upGrad

Introduction to AWS SageMaker

MLOPs Principles

The following are the principles of MLOps:

- Code, artifact, and experiment tracking
- Cross-team collaboration
- Reproducible results
- Development/Production symmetry
- Continuous integration
- Continuous deployment
- Continuous training
- Module health check

These are major principles of MLOps that remain constant in all the problems we will solve, even if we change the tools or the use case. Hence, it is very important to remember the principles.

MLOps Maturity Levels

The different versions of MLOps are as follows:

- MLOps V1.0: Manually build, train, tune and deploy models
- MLOps V2.0: Manually build and orchestrate model pipelines
- MLOps V3.0: Automatically run pipelines when new data arrives or code changes (deterministic triggers such as GitOps)
- MLOps V4.0: Automatically run pipelines when models start to decay (statistical triggers, such as drift, bias and explainability)

Different companies use different versions of MLOps as per their requirements. It basically depends on the use case or the problem statement as to which version of MLOps should be used.

In each version of MLOps framework, an extra automation level is added.

Startups normally are at version 1.0, and the companies such as Google and Amazon are achieving version 4.0 of the MLOps framework.

Ways to Implement MLOps

There are two ways to implement MLOps:

1. Open source
2. Managed services

Some of the open source tools available in the market are shown below.

The major and most widely used cloud managed services that can be used for MLOps practices are shown below.

Vertex AI

Azure Machine Learning, Amazon SageMaker and Vertex AI are provided by Microsoft, Amazon and Google, respectively.

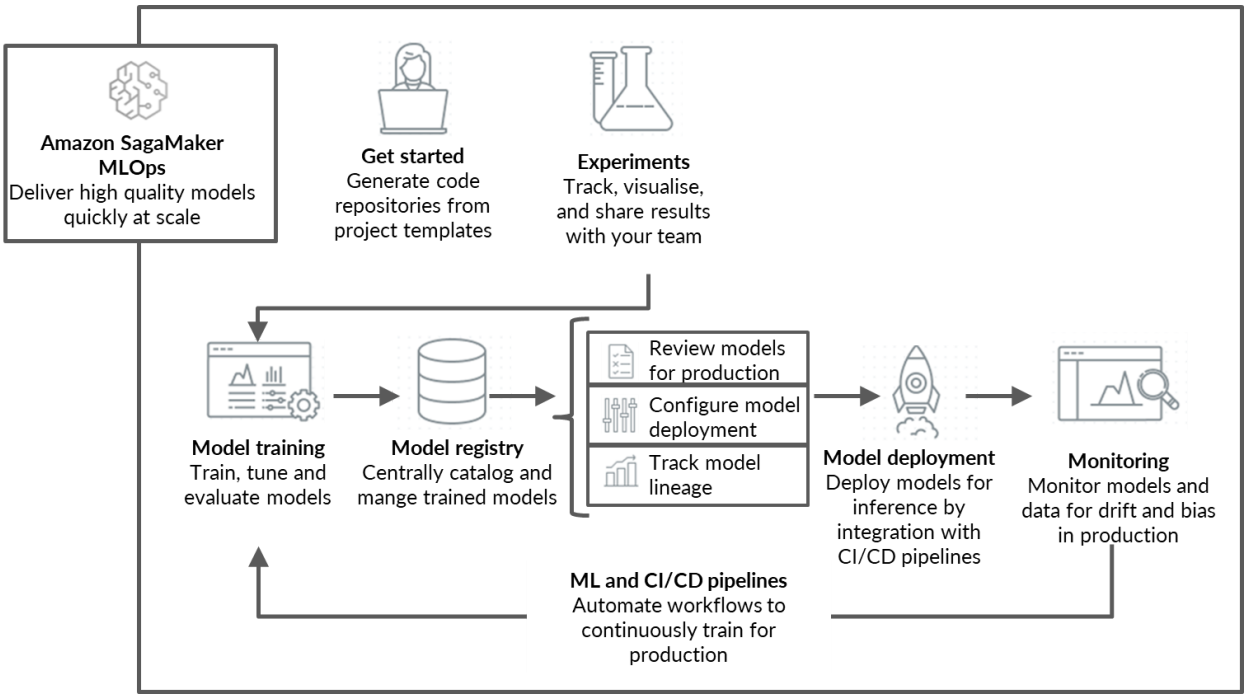
What is SageMaker?

