|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data Preparation for ML: 1) Ingest & store data. 2) Transform data & feature engineering  3) Ensure data integrity & prepare data for modelling | | | | | | |
| Data Formats | | | Structured: - Row-based (Protobuf or Avro recordIO). – Column-based (Parquet, Apache ORC)  Semi-structured: - Row-based (Excel, csv). – Object-notation (JSON, JSONL, XML)  Unstructured: - Images (.png, .jpg). – Video (.mp4, .ogg, .webm). – Text (.txt, raw text)  1) Row-based data format:  - CSV, Avro RecordIO (stores records sequentially. Benefits ML workloads that need to iterate over full dataset multiple times during model training. Also has schema which can improve data processing speeds & better data management compared to schema-less formats)  2) Column-based: Queries extract insights from patterns within a col rather than entire record  - Parquet (more for read-heavy), ORC (optimized row columnar. more for write-heavy. Used in big data workloads like Apache Hive and Spark)  3) Object-notation data: for non-tabular, hierarchical data like graphs or textual data. Data is structured into hierarchical objs w features and key-value pairs  - JSON (JavaScript object notation: document based data format. JSON is represented in objs and arrays. An **obj** = data defined by key-value pairs and enclosed in braces)  - JSONL (JavaScript Object Notation Lines: newline-delimited JSON. Each json obj written on its own line. Improves efficiency as individual objs can be processed w/o loading a larger JSON array. JSONL structure can map to columnar formats like Parquet) | | | |
| Data Quality (5Rs) | | | 1) Relevant: irrelevant data should not be included in training  2) Representative: training dataset should be representative of “test” data  3) Rich: Want data from multiple sources to be rich w relevant features  4) Reliable: incoming data should have consistent features and formatting  5) Responsible : ethically sourced, limited access, encrypted, sensitive data masked/removed | | | |
| Data Storage: Data repos | | | 1) Data lake: AWS S3 & Lake Formation. 2) Data warehouse: Redshift  3) Relational DB: Amazon Relational Database Service (RDS). 4) Non-relational DB: Dynamo DB  Lake formation: centralize governance and management of many related S3 buckets  Data warehouse can store structured data optimized for business analytics  RDS used for OLTP, web apps, mobile apps, PostgreSQL can be used for vector search  DynamoDB: serverless NoSQL DB. Stores semi-structured data in JSON-like items.  - Single-digit millisecond latency at any scale.  – Facilitates event-driven architectures w DynamoDB Streams | | | |
| Data Storage: for ML | | | S3, Elastic Block Store (EBS), Elastic File System (EFS), FSx for Lustre  S3: used as a data lake, intermediate data storage, and for training and evaluation data  EBS: used w EC2. Training data can be pre-loaded or streamed to EBS volumes  EFS: shared file system that can mount directly to Linux training instances (EC2)  FSx for Lustre: can mount to EC2 and read data from S3.  - Handle hundreds of GBs of throughput, & millions of IOPS for super-low-latency file retrieval | | | |
| Choosing Data Storage | | | I/O throughput and latency, scalability and elasticity, and data access patterns determine storage service needs for a given ML workload.  1) Inference workloads: require fast response times for delivering predictions, but don’t require high I/O performance, except for real-time inference cases.  - EBS gp3 volumes or EFS are suitable. For increased low-latency demands: EBS io2 volume  2) Training workloads: high performance and frequent random I/O access to data  - EBS volumes (EBS instance store volumes offer extremely low-latency data access due to data being stored directly on instances rather than on network-attached volumes)  3) Real-time and streaming workloads (media processing, web serving, content management)  - EFS allow low latency and concurrent data access. Provides high throughput access  4) Dataset storage: S3 (don’t need quick access) | | | |
| Access patterns:  1) Copy and load: data copied from S3 to a training instance backed by EBS  2) Sequential streaming: typically from S3 to instances backed by EBS  3) Randomized access: data randomly access like w shared file system (FSx and EFS) | | | |
| S3 | | | Storage classes: S3 Standard, S3 Standard-IA. S3 Express One Zone, S3 One Zone – IA. (One zone faster than standard class as only in 1 AZ)  S3 Glacier Deep Archive -> S3 Glacier Flexible Retrieval -> S3 Glacier Instant Retrieval  Use S3 Intelligent-Tiering if access patterns of data is unknown | | | |
| Use S3 Lifecycle Management if access patterns are known:  - Define Lifecycle policies to move object to different class (obj transition OR obj expiration) | | | |
| Users -> IAM permissions -> Lake Formation permissions (restrict access to particular rows, columns, or individual cells) -> Data | | | |
| S3 events: obj placed in bucket -> New obj event -> EventBridge -> Lambda/Glue job  Automatically trigger event-driven workloads w S3 events  Send events to event handlers like SNS, SQS, or EventBridge, or directly to trigger Lambda | | | |
