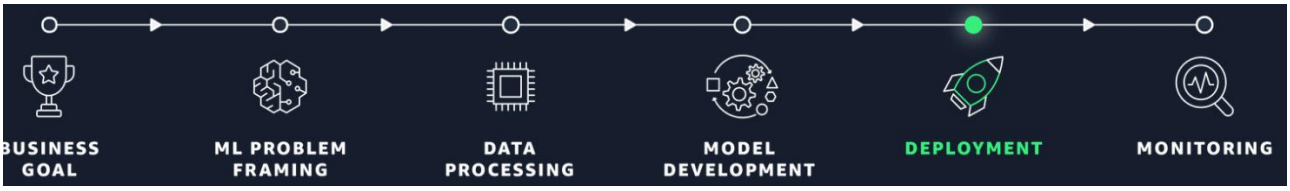
Domain 3: Select a deployment infrastructure



Model building and deployment infrastructure involves the following:

- Handling training and optimizing the machine learning models
- Data processing, feature engineering, model training, validation, and optimization
- Requires compute resources like GPUs for quickly training complex models
- Outputting the final trained model artifacts that will be used for inference

Inference infrastructure involves the following:

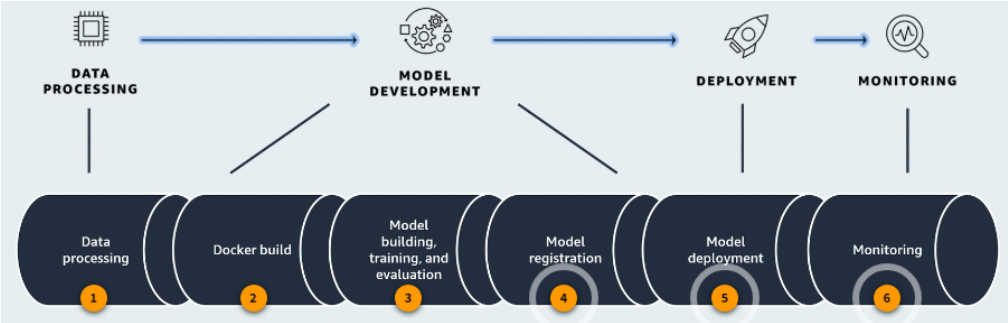
- Hosting the deployed trained models and handles running inference
- Receiving new unseen data, runs it through the models, and returns predictions and results
- Focusing on low latency, high throughput inference serving and scales to handle high query volumes without affecting latency
- Can be hosted on the cloud, on-premises, or at edge locations

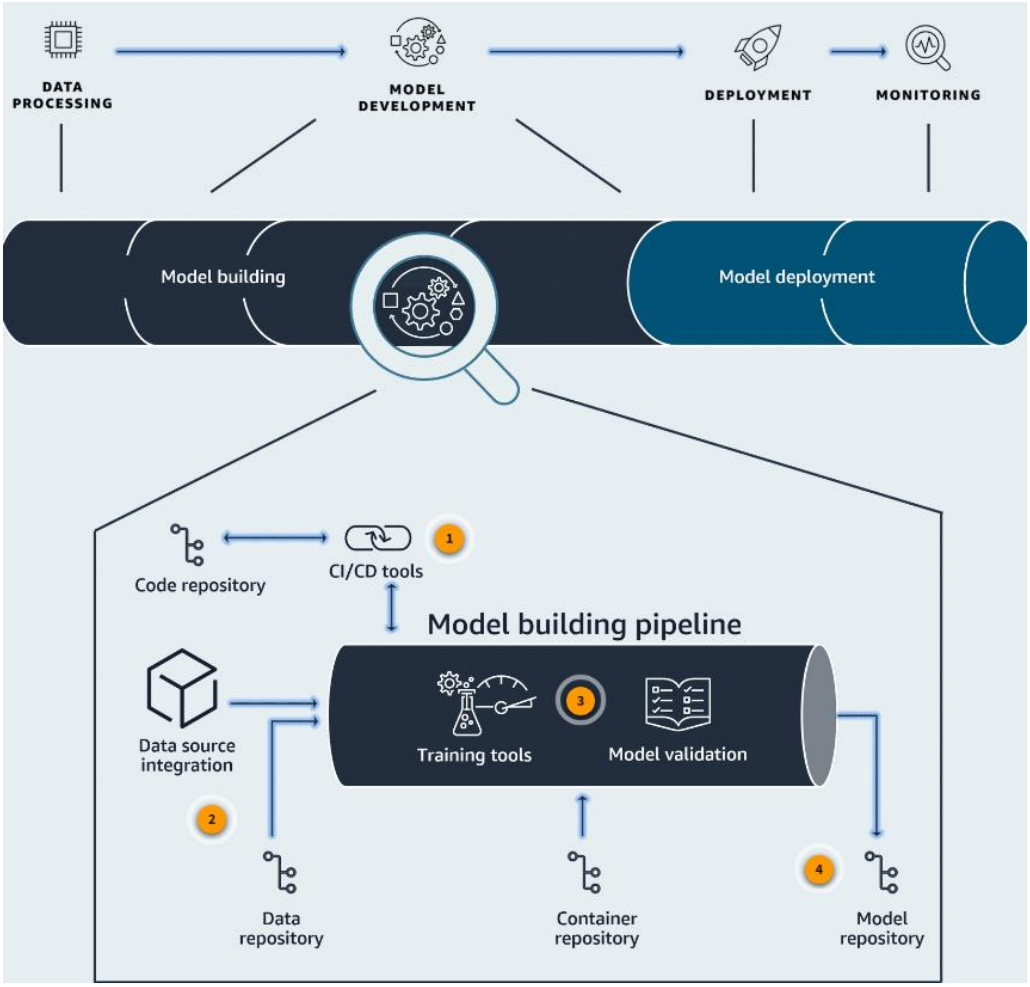
3.1 Select a Deployment Infrastructure

3.1.1 Model building & Deployment Infra

Building a Repeatable Framework

a) Example pipeline sequences -Options





Workflow Orchestration Options

a) Comparisons

Deployment Option	When to Use
SageMaker Pipelines	• When working entirely within the AWS SageMaker ecosystem • For end-to-end ML workflows that need to be automated and managed at scale
AWS Step Functions	• For serverless orchestration of ML pipelines • When you need to integrate ML workflows with other AWS services • For complex workflows with branching and parallel execution • When you want visual representation of workflow
Amazon MWAA (Managed Workflows for Apache Airflow)	• When you're familiar with Apache Airflow and prefer DAG-based workflows • For complex scheduling requirements • When you need to integrate with both AWS and non-AWS services
MLflow	• When you need an open-source platform for the complete ML lifecycle • For tracking experiments, packaging code into reproducible runs, and sharing and deploying models • When working in a multi-cloud or hybrid cloud environment • When you want to use a tool that integrates well with many ML frameworks and libraries
Kubernetes	• For container orchestration of ML workflows and deploying ML models at scale • When you need fine-grained control over resource allocation and scheduling • For multi-cloud or hybrid cloud deployments • When you want to leverage Kubernetes' extensive ecosystem (e.g., Kubeflow for ML-specific workflows