- SageMaker is a cloud managed service provided by Amazon and is used by a myriad of tech giants for its MLOps practices
- Amazon SageMaker is a fully managed machine learning service.
- With SageMaker, data scientists and developers can quickly and easily build and train machine learning models, and then directly deploy them into a production-ready hosted environment.
- It provides an integrated Jupyter authoring notebook instance for easy access to your data sources for exploration and analysis, so you do not have to manage servers.
- It also provides common machine learning algorithms that are optimised to run efficiently against extremely large data in a distributed environment. With native support for bring-your-own-algorithms and frameworks, SageMaker offers flexible distributed training options that adjust to your specific workflows. Deploy a model into a secure and scalable environment by launching it with a few clicks from SageMaker Studio or the SageMaker console.
- Training and hosting are billed by minutes of usage, with no minimum fees and no upfront commitments.

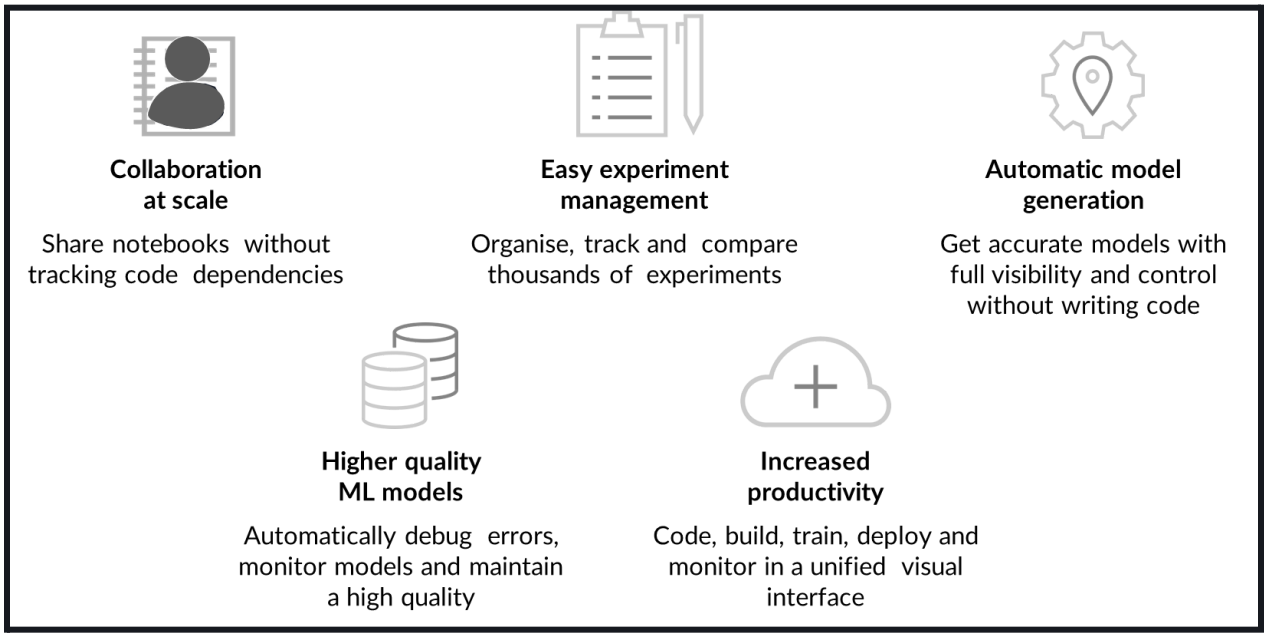
The following are the key features of AWS SageMaker:

- Amazon SageMaker Studio - First fully integrated development environment (IDE) for machine learning. This is similar to Jarvis
- Amazon SageMaker Notebooks - Enhanced notebook experience with quick-start and easy collaboration
- Amazon SageMaker Experiments - Experiment management system to organise, track and compare thousands of experiments. This is similar to MLflow
- Amazon SageMaker Debugger - Automatic debugging analysis and alerting
- Amazon SageMaker Monitor - Model monitoring to detect deviation in quality and take corrective actions. This is similar to Evidently
- Amazon SageMaker Autopilot - Automatic generation of machine learning models with full visibility and control. This is similar to PyCaret

