against various scenarios: for example, a trading algorithm may be tested in conditions of low and high market volatility to understand model risk.

Finally, the trained model source code, model features, model artifacts and associated metadata, builds such as containers and associated metadata can be stored in the appropriate registries.

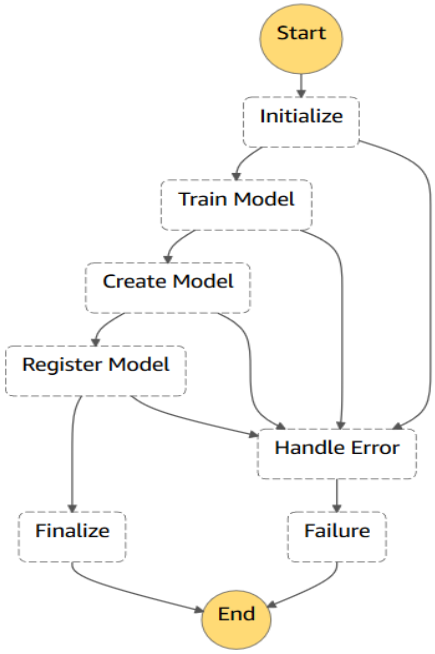


Figure 8 – Testing model workflow

```
check_accuracy_step = steps.states.Choice(
    'Accuracty > 90%'
)
```

You can use AWS CodePipeline to create a manifest that picks up the tested model, requiring a human approval flow to move the model to production. AWS CodePipeline integrates with infrastructure-as-code tools such as AWS CloudFormation to provision the resources needed to deploy the model.

Production and continuous monitoring workload

Once the production pipeline is ready, AWS CodePipeline can run the AWS CloudFormation scripts required to create the underlying infrastructure and AWS Step Functions state machine needed to deploy the model to a production endpoint, as shown in the Figure 9. AWS CodePipeline can also be used to launch the AWS Step Functions workflow needed to deploy the model into production as shown below, by pulling code from AWS CodeCommit. If the model has already been validated against production data in the pre-production stage, it can be deployed directly to an endpoint as shown in the Figure 7; otherwise, a second retraining step may be needed against production data. This helps minimize any skew between training and deployed model performance.

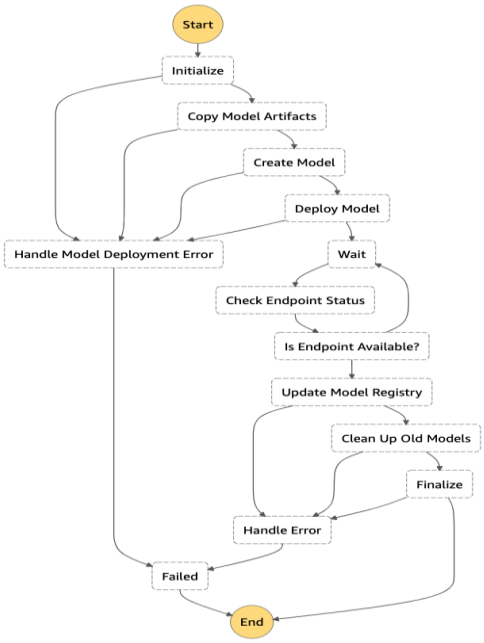


Figure 9: AWS Step Functions state machine

In addition to continuous deployment, a key element of deploying ML workflows is monitoring. This includes both performance monitoring of the endpoint and monitoring the model itself for drift as described in the earlier sections. In previous sections, we have described how to use SageMaker Model Monitor to establish model monitoring against deployed endpoints.

Once continuous monitoring has been set up, it is essential to retrain models often and set up triggers for model re-training. These may include data or label/concept drift triggers, model performance drift triggers, inclusion of new data or new features in the feature store, etc.; and can be event driven or scheduled. AWS Lambda can then be used to trigger AWS Step Functions pipelines above for retraining models.

Customers may also use human-in-the-loop workflow to periodically send model outputs to a human reviewer. The outputs of the review process can be re-incorporated back into the training data as true ground truth labels. Using SageMaker's Augmented AI capability, customers can easily create human-in-the-loop workflows using external or private workforce.

Conclusion

This whitepaper detailed some considerations for provisioning and operationalizing a well-governed ML workflow. An ML workflow is an orchestrated effort across many different stakeholders. You should work with your compliance team to evaluate how specific regulatory guidelines and your company's own policies may impact your use of machine learning.

For more information about implementation details, see the Secure and compliant ML workflows with Amazon SageMaker video in the SageMaker Developer Resources.

Contributors

Contributors to this document include:

- Stefan Natu, Sr. Machine Learning Solutions Architect
- David Ping, Principal Machine Learning Solutions Architect
- Kosti Vasilakakis, Sr. Business Development Manager, AWS Machine Learning
- Alvin Huang, Capital Markets Business Development
- Paul Jeyasingh, Advisory Consultant, AWS Professional Services

Document Revisions

Date	Description
July 2020	First publication