*b) **Comparisons:** AWS Controllers for Kubernetes (ACK) and SageMaker Components for Kubeflow Pipelines.*

AWS Controllers for Kubernetes (ACK)	SageMaker Components for Kubeflow Pipelines
• SageMaker Operators for Kubernetes facilitate the processes for developers and data scientists who use Kubernetes to train, tune, and deploy ML models in SageMaker. • You can install SageMaker Operators on your Kubernetes cluster in Amazon Elastic Kubernetes Service (Amazon EKS). • You can create SageMaker jobs by using the Kubernetes API and command-line Kubernetes tools, such as kubectl.	• You can move your data processing and training jobs from the Kubernetes cluster to the SageMaker ML-optimized managed service. • You have an alternative to launching your compute-intensive jobs from SageMaker. • You can create and monitor your SageMaker resources as part of a Kubeflow Pipelines workflow. • Each of the jobs in your pipelines run on SageMaker instead of the local Kubernetes cluster so you can take advantage of key SageMaker features.

3.1.2 Inference Infrastructure

Deployment Considerations & Deployment Infrastructure

a) Deployment Targets

***Best practice**: When*

	Benefits	Keep in mind	Choose when..	Use case
SageMaker endpoints	• Fully managed service • Convenient to deploy and scale • Built-in monitoring and logging • Supports various ML frameworks	• Not as customizable as other options • Potentially higher cost than other options	• You want a fully managed solution with minimal operational overhead and don't require advanced customization.	• A bank decides to use SageMaker endpoints to deploy ML models that detect fraud.
EKS	• Highly scalable and flexible • Supports advanced deployment scenarios • Supports custom configurations	• Possible higher operational overhead • Steeper learning curve to manage tool effectively	• You need advanced deployment scenarios and customized configurations, and you have the resources to manage the Kubernetes cluster	• A biomedical company uses EKS clusters to process DNA sequencing data. •
ECS	• Managed container orchestration service • Convenient to scale • Integrates well with other AWS services • Can run in Batch mode	• Limited advanced features compared to Kubernetes • Vendor lock-in	• You want a managed container orchestration service with good AWS integration and you don't require advanced Kubernetes features	• A renewable energy firm uses Amazon ECS to scale solar energy forecasting workloads.
Lambda	• Serverless • Automatically scales • Low operational overhead	• Limited run time • Cold starts can impact latency • Not suitable for long-running or complex models	• You have lightweight, low-latency models and want a serverless, pay-per-use solution	• A telehealth company uses Lambda functions for appointment reminders.

Choosing a model inference strategy

a) Amazon SageMaker inference options

SageMaker provides multiple inference options, including real-time, serverless, batch, and asynchronous to suit different workloads.

Inference Option	Description	When to Choose
Real-time	For low latency, high throughput requests	• When you need immediate responses (e.g., real-time fraud detection) • For applications requiring consistent, low-latency predictions • When your model can handle requests within milliseconds • For high-traffic applications with steady request rates
Serverless	Handles intermittent traffic without managing infrastructure	• For unpredictable or sporadic workloads • When you want to avoid managing and scaling infrastructure • For cost optimization in scenarios with variable traffic • For dev/test environments or proof-of-concept deployments
Asynchronous	Queues requests and handles large payloads	• For time-insensitive inference requests • When dealing with large input payloads (e.g., high-resolution images) • For long-running inference jobs (up to 15 minutes) • When you need to decouple request submission from processing
Batch Transform	Processes large offline datasets	• For offline predictions on large datasets • When you need to process data in bulk (e.g., nightly batch jobs) • For scenarios where real-time predictions are not required • When you want to precompute predictions for faster serving

Container and Instance Types for Inference

a) Choosing the right container for Inference

Approach	Description	When to Choose
SageMaker managed container images	Pre-built containers with inference logic included	• When using standard ML frameworks (e.g., TensorFlow, PyTorch, Scikit-learn) • For quick deployment without custom code • When built-in inference logic meets your needs • To leverage SageMaker's optimizations and best practices
Your own inference code	Custom containers with your specific inference logic	• When you need custom preprocessing or postprocessing • For proprietary algorithms or frameworks not supported by SageMaker • When you require specific dependencies or libraries • For full control over the inference environment

b) Choosing the right compute resources (AWS instance)

Instance family	Workload type
t family	Short jobs or notebooks
m family	Standard CPU to memory ratio
r family	Memory-optimized
c family	Compute-optimized
p family	Accelerated computing, training, and inference
g family	Accelerated inference, smaller training jobs
Amazon Elastic Inference	Cost-effective inference accelerators

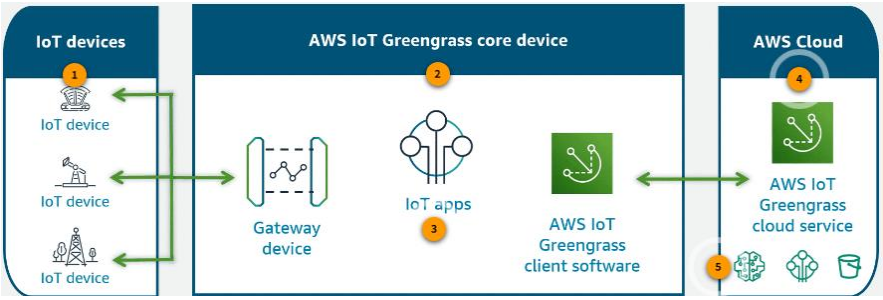
When to choose CPU, GPU of Inf2

Amazon EC2 P5 instances	GPU-based instances:	C2 Inf2 instances:
• High serial performance • Cost efficient for smaller models • Broad support for models and frameworks	• High throughput at desired latency • Cost efficient for high utilization • Good for deep learning, large models	• Accelerator designed for ML inference • High throughput at lower cost than GPUs • Ideal for models that AWS Neuron SDK supports

Optimizing Deployment with Edge Computing

a) *Using edge devices - AWS Options*

AWS IoT Greengrass



Amazon SageMaker Neo

b) *When to use which*

AWS IoT Greengrass	SageMaker Neo
• **Run at the edge**: Bring intelligence to edge devices, such as for anomaly detection in precision agriculture or powering autonomous devices. • **Manage applications:** Deploy new or legacy apps across fleets using any language, packaging technology, or run time. • **Control fleets:** Manage and operate device fleets in the field locally or remotely using MQTT or other protocols. • **Process locally**: Collect, aggregate, filter, and send data locally.	• **Optimize models for faster inference:** SageMaker Neo can optimize models trained in frameworks like TensorFlow, PyTorch, and MXNet to run faster with no loss in accuracy. • **Deploy models to SageMaker and edge devices:** SageMaker Neo can optimize and compile models to run on SageMaker hosted inference platforms, like SageMaker endpoints. As you've learned, it can also help you to run models on edge devices, such as phones, cameras, and IoT devices. • **Model portability:** SageMaker Neo can convert compiled models between frameworks, such as TensorFlow and PyTorch. Compiled models can also be run across different platforms and hardware, helping you to deploy models to diverse target environments. • **Compress model size:** SageMaker Neo quantizes and prunes models to significantly reduce their size, lowering storage costs and improving load times. This works well for compressing large, complex models for production.