Following is the step-by-step process of model deployment through SageMaker



The features of Amazon SageMaker Studio, which is an IDE for machine learning are as follows:



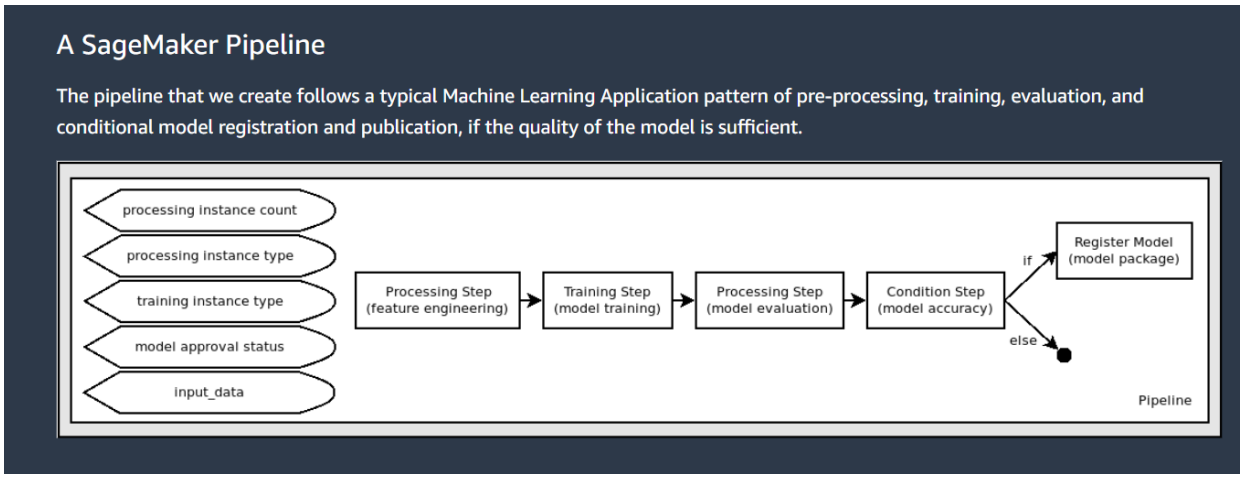
upGrad

To create an AWS account, you can follow the steps mentioned in this link

SagMaker tabs and functionalities

- Once you can see your user, you can click on the user and then on the Launch Studio button and you will be redirected to the Amazon Sagemaker Studio.
- 'File Browser' option - you can see the folders of your created projects.
- 'Running Terminals and Kernels' - you can use to view the running instances, apps, terminal and kernel sessions and also shut them down.
- 'Git' option - enables you to manage and link your git repositories with SageMaker studio.
- 'SageMaker JumpStart' option - gives you various prebuilt solutions to some of the generic industry problems. You can use these solutions to jumpstart your project and then customise it further as per your requirements.
- 'SageMaker resources' option - provides you with a dropdown menu that has projects, pipeline, endpoints, experiments and trials, etc.
- Projects -> Create Project - you will be able to see different templates to choose from. For our project, we have use MLOps template for model building, training and deployment.
- Once your project is created, you can clone the model build and model deploy repositories to obtain the folder structure.
- SageMaker resources -> 'Experiments and trials' option - You can see that for every pipeline execution, a trial with a unique identifier is created.
- SageMaker resources ->'Model registry' option - allows you to approve one model out of many trained models that will go into production. You can select the 'Model registry' option from the dropdown and select your project. Then, you would be able to see one or more model versions on the main page. Here, you can choose the version and manually change the status of that version from pending to approved or rejected as per your requirement.
- SageMaker resources ->Endpoints option - you can see the various endpoints created in the endpoints option from the dropdown

SageMaker Pipeline



Sagemaker pipeline is a directed acyclic graph of steps and conditions to orchestrate SageMaker jobs and resource creation.