| S3 -> AWS Glue -> Glue Data Catalog. Athena then reads data from S3 + Catalog  Metadata stored in Glue data Catalog can be queried by Athena  Athena can query petabytes of data stores in S3. Great for data exploration and ad hoc analysis  At scale, Athena queries can experience increased runtime and cost  By using S3 Partitioning, query performance can be improved  - Partitions are Apache Hive-like storage patterns that you can define in Athena | | | |
| S3 Gateway Endpoints. Access S3 w/o traversing public internet (cheaper)  AWS Region { VPC [ Private Subnet ( EC2 -- ) --> Gateway endpoint -- ] --> S3 bucket }  - Gateway endpoints only work in same region. (If diff region, use interface endpoint)  - gateway endpoints for S3 & DynamoDB | | | |
| Data Ingestion | | | 1) Batch ingestion: data arrives in batches, often one-time or recurring schedule.  - Used for historical data analysis and ML model training  - CLI/API (*aws s3 sync local s3://bucket*), AWS DataSync, AWS DMS  2) Streaming ingestion: data continuously generated.  – Used for real-time analysis and ML inference streaming  - Kinesis Data Streams, Amazon Managed Service for Apache Flink, Kinesis Data Firehose | | | |
| KDS: fully managed service to ingest data streams  - Write up to 1MB or 1000 records per sec & read up to 2MB or 2000 records per sec for each shard  - Default limit of 10000 shards per stream, but technically no upper limit  Flink: interactively query real-time data and generate continuous insights  - Detect outliers and threshold breaches as early as possible  KDF: deliver data to integrated services like S3, Redshift, Amazon OpenSearch Service  - Can also deliver to apps like Splunk, Snowflake or a custom HTTP endpoint  - Can convert data to Parquet or ORC. – Can integrate w Lambda for custom transformations  - Can dynamically partition data as it is delivered to S3  Amazon Managed Streaming for Apache Kafka (MSK): create Kafka clusters from scratch or deploy existing Kafka cluster to AWS  - Optimized for capturing log and event streams  - Native integrations w Kinesis family, EC2, Lambda, Redshift and more  - More control but more operational overhead than Kinesis  - Define stream processing logic w Apache Zeppelin notebooks | | | |
| Data Transform-ations | | | Batch transformations: Glue or Glue DataBrew  Glue: family of integrated services for serverless data discovery, preparation and ETL  - Define ETL jobs as Python Shell jobs or Apache Spark jobs  - Define low-code ETL jobs using AWS Glue Studio. – Run event-driver or scheduled batch jobs  - Charged per DPU-hour. Standard DPU provides 4vCPU and 16GB of memory  Python shell jobs: used for very small transformations on datasets < 10GB  - Uses 1 DPU or a fraction of a DPU (≈ 1GB memory)  Spark jobs: allocate 2 to 100 DPUs, or for Glue 2.0 jobs specify type and num of workers  - For more memory intensive jobs on larger data sets | | | |
| DataBrew: Simple data transformation tool for cleaning and normalizing data  - For non-technical data analysts. – Visually explore data sets. – Map data lineage  – Assemble predefined no-code data transformations in visual interface. | | | |
| Streaming Transformations: EMR, Glue Streaming job, Kinesis Data Firehose (w Lambda)  Glue Streaming Jobs: Less operational overhead than EMR.  - Writes out data in micro-batches that are 100seconds windows by default  EMR: More real-time than Glue Streaming Jobs but More operational overhead, and incur cost when not in use. - More cost-effective for constant ETL. | | | |
| Transfer Data to AWS | | AWS CLI/SDK, DataSync, Database Migration Service (DMS), S3 Transfer Acceleration, Snowball  Programmatic data transfer: CLI or SDK. – Can transfer data over internet or over dedicated connection (Direct Connect). – Good for one-time or scheduled transfers of bulk data  DataSync: define a connection btw a data store on AWS (EFS, S3, FSx) and on prem  - DataSync agents installed on local file systems/obj stores to securely replicate objs to and from AWS. – DataSync can also work btw diff data stores on AWS  DMS: migrate relation DB, data warehouses, NoSQL DB to S3 or other AWS DB services  - Need install agent on database server  - DMS can capture a snapshot OR ongoing changes to source data as change data capture (CDC)  S3 Transfer Acceleration: provides shortcut for data over public internet.  - Good for distributed apps distant from their target region. – Makes use of AWS Edge Location  Snowball: physical device to upload petabytes of data -> physically shipped to AWS datacentre | | | | |
| Extracting Data on AWS | | | S3: - programmatically transfer data from S3 using CLI or SDK  - ETL data using Glue. – Replicate objects to FSx or EFS using DataSync  - Use S3 as direct data source from SageMaker Data Wrangler  - EFS -> DataSync -> S3. - EBS -> Gateway Endpoint -> S3. - RDS/DynamoDB -> DMS -> S3  - DynamoDB -> Glue -> S3 | | | |