3.2 Create and Script Infrastructure

These pillars provide a consistent and scalable designs.

The security pillar

- create ML solutions that anonymize sensitive data, such as personally identifiable information
- guides the configuration of least-privileged access to your data and resources
- suggests configurations for your AWS account structures and Amazon Virtual Private Clouds to provide isolation boundaries around your workloads.

The reliability pillar

- helps construct ML solutions that are resistant to disruption while recovering quickly
- guides you to design data processing workflows to be resilient to failures by implementing error handling, retries, and fallback mechanisms
- **recommends data backups, and versioning**.

The performance efficiency pillar

- focuses on the efficient use of resources to meet requirements.
- help you optimize ML training and tuning jobs by selecting the most suitable EC2 instance types for a particular task, running model inference using edge computing to minimize latency and maximize performance.

The cost optimization pillar

- focuses on building and operating systems that minimize costs
- In the data processing stage -> guides storage resource selection and tools for automation such as Amazon SageMaker Data Wrangler.
- During model development -> rightsizing compute resources
- Finally, during model deployment -> auto scaling

The sustainability pillar

- focuses on environmental impacts (energy consumption, efficient resource usage)

The operational excellence pillar

- focuses on the efficient operation, performance visibility, and continuous improvement

3.2.1 Methods for Provisioning Resources

IAC

a) Tools

Tool	Description	Language Support	Multi-Cloud Support	Typical Use Cases
CloudFormation	AWS-native IaC service	JSON, YAML	AWS only	• AWS-only • Teams familiar with AWS ecosystem • Simple to moderate complexity deployments
CDK (Cloud Development Kit)	IaC framework that compiles to CloudFormation	TypeScript, Python, Java, C#, Go	AWS only (can be extended)	• Teams with strong programming skills • Complex AWS infrastructures • Reusable ML infrastructure components
Terraform	Open-source IaC tool	HCL, JSON	Excellent	• Multi-cloud ML deployments • Hybrid cloud scenarios • Teams preferring declarative syntax
Pulumi	Modern IaC platform	TypeScript, Python, Go, .NET	Excellent	• Requires complex logic • Teams preferring familiar programming languages • Multi-cloud, complex architectures

Working with CloudFormation

a) Template

"**AWSTemplateFormatVersion**" : "2010-09-09"	**Format version** This first section identifies the AWS CloudFormation template version to which the template conforms.
"**Description**" : "Write details on the template."	**Description** This text string describes the template.

"**Metadata**" : {
"**Instances**" : {"Description" : "Info on instances"},
"**Databases**" : {"Description" : " Info about dbs"}
}
	Metadata
	These objects provide additional information about the template.

"**Parameters**" : {
"**InstanceTypeParameter**" : {
"Type" : "String",
"Default" : "t2.micro",
"**AllowedValues**" : ["t2.micro", "m1.small"],
"Description" : "Enter t2.micro or m1.small"
}
}
	Parameters
	Values **passed to your template** when you create or update a stack. You can refer to parameters from the Resources and Outputs sections of the template.

"**Rules**" : {
"Rule01": {
"**RuleCondition**": {
...
},
"**Assertions**": [
...
]}
}
	Rules
	Rules validate parameter values passed to a template during a stack creation or stack update.

"**Mappings**" : {
"Mapping01" : {
"Key01" : {
"**Name**" : "Value01"
}, ...
}}
	Mappings
	These are map keys and associated values that you can use to specify conditional parameter values. This is similar to a lookup table

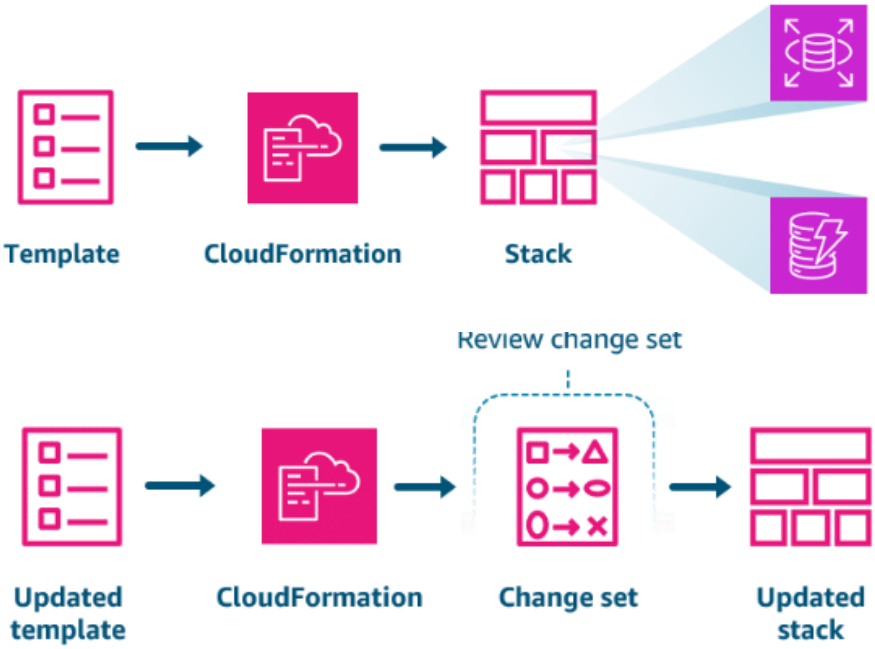
"**Conditions**" : {
"MyLogicalID" : {*Intrinsic function*}
}
	Conditions
	Control whether certain resources are created, or whether certain resource properties are assigned a value during stack creation or an update.

"**Transform**" : {
set of transforms
}
	Transform
	For **serverless** applications, transform specifies the version of the AWS SAM to use.

"**Resources**" : {
"Logical ID of resource" : {
"**Type**" : "Resource type",
"**Properties**" : {
Set of properties
}}
}
	Resources
	This section specifies the stack resources, and their properties that you would like to provision. You can refer to resources in the Resources and Outputs sections of the template.
	Note: This is the only required section of the template.