SageMaker modelbuild project template

MLOps Practices and Benefits

Listed below are the MLOps practices and their benefits:

- Code, artifact and experiment tracking
  - **Challenge**: Bridging gap between model building and model deployment tasks
  - **Practice**: Lineage tracking and configuration management
  - **Benefit**: Repeatable process
  - **Solution**: Amazon SageMaker Experiments and Trials
- Continuous integration and deployment
  - **Challenge**: Providing end-to-end traceability
  - **Practice**: Auditable ML pipeline
  - **Benefit**: Improve time to market
  - **Solution**: Amazon SageMaker projects and pipelines
- Continuous training and model monitoring
  - **Challenge**: Continuous delivery and monitoring
  - **Practice**: Maintain model performance over time
  - **Benefit**: Improve time to market
  - **Solution**: Amazon SageMaker model monitor

upGrad

Open Source Versus Managed Services

The difference between setting up an MLOps system using cloud-based tools and using open source tools are summarised in the table below.

Services	Native cloud-based approach	Open source tools integration
End2End MLOps	Integrated	Plug and play
Time to set up	Less	High
Maintenance of infrastructure	Low	High
Ease of deployment	Low	High
Learning curve	Low	High
IDE Studio support	In-built	Need to be configured
Endpoint deployment	Integrated via SDK	Need to be configured
Pre configured MLOps template	Available	Not available
Companies leveraging	Cloud first companies, which have majority of infrastructure on Cloud	Companies that have on-premises infrastructure

The factors that you should consider when taking the decision regarding build versus buy are as follows:

- **The current stage of your company:** In the beginning, you might want to leverage vendor solutions to get started as quickly as possible so that you can focus your limited resources on the core offerings of your product. As your use cases grow, however, vendor costs might become exorbitant and it might be cheaper for you to invest in your own solution.
- **Competitive advantages of your company:** If machine learning (ML) infrastructure is something that the company wants to be really good at, then generally, it decides to build it in-house. For companies, where ML infrastructure is not their main focus, they tend to buy them.
- **Maturity of the available tools:** Companies that are early adopters build out their own infrastructure because there are no solutions mature enough for their needs. A few years later, solution offerings mature and new companies can opt for these solutions instead of building everything from scratch.

upGrad

Productionising the NLP Model - I

S3

S3 is an object storage service that lets users store and retrieve any amount of data at any time from anywhere. Here is what S3 can be used for:

- Back up and store data
- Archive data
- Host static websites
- As intermediate storage for other AWS services

CodeCommit

CodeCommit is a version control service hosted by AWS. You can use CodeCommit to store documents and source code, etc. in the cloud. It also supports the standard functionality of Git.

Data Preparation - Steps

- You started with the data preparation step.
- First, you uploaded the Jupyter notebook and the JSON file in Sagemaker Studio.
- Since the file size is approximately 80 MB, it takes some time to get uploaded, depending on your network bandwidth.
- To open the Jupyter notebook, you had to select an image and the corresponding kernel. Once that was done, you also changed the underlying machine to a more powerful one with 4vCPUs and 16 GiB of RAM to help with executing the different steps faster.
- Next, you loaded the data from JSON to a CSV file and started cleaning it.
- Then you ran a cell to create lemmatized text. Please note that this step takes some time to execute.
- Then, you cleaned the text further and had it ready to create the TFIDF and the document term matrix.
- Next, you found out the best number of topics, which was 5. Then you performed the test–train split and created CSV files for the train and the test data.
- Once the CSV files are ready, you will need to push them to an S3 bucket.
- You uploaded these files in the default S3 bucket created for your account. With these steps, you have prepared the data needed for training and testing the model.

- The next step was converting the raw text into a matrix of TF-IDF features and creation of document term matrix. Then using NMF, you found out the best number of topics for this dataset. You then assigned topic to each of the customer complaint.