| S3 vs EFS vs FSx | S3: migrate and collect raw data. – Stage data after transformations. – Supply training data directly to training EC2 instances  EFS: Fully managed, scalable file system for Linux-based EC2 instances or containers  - Uses NFS v4 protocol. – Support thousands of connections w/o impacting performance  - Not as cost effective as S3, not as performant as FSx = only recommended for training data if data already resides in EFS  a) FSx for Lustre: open-source file system designed for High Performance Compute environments  - Can scale up to TB/s throughput and millions of IOPS  - Integrate and sync w S3 to deliver data faster for training. – Leverage S3 for cold storage  b) FSx NetApp ONTAP: specifically for NetApp’s ONTAP file system  - Provides an in-VPC access point to data loaded from an ONTAP server  - Use S3 protocol for reads, NFS protocol for writes  c) FSx for Windows File Server: Supports Service Message Block (SMB) protocol used by Windows servers  - Only recommended if a shared file system needed for Windows-based EC2 apps  Use FSx for Lustre over S3 for training for large & complex data sets, to train on multiple nodes w distributed data parallelism, extremely compute-intensive training  S3 Express One Zone: single-digit millisecond latency vs FSx for Lustre: sub-millisecond latency (more ex)  S3 <- Read through less frequently access data – FSx – Mount to provide frequently access data -> EC2 | | | | | |
| Merging Datasets from Multiple Sources | From High operational overhead + Code-heavy but highly customizable -> Low operational overhead but less customizable: EMR -> Glue -> SageMaker Data Wrangler -> Glue DataBrew  1) EMR: managed cluster for running big data operations.  - Allows for simplified ETL for large amts of data into and out of AWS data stores  - Leverage Hadoop framework for ETL (Hive, Spark) and for Data Analysis (Spark MLlib, presto)  Streaming data -> Distribute task across EMR Cluster -> Nodes transform data -> Output to S3  2) Glue: provides more built-in features for data discovery, connectors, job monitoring, and orchestration compared to EMR. – Also uses Spark.  – More operational efficient than EMR but more expensive  - Data source -> AWS Glue Crawler -> Glue Catalog -> ETL scripts and jobs -> Output results  3) SageMaker Data Wrangler: pre-built no-code data transformations to prepare data for ML  - Designed for ML engineers. – Write custom transformations w pandas, SQL or PySpark  - Designed for ML data pipelines  4) Glue DataBrew: visual data preparation tool in AWS console to enable data analysts  - Provides no-code solutions for validating and detecting anomalies in source data  - Provides over 250 transformations to clean and normalize source data  - Visually map data lineage. – Can define and reuse transformations, and apply automatically  - Remove/replace missing values, Combine datasets, Create cols, Filter, Label mapping, Aggregate data | | | | | |
| Data Analysts use Glue DataBrew for Data exploration, cleaning, visualization, and pre-processing. Defining data quality rules and mapping data lineage  Machine Learning Engineers use SageMaker Data Wrangler for Data exploration, visualization, and feature engineering for ML workloads | | | | | |
| Amazon QuickSight: managed service to build dashboards to share w stakeholders & users external to AWS  - Easiest way to perform simple ML forecasting w little operational overhead | | | | | |
| Streaming Data for Inference | | | | | | Data -> KDS -> Flink -> SageMaker endpoint |
| Monitoring | | | 1) CloudWatch Metrics: give insights into performance & health of storage resources  2) CloudWatch Logs: gain insights for potential issues for data ingestion and storage  3) Service Dashboards: Integrate w CloudWatch Metrics to monitor | | | |
| 1) EBS Metrics: - VolumeReadBytes/ VolumeWriteBytes (total num of bytes read from volume)  - VolumeReadOps / VolumeWriteOps (total num of read/write operations from volume)  - VolumeThroughputPercentage (% of provisioned IOPS used by volume)  - VolumeQueueLength (num of outstanding I/O requests waiting to be completed)  2) S3 metrics: – BucketSizeBytes (total size of all objs). – NumberOfObjects  - AllRequests (total num of requests made to bucket). – GetRequests (num of successful GET requests made to bucket). – PutRequests  3) EFS metrics: - StorageBytes (total storage used). – PercentIOLimit (% of throughput limit used). – TotalIOBytes (total num of bytes read & write to file system)  4) FSx metrics: - DataReadBytes (total num of bytes read). – DataWriteBytes  - ThroughputCapacity (total throughput capacity). - PercentThroughputCapacityUtilized | | | |
| Scalable Data Ingestion | | | To keep streaming ingestion scalable:  - Use JSONL file format to efficiently stream data w diverse structures  - If Kinesis throughput is bottleneck = increase num of shards on Kinesis stream  - Partition data dynamically as it is delivered to S3 using KDF  - Batching w Amazon Kinesis (less calls). – Compress data but incur CPU costs  To keep batch ingestion scalable:  - Automate data preprocessing w Step Functions or SageMaker Data Pipelines  - Leverage scalable managed services for processing like Glue and SageMaker Data Wrangler  - Use scheduled scaling