"**Outputs**" : {
"Logical ID of resource" : {
"**Description**" : "Information on the value",
"**Value**" : "Value to return",
"**Export**" : {
"**Name**" : "Name of resource to export"
}}}
	Outputs
	Describe the values that are returned whenever **you** view your stack's properties. For example, you can declare an output for an Amazon S3 bucket name and then call the aws cloudformation describe-stacks AWS CLI command to view the name**.**

b) CF Stacks



c) Provisioning stacks using CloudFormation templates

$ **aws cloudformation** create-stack \
  --**stack-name myteststack** \
  --**template-body** file:///home/testuser/**mytemplate.json** \
  --**parameters** ParameterKey=Parm1,ParameterValue=test1 ParameterKey=Parm2,ParameterValue=test2

Working with CDK

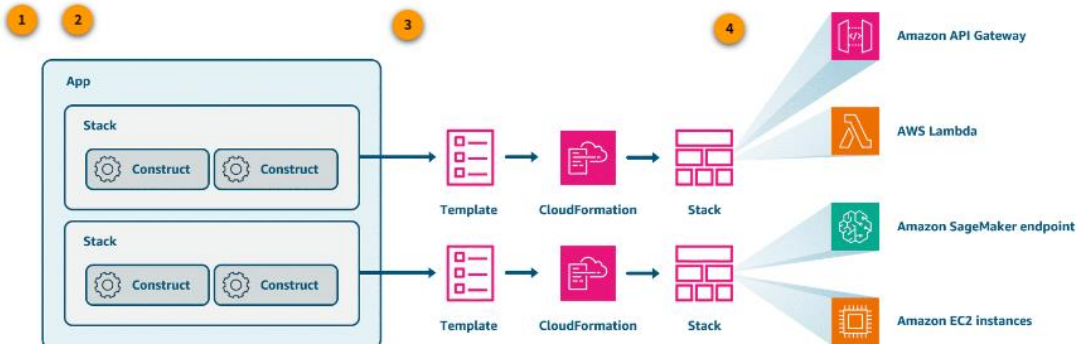
The AWS CDK consists of two primary parts:

- **AWS CDK Construct Library:** This library contains a collection of pre-written modular and reusable pieces of code called constructs. These constructs represent infrastructure resources and collections of infrastructure resources.
- **AWS CDK Toolkit:** This is a command line tool for interacting with CDK apps. Use the AWS CDK Toolkit to create, manage, and deploy your AWS CDK projects.

a) CDK Construct level comparisons

Level	*Description*	*Abstraction*	*Ease of Use*	*Typical Use Case*
L1	*Direct CF resources representation*	*Low*	*Low*	*full control over CF resources*
L2	*Logical grouping of L1 resources*	*Medium*	*Medium*	*most common*
L3	*High-level abstractions that represent complete solutions*	*High*	*High*	*Quickly deploy common architectural patterns*

b) CDK LifeCycle



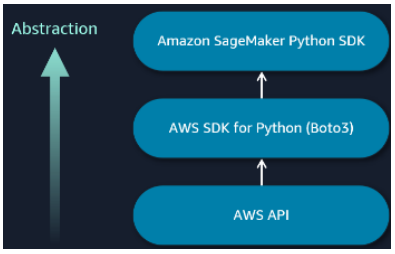
1	**cdk init** When you begin your CDK project, you create a directory for it, run **cdk init**, and specify the programming language used: • **mkdir** my-cdk-app • **cd** my-cdk-app • **cdk init** app --language typescript
2	**cdk bootstrap** You then run **cdk bootstrap** to prepare the environments into which the stacks will be deployed. This creates the special dedicated AWS CDK resources for the environments.
3	**cdk synth** Creates the CloudFormation templates using the **cdk synth** command.
4	**cdk deploy** Finally, you can run the **cdk deploy** command to have CloudFormation provision resources defined in the synthesized templates.

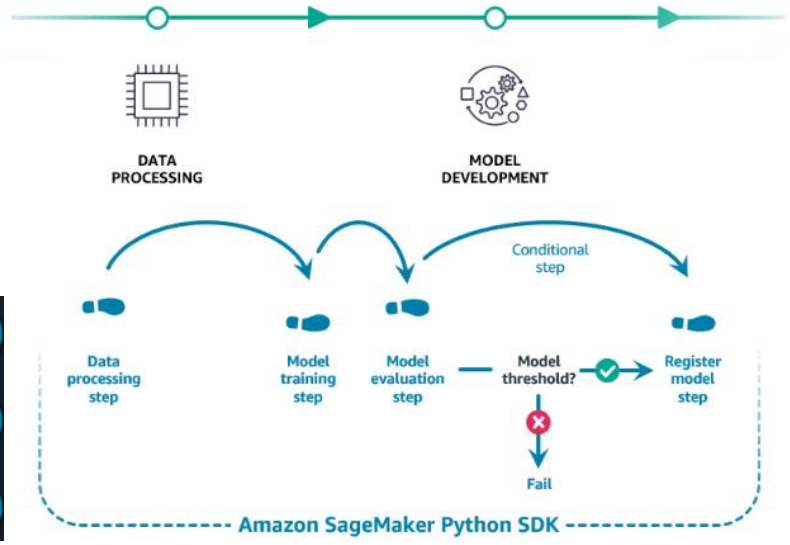
Comparing CF and CDK

The AWS CDK consists

	AWS CloudFormation	AWS CDK
Authoring experience	CF only uses JSON or YAML templates to define your infrastructure resources.	M modern programming languages, like **Python, TypeScript, Java, C#, and Go**.
IaC approach	CloudFormation templates are declarative. You define the desired state of your infrastructure and CloudFormation handles the provisioning and updates.	AWS CDK provides an imperative approach to generating CloudFormation templates, which are declarative, means you can introduce logic and conditions that determine the resources to provision in your infrastructure.
Debugging and troubleshooting	Troubleshooting CloudFormation templates requires learning specific CloudFormation error handling and messages.	With the CDK, you can use the debugging capabilities of your chosen programming language, making it more convenient to identify and fix issues in your infrastructure code.
Reusability and modularity	create nested stacks and cross-stack references, resulting in modular and reusable infrastructure designs. However, this approach can become complex and difficult to manage as your infrastructure grows.	Supports programming languages that you can use to apply object-oriented programming principles. This makes it more convenient to create modular and reusable IaC code blocks for your infrastructure.
Community support	CloudFormation has been around for a longer time and has a larger community for support. It also has a variety of third-party tools and resources.	Newer offering than AWS CloudFormation, but it is rapidly gaining adoption.
Learning curve	steeper learning curve for developers who are used to a more programmatic approach over a template-driven approach.	If you're already familiar with programming languages like Python or TypeScript, AWS CDK will have a gentler learning curve.

3.2.2 Deploying and Hosting Models

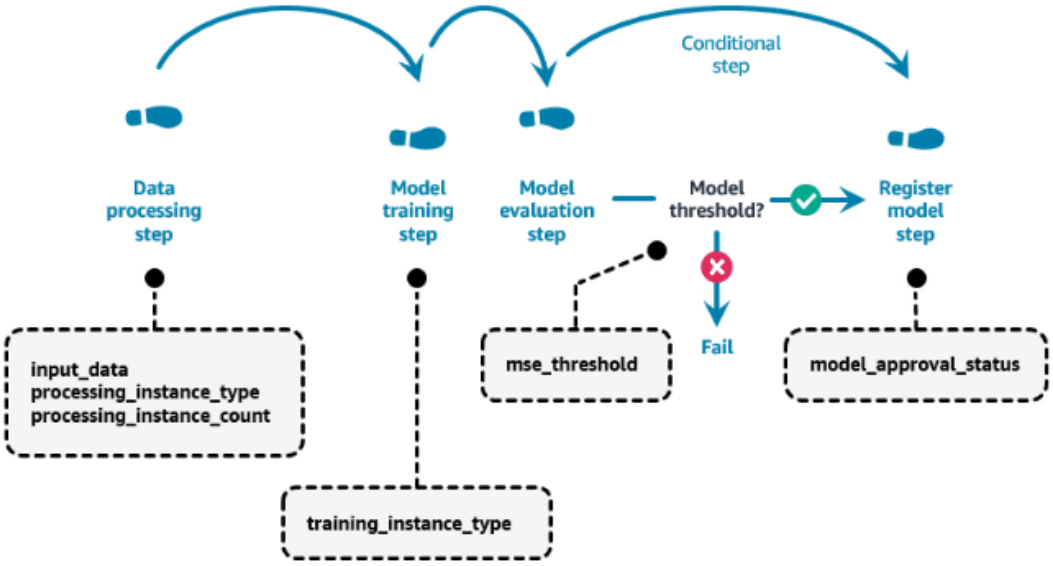




SageMaker Python SDK

a) Creating pipelines with the SageMaker Python SDK to orchestrate workflows

```
pipeline = Pipeline(
    name=pipeline_name,
    parameters=[input_data, processing_instance_type,
                processing_instance_count, training_instance_type,
                mse_threshold, model_approval_status],
    steps = [step_process, step_train, step_evaluate, step_conditional]
)
```



b) Automating common tasks with the SageMaker Python SDK

Preparing data (the.run())

With Amazon SageMaker Processing, you can run processing jobs for data processing steps in your machine learning pipeline. Processing jobs accept data from Amazon S3 as input and store data into Amazon S3 as output.

I. Creating the Processor

To define a processing job, you first create a Processor. The following example instantiates the **SKLearnProcessor()** class, which streamlines using **scikit-learn** in your data processing step:

```
from sagemaker.sklearn.processing import SKLearnProcessor

sklearn_processor = SKLearnProcessor(
framework_version="0.20.0",
role="[Your SageMaker-compatible IAM role]",
instance_type="ml.m5.xlarge",
instance_count=1,
)
```

II. Running the Processor

You then use the **.run()** method on the processor to run a processing job.

```
from sagemaker.processing import ProcessingInput, ProcessingOutput

sklearn_processor.run(
    code="preprocessing.py",
    inputs=[
        ProcessingInput(source="s3://your-bucket/path/to/your/data",
destination="/opt/ml/processing/input"),
    ],
    outputs=[
        ProcessingOutput(output_name="train_data",
source="/opt/ml/processing/train"),
        ProcessingOutput(output_name="test_data", source="/opt/ml/processing/test"),
    ],
    arguments=["--train-test-split-ratio", "0.2"],
)
```

III. Adding the data preprocessing step to a SageMaker Pipeline

Finally, you define a data preprocessing step in your pipeline using **ProcessingStep()**:

```
step_process = ProcessingStep(
    name="MyProcessingStep",
    ...,
    step_args = sklearn_processor.run(
    code="preprocessing.py",
    inputs=[
        ProcessingInput(source="s3://your-bucket/path/to/your/data",
destination="/opt/ml/processing/input"),
    ],
    outputs=[
        ProcessingOutput(output_name="train_data",
source="/opt/ml/processing/train"),
        ProcessingOutput(output_name="test_data", source="/opt/ml/processing/test"),
    ],
    arguments=["--train-test-split-ratio", "0.2"],    ),
)
```

Training Models (the.fit())

You can run model training jobs using the SageMaker Python SDK. The following model training job example manages the training script, framework, training instance, and training data input.

I. Creating the estimator for the training job

To define a model training job, you instantiate the **estimator** class. This class encapsulates training on SageMaker. The following code creates a model training job using the **MXNet()** class to train a model using the MXNet framework:

```
from sagemaker.mxnet import MXNet

mxnet_estimator = MXNet('train.py',
                        role='SageMakerRole',
                        instance_type='ml.p2.xlarge',
                        instance_count=1,
                        framework_version='1.2.1')
```

II. Running the training job

After you create the estimator, you can then use the **.fit()** method to run the training job. This method takes an argument that identifies the path to the training data. In this example, the training dataset is stored in Amazon S3:

```
mxnet_estimator.fit('s3://my_bucket/my_training_data/')
```

III. Creating a model training step in SageMaker Pipelines

Finally, you define a model training step in your pipeline using the **TrainingStep()** method:

```
step = TrainingStep(
    name="MyTrainingStep",
    step_args= mxnet_estimator.fit('s3://my_bucket/my_training_data/')
)
```

Deploying Models (deploy())

You can use SageMaker Python SDK to deploy a SageMaker model endpoint using the **deploy()** and **predict()** methods. You start by defining your endpoint configuration. The following code shows the configuration for a serverless endpoint:

```
from sagemaker.serverless import ServerlessInferenceConfig
serverless_config = ServerlessInferenceConfig(
    memory_size_in_mb=4096,
    max_concurrency=10,
)
```

You use this configuration in a **deploy()** method. If the model is already created, you use the **Model** class to create a SageMaker model from a model artifact. The model artifact location in Amazon S3 and the code to use to perform inference as the **entry_point** is specified:

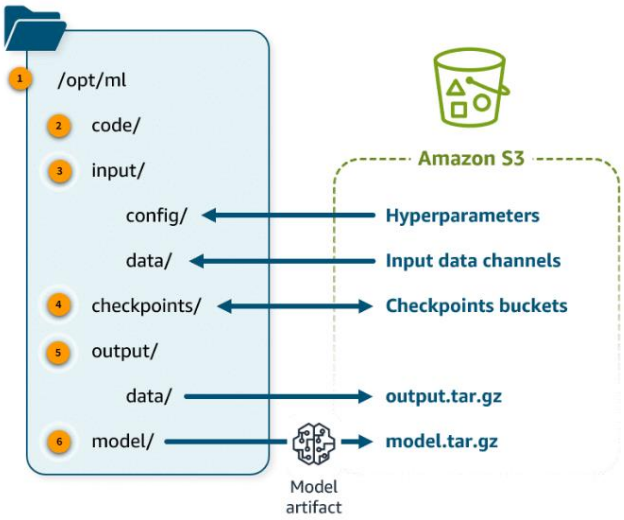
```
serverless_predictor = model.deploy(serverless_inference_config=serverless_config)
```

After deployment is complete, you can use the predictor's **predict()** method to invoke the serverless endpoint:

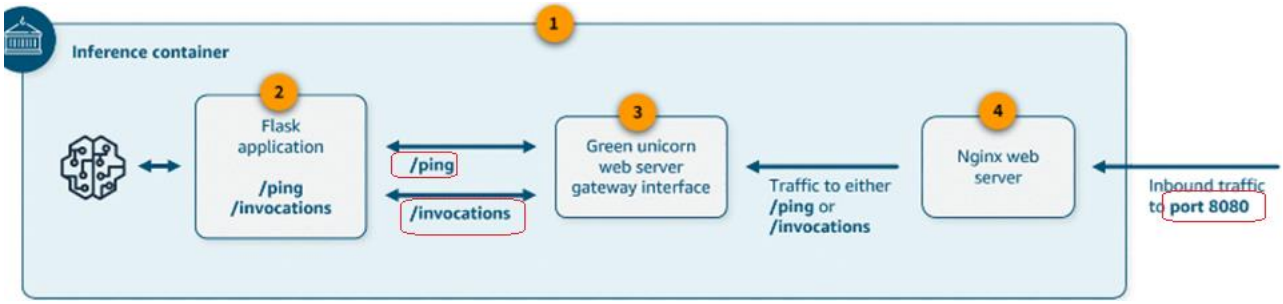
```
response = serverless_predictor.predict(data)
```

c) Building and Maintaining Containers

Training Container



Inference container



Below example is for Python script

Point	File
1	**serve.py** when the container is started for hosting. It starts the inference server, including the nginx web server and Gunicorn as a Python web server gateway interface.
2	**predictor.py** This Python script contains the logic to load and perform inference with your model. It uses Flask to provide the /ping and /invocations endpoints.
3	**wsgi.py** This is a wrapper for the Gunicorn server.
4	**nginx.conf** This is a script to configure a web server, including listening on port 8080. It forwards requests containing either /ping or /invocations paths to the Gunicorn server.

When creating or adapting a container for performing real-time inference, your container must meet the following requirements:

- Your container must include the path **/opt/ml/model**. When the inference container starts, it will import the model artifact and store it in this directory.

  Note: This is the same directory that a training container uses to store the newly trained model artifact.

- Your container must be configured to run as an executable. Your Dockerfile should include an ENTRYPOINT instruction that defines an executable to run when the container starts, as **ENTRYPOINT** ["<language>", "<executable>"]
  e.g. **ENTRYPOINT** ["python", "serve.py"]

- Your container must have a web server listening on **port 8080**.

- Your container must accept **POST** requests to the **/invocations** and **/ping** real-time endpoints.

- The requests that you send to these endpoints must be returned with 60 seconds and have a size <6 MB.

Auto scaling strategy

a) SageMaker model auto scaling methods

Scaling Method	Description	Use Case	Key Features
Target tracking scaling policy	Adjusts capacity to maintain a specified metric near a target value	When you want to maintain a specific metric (e.g., CPU utilization) at a target level	• Specify a metric and target value • Automatically adds or removes capacity • Good for maintaining consistent performance
Step scaling policy	Defines multiple policies for scaling based on specific metric thresholds	When you need more granular control over scaling actions at different metric levels	• Define multiple thresholds and corresponding scaling actions • More aggressive response to demand changes • Allows fine-tuning of scaling behavior
Scheduled scaling policy	Scales resources based on a predetermined schedule	When demand follows a predictable pattern (daily, weekly, monthly, yearly)	• Set one-time or recurring schedules • Use cron expressions with start and end times • Ideal for known traffic patterns
On-demand scaling	Manually increase or decrease the number of instances	For unpredictable or one-off events that require manual intervention	• Full manual control over scaling • Useful for new product launches, unexpected traffic spikes, or special promotions • Flexibility to respond to unforeseen circumstances

3.3 Automate Deployment

3.3.1 Introduction to DevOps

Code repositories

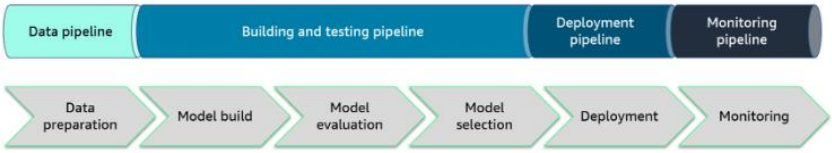
a) GitHub vs GitLab

Feature	GitHub	GitLab
Hosting Options	Cloud-hosted, GitHub Enterprise (self-hosted)	Cloud-hosted, Self-hosted (Community and Enterprise Editions)
CI/CD	GitHub Actions (built-in)	GitLab CI/CD (built-in)
Project Management	Projects, Kanban boards	Issue boards, Epics, Roadmaps
Third-party Integrations	Extensive marketplace	Fewer, but strong built-in tools
Open Source	Many open-source projects	Fully open-source core

3.3.2 CI/CD: Applying DevOps to MLOps

Introduction to MLOps

a) CICD in ML Lifecycle



b) Teams in the ML process

- **Data engineers**: Data engineers are responsible for data sourcing, data cleaning, and data processing. They transform data into a consumable format for ML and data scientist analysis.
- **Data scientists:** Responsible for model development including model training, evaluation, and monitoring to drive insights from data.
- **ML engineers: R**esponsible for MLOps - model deployment, production integration, and monitoring. They standardize ML systems development (Dev) and ML systems deployment (Ops) for continuous delivery of high-performing production ML models.

c) Nonfunctional requirements in ML

- **Consistency**
- **Flexibility**: Accommodates a wide range of ML frameworks and technologies to adapt to changing requirements.
- **Reproducibility**
- **Reusability**:
- **Scalability**:
- **Auditability**: Provides comprehensive logs, versioning, and dependency tracking of all ML artifacts for transparency and compliance.
- **Explainability**: Incorporates techniques that promote decision transparency and model interpretability.

d) Comparing the ML workflow with DevOps

Feature	DevOps	MLOps
Code versioning	✓	✓
Compute environment	✓	✓
CI/CD	✓	✓
Monitoring in production	✓	✓
Data provenance		✓
Datasets		✓
Models		✓
Model building with workflows		✓
Model deployment workflows		✓

Automating testing in CI/CD Pipelines
SageMaker projects with CI/CD practices

- **Unit tests**
  validate smaller components like individual functions or methods.
- **Integration tests**
  can check that pipeline stages, including data ingestion, training, and deployment, work together correctly. Other types of integration tests depend on your system or architecture.
- **Regression tests**
  In practice, regression testing is re-running the same tests to make sure something that used to work was not broken by a change.

Version Control Systems: Getting started with Git
SageMaker projects with CI/CD practices

Command	Description
git init	Initializes a new Git repository in the current directory
git clone [repository_url]	Creates a local copy of a remote repository
git add [file(s)]	Stages the specified file(s) for the next commit
git commit -m "commit message"	Records the staged changes with a descriptive commit message
git push	Uploads local committed changes to a remote repository
git pull	Fetches and merges changes from a remote repository into the local branch
git branch	Lists all available branches in the repository
git checkout [branch_name]	Switches to the specified branch
git merge [branch_name]	Merges the specified branch into the current branch

Continuous Flow Structures : Automate deployment

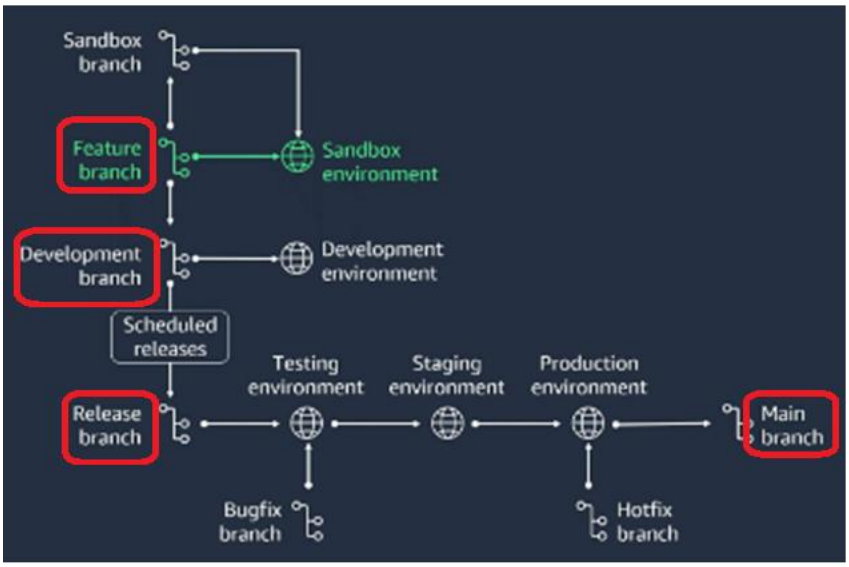
a) *Key components*

1) ***Model training and versioning:***
2) ***Model packaging and containerization:***
3) ***Continuous integration (CI):***
4) ***Monitoring and observability:***
5) ***Rollback and rollforward strategies:***

b) *Gitflow and GitHub flows*

Feature	Gitflow	GitHub Flow
Complexity	More complex	Simpler
Main Branches	main and develop	Single main branch
Feature Development	Feature branches from develop	Feature branches from main
Release Process	Dedicated release branches	Direct to main via pull requests
Hotfixes	Separate hotfix branches	Treated like features
Suited For	Scheduled releases, larger projects	Continuous delivery, smaller projects
Integration Branch	develop branch	N/A (uses main)
Learning Curve	Steeper	Flatter
Flexibility	More rigid structure	More flexible

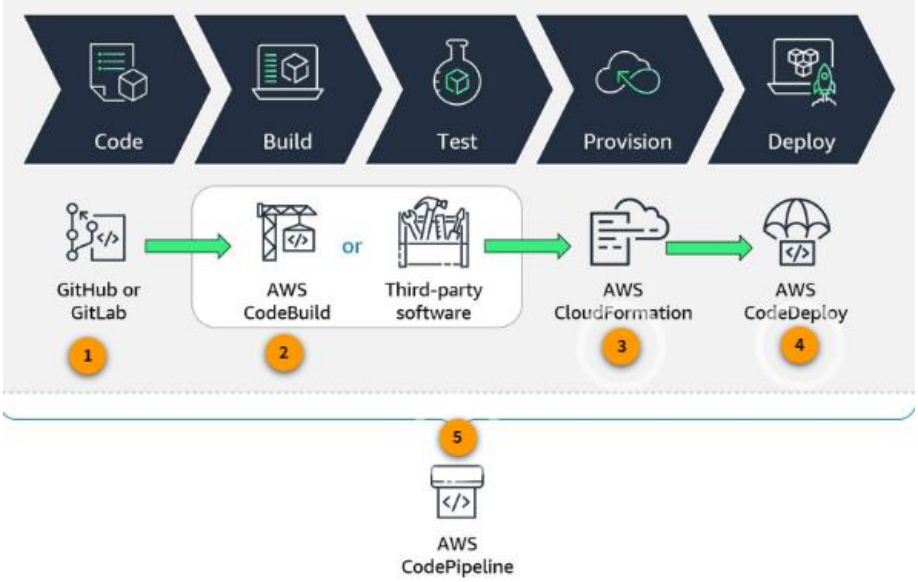
c) *GitFlow*



3.3.3 AWS Software Release Processes

Continuous Delivery Services

a) AWS CI/CD Pipeline



CodePipeline *Provides configurable manual approval gates to control releases, detailed monitoring capabilities, and granular permissions to manage pipeline access, as we will explore further.*	• ***Each AWS account: 1000 pipelines*** • ***The number of actions in a single pipeline: 500.*** • ***The size of input artifact for a single action: 1 GB*** • ***#of custom actions for/region/account: 50.*** • ***# of webhooks/region/account : 300.***
CodeBuild *CodeBuild sets service quotas and limits on builds and compute fleets. These quotas are for each supported AWS Region for each AWS account.*	• ***detailed logging, auto-scaling capacity, and high availability for builds*** • ***integrates with other AWS services like CodePipeline and ECR for end-to-end CI/CD workflows*** • ***artifacts can be stored in S3 or other destinations*** • ***build can be monitored through the CodeBuild console, Amazon CloudWatch, and other methods*** • ***fine-grained access controls for build projects using IA) policies.***
CodeDeploy *CodeDeploy is a deployment service that provides automated deployments, flexible deployment strategies, rollback capabilities, and integration with other AWS services to help manage the application lifecycle across environments.*	• ***facilitates automated deployments to multiple environments*** • ***supporting deployment strategies like blue/green, in-place, and canary deployments*** • ***provides rollback capabilities,*** • ***detailed monitoring and logging, and integration with services like EC2, Lambda, and ECS.***

Best Practices for Configuring & Troubleshooting

Service	Purpose	Key Configuration Steps	Troubleshooting Tips
CodeBuild	Compiles source code, runs unit tests, and produces deployment-ready artifacts	1. Create a CodeBuild project 2. Define build specification 3. Configure build environment 4. Set up build artifacts 5. Configure CloudWatch Logs (optional)	• Validate buildspec file • Verify IAM permissions • Review service limits • CloudTrail **Unique** • Check for network issues • Check CodeBuild logs
CodeDeploy	Automates application deployments to various compute platforms	1. Set up IAM role 2. Create CodeDeploy app 3. Create deployment group 4. Define deployment configuration	**Unique** • Review CodeDeploy logs • Verify CodeDeploy agent • Validate AppSpec file • Check instance health • Analyze the rollback reason
CodePipeline	Models, visualizes, and automates software release steps	1. Create CodePipeline pipeline 2. Add source stage 3. Add build stage 4. Add deploy stage 5. Review and create pipeline	• Validate build specifications ((buildspec.yml) • Verify input configuration • Use CloudTrail • Check the IAM permissions **Unique** • Use CloudWatch • Examine runtime details • Check pipeline history

Automating Data Integration in ML Pipeline

Code Pipeline vs Step Functions

Aspect	*AWS CodePipeline*	*AWS Step Functions*
Primary Purpose	*CI/CD* and release automation	Workflow orchestration and coordination
Workflow Type	Linear, predefined stages	Complex, branching workflows with conditional logic
Best For	Standard software deployment pipelines	Complex, multi-step processes and microservices orchestration

MLOps with Code Pipeline and Step Functions

- **MLOps Overview:**
  - Set of practices and tools for streamlining ML model deployment, monitoring, and management
  - Focuses on automating ML workflows in production environments
- **AWS Step Functions:**
  - Fully managed visual workflow service for building distributed applications
  - Represents pipeline stages (preprocessing, training, evaluation, deployment) as task states
  - Manages control flow between stages
  - Can integrate with Lambda, AWS Batch, and other AWS services
- **AWS CodePipeline:**
  - Fully managed continuous delivery service
  - Automates release pipelines for MLOps workflows
  - Represents each stage of the pipeline as an action
- **Integration of Step Functions and CodePipeline:**
  a) CodePipeline invokes MLOps pipeline based on events (e.g., new model version commit)
  b) Pipeline stages include source code management, model building, testing, and deployment
  c) CodePipeline starts Step Functions state machine to initiate MLOps workflow
  d) Can pass input data or parameters to configure the workflow
- **Benefits of Integration:**
  - Enables efficient movement of models through development lifecycle
  - Automates the entire process from training to production deployment
  - Provides flexibility in configuring and managing complex ML workflows

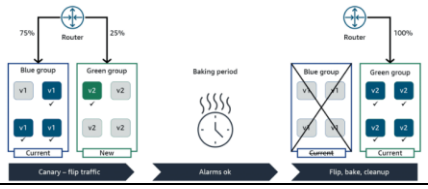
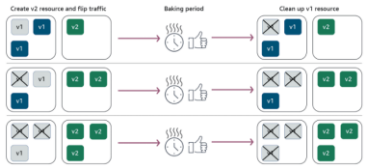
Deployment strategies

All at once
Shifts all endpoint traffic from the existing blue fleet to the new green fleet in a single step. The pre-specified CloudWatch alarms then monitor the green fleet for issues during the baking period. If no alarms trip, SageMaker terminates the old blue fleet after the baking period.

Canary
Traffic is shifted to a new fleet in two steps. First, a small portion of traffic (the canary) is shifted to the new fleet and monitored during a baking period. If the canary succeeds, the remainder of traffic is shifted from the old fleet to the new fleet, and the old fleet is terminated.

Linear
Traffic is shifted gradually in a pre-specified number of equal, incremental steps. The number of steps and percentage of traffic shifted at each step can be customized. This extends basic canary shifting from 2 steps to n linearly spaced steps for more gradual and customizable traffic shifting.

Comparison deployment

Blue/green deploy	***Canary deployment***	***Rolling deployment***
maintaining two identical prod environments, one blue (existing) and one green (new), and gradually shifting traffic between them.	Gradually rolling out a new model version to a small portion of users	Gradually replace previous deployment of model version with new version by updating endpoint in configurable batch sizes
		
• When you need instant rollback capability • For critical applications requiring zero downtime • When your application can handle sudden traffic shifts	• To test new features with a subset of users • When you want to gather user feedback before full release • For applications with high traffic where you want to minimize risk	• When you have stateful application • For large-scale deployments where cost is an issue • When you can tolerate having mixed versions temporarily

The baking period is a set time for monitoring the green fleet's performance before completing the full transition, making it possible to rollback if alarms trip. This period builds confidence in the new deployment before permanent cutover.

3.3.4 Retraining models

Retraining models

a) Retraining mechanisms

Automated retraining pipelines	Invoke model retraining when new data becomes available	
Scheduled retraining	Periodic jobs to retrain the model at regular intervals	
Drift detection and invoked retraining	invoke retraining when the model's performance starts to degrade.	SageMaker Model Monitor + Lambda (SageMaker Model Monitor can detect model drift, and Lambda can be used to initiate the retraining process.)
Incremental learning	Incremental learning allows the model to be updated with new data without completely retraining the model from scratch.	• XGBoost • Linear Learner (AWS SageMaker supports several algorithms for this, as above)
Experimentation and A/B testing	Retraining can be paired with experimentation and A/B testing to compare various model versions	AWS SageMaker + Personalize (AWS SageMaker and Amazon Personalize can be used to deploy and manage these experiments.)

b) Catastrophic forgetting during retraining and transfer learning

Catastrophic forgetting is a phenomenon that occurs in machine learning models, particularly in the context of continual or lifelong learning.

***Catastrophic forgetting is a type of over-fitting*.** The model learns the training data too well such that it no longer performs well on other data.

Why does catastrophic forgetting happen?

a. **Retraining and optimized training:** The primary reason for catastrophic forgetting is that the parameters of the model are typically updated to optimize for the current task. These updates effectively overwrite the knowledge acquired from previous tasks.
b. **Transfer Learning:** Transfer learning is an ML approach where a pre-trained model, which was trained on one task, is fine-tuned for a related task. Organizations can make use of transfer learning to retrain existing models on new, related tasks using a smaller dataset.

Solving catastrophic forgetting

Method	Main Idea	Advantages	Limitations
Rehearsal-based	Retrain on a subset of old data along with new data	• Directly addresses forgetting • Conceptually simple	• Requires storing old data • Can be computationally expensive
Architectural	Modify network architecture to accommodate new tasks	• Can be very effective • Doesn't require old data	• May increase model complexity • Can be challenging to design
Replay-based	Generate synthetic data to represent old tasks	• Doesn't need storing old data • Work well with Gen AI models	• Quality depends on model • May not capture all aspects of old data
Regularization-based	Add constraints to limit changes to important parameters	• Doesn't require old data or architecture changes • Often computationally efficient	• May limit learning of new tasks • Determining importance can be challenging

Configuring Inferencing Jobs

a) Inference types

Real-time inference
Real-time inference is ideal for inference workloads where you have real-time, interactive, low latency requirements. These endpoints are fully managed and support autoscaling. For more information, see Real-time inferencing.

Serverless inference
Serverless inference can be used to serve model inference requests in real time without directly provisioning compute instances or configuring scaling policies to handle traffic variations. For more information, see Serverless Inference.

Asynchronous inference
Asynchronous inference queues incoming requests for asynchronous processing. This option is ideal for requests with large payload sizes, long processing times, and near-real-time latency requirements. The latency requirement is one reason to choose asynchronous over batch. For more information, see

Batch transform
Batch transform provides offline inference for large datasets. Batch transform is helpful for preprocessing datasets or gaining inference from large datasets. For more information, see Use Batch Transform.

Differences between training and inferencing

Training	*Inferencing*
Requires high parallelism with large batch processing for higher throughput	*Usually run on a single input in real time*
More compute- and memory- intensive	*Less compute- and memory-intensive*
Standalone item not integrated into application stack	*Integrated into application stack workflows*
Runs in the cloud	*Runs on different devices at the edge and cloud*
Typically runs less frequently and on an as-needed basis	*Runs **an indefinite amount of time***
Compute capacity requirements are typically predictable, so wouldn't require auto scaling	*Compute capacity requirements might be dynamic and unpredictable, **so would require auto scaling***