Productionising the NLP Model - II

- You first updated the code repository for our use case. Then, you created a folder and uploaded all the .py files required for this use case.
- Then, in the preprocess.py file, we loaded the test and train data from the S3 bucket into a dataframe.
- You first need to import the different libraries that will be used for building the entire pipeline.
- Then, you defined a function to get the SageMaker session. This function takes two arguments - the AWS region to start the session and the default bucket to store the artifacts - and returns an instance of the SageMaker session.
- You then defined a function to fetch the Docker image from the AWS Elastic Container Registry (ECR). This function fetches the ECR URI of the Docker image based on the image name.
- Later, you defined some variables such as model_package_group_name and pipeline_name. Then, you defined some of the parameters for executing the pipeline. You defined the parameters such as the number of processing instances; default values for model approval status; input data; as well as the processing, training and inference image.
- For the input data, you updated the S3 URL as well and then defined the processing and training steps.
- After the training step, you defined the evaluate step in the pipeline.py file. Then, you updated the code in the evaluate.py file for our use case.
- Next, you calculated the accuracy score for the model, created a report dictionary and stored it in the evaluation_path. Later, you took a look at the code for the register model step.
- Next, you defined a condition step, wherein you fetched the report dictionary from the evaluation_path, and if the condition was true, you defined that the register step would be called and the model will be registered in the model registry. Finally, you created the pipeline instance and passed in the different parameters such as the following:
    - Name of the pipeline
    - Parameters such as the type and count of the processing instance, type of training instance, approval status of the model and the input data
    - Steps that need to be executed
    - An instance of the pipeline session
- Finally, the get_pipeline() function returned this pipeline instance that you created.

- Summarising the similarity and the difference between the initial pipeline.py file that was there as a result of cloning the repositories and the current version of the file for our use case - At both places, a bunch of libraries needed to be imported which will be used later in the code. We imported one additional library, logging. Using this library we will be logging the details so that we can refer to these logs, in the future and they will also come in handy to help debug or fix issues if we face some error in future.Once, we had imported all the required libraries, we wrote some functions. In the initial file, there a function was defined named get_pipeline_custom_tags() which is more of a utility function. This function returns the custom tags that we define for the pipeline. Since we would not require it in our project, we removed this function from our code. Also, this function was calling another function get_sagemaker_client() which was returning the sagemaker client. Since there was no use of this function as well we removed this function in our code as well. Then there were two functions get_session() and get_pipeline_session(). The first function get_session() returns an instance of a sagemaker session based on the region passed in the function parameter. The function get_pipeline_session() returns an instance of a pipeline session based on the region passed in the function parameter. These functions will be used in the get_pipeline() function to get an instance of sagemaker and pipeline session. Note that we have not made any changes to this function.
- We then defined two helper functions to resolve_ecr_uri() and resolve_ecr_uri_from_image_versions(). These functions are used to get the Elastic Container Registry (ECR) Universal Resource Identifier (URI) or in simpler terms, it fetches the docker images from the Elastic Container Registry.
- Now coming to the most important function which is the get_pipeline() function. This function returns an instance of a pipeline. It is this function only, where we define all our steps which need to be a part of the pipeline. In the function parameter, we passed in some parameters. Most of these parameters were already present in the initial code as well, but we passed two additional parameters ‘“project id” and “inference_instance_type”. Now let us look at the changes we made to this function to suit our requirements. Let’s try to understand it by considering one step at a time.
- Before going through the steps, we first created an instance of the sagemaker session and pipeline session and no changes need to be made in the initial code here. Then we defined a few parameters for pipeline execution. The parameters processing_instance_count and model_approval_status were used as it is. The next parameter, which is the input_data, you will need to provide the correct link for the S3 file. Also, we defined few additional parameters such as processing_image_name, training_image_name and inference_image_name which will be later used in the different steps. Let’s now look at each of the steps.
- The first step is the processing step for feature engineering. Here we first the URI of the processing image based on the processing image name that we had defined earlier in the parameters. In the original code, we had defined a SKLearn Processor; for our use case we define a ScriptProcessor variable and pass in multiple parameters. Then we define the call the run() method on the ScriptProcessor and store it in variable step_args which is then passed as a parameter to the ProcessingStep definition. The

ProcessingStep() function takes another parameter which is name of the step and for our use case we defined this name to be StepPreprocessComplaintsData. You can give it any name that you want. Let's now move on to the training step.

- In the training step, we first defined the model path. We then fetched the URI of the training image based on the training image name that we had declared earlier. We then defined the Estimator. The changes that we made while declaring the Estimator and the model path was around changing the folder name or base job name. Instead of the name Abalone, we changed the name to complaints. The major difference comes while defining the hyperparameters. This is completely dependent on the use case or the problem statement that you are working on. We defined the hyperparameters and then called the fit() method on the Estimator that we had defined and stored it in the step_args variable. We did not make any changes to the parameters passed in the fit method and finally, we called the TrainingStep() function and passed the step_args variable and the name of this step as parameters. Let's now move to the next step, which is the evaluation of the model.
- In this step, we are not making major changes to the original code. Wherever there was mention of Abalone, we replaced it with Complaints. For example, when we were creating an instance of ScriptProcessor, in the parameter base_job_name we updated it to script-complaint-eval from script-abalone-eval. So, these changes were mostly related to the variable names, step names or path names. There was no major change done with respect to the code or the logic.
- Now, we are left with two steps, register and condition. Next, let's look at the register step and find out what changes we made here. We first created an instance of ModelMetrics. This will be used when calling the register method. We need not make any changes in this part of the code. Next, we fetched the URI of the inference image based on the inference image name that we had defined earlier. This image URI needs to be passed as a parameter when we are creating an instance of the Model class and this is the only change that needs to be done here. And one more small change that needs to be done is around changing the name of this step according to our use case. We changed the name to StepRegisterComplaintsModel from RegisterAbaloneModel.
- Next, let's look at the condition step. Here, we made a few small changes to the existing code. We updated the parameter json_path inside the JsonGet() method. Also, we updated the right parameter in the ConditionLessThanOrEqualTo method. Apart from that, we just changed the name of the step according to our use case.
- At last, we created an instance of the pipeline and returned it. We did not make any changes in the code here.
- Then, you took a look at the Jupyter notebook to run the pipeline. You imported some of the libraries needed to run the notebook and defined some of the variables that will be passed to get an instance of the pipeline.
- Next, you submitted the pipeline and created an execution of the pipeline. Note that completing the entire execution will take some time.
- Later, you committed the changes and pushed the final code to the CodeCommit repository. Once you pushed your changes, you saw that in the CodeCommit repository, the different files that you had created were getting reflected. You observed that some of

the steps in the pipeline had completed their execution, and for those steps that were successfully executed, their colour changed to green. The step that is currently running is displayed in blue, and if any step fails to execute, its colour will change to red. To view additional information about each step, you can click on that particular step and you will be able to see different options such as input, output, logs and information.

- Once the entire pipeline is executed successfully, the trained model is saved in the model registry with the status 'Pending Manual Approval'. You had defined this in your code for the model status. An additional step that you needed to take was to go to CodeBuild and update the SOURCE_MODEL_PACKAGE environment variable.Then, you came back to the model registry and updated the status of the model to 'Approved'. Once this is done, the model will be deployed to an endpoint. Note that it might take approximately 15–20 minutes or more for the endpoint to show up.

Different capabilities of SageMaker pipelines

- **Build ML workflows**: Using Python SDK, you can build ML workflows comprising parameters, different steps and data dependencies. You can also orchestrate SageMaker jobs such as the processing job and the training job and can also trigger the execution of these pipelines.
- **Troubleshoot ML workflows**: You can visualise the execution of the pipeline and the status of each step in the pipeline in real time in SageMaker Studio. You can also view additional information about each of the steps in SageMaker Studio.
- **Manage models:** You can manage different versions of models using the Model Registry. You also have the capability to approve/reject models in the model registry. The model registry consists of different model packages, and each model package consists of multiple versions of the model.
- **Scaling MLOps**: You can create a project in SageMaker Studio and get a code repository, seed code and the MLOps infrastructure set up for you. You are provided with MLOps templates published by SageMaker for building, deploying and establishing end-to-end workflows.
- **Track lineage**: With in-built lineage tracking for SageMaker pipelines, you can track data, models and artefacts. Also, support is provided for tracking custom entities.

The Amazon SageMaker Model Monitor continuously monitors the quality of Amazon SageMaker ML models in production. The Model Monitor provides the following types of monitoring:

- Monitor data quality: Monitor drift in data quality
- Monitor model quality: Monitor drift in model quality metric, such as accuracy
- Monitor bias drift for models in production: Monitor bias in your model's predictions

You can set alerts using AWS CloudWatch when in the case of deviations in the model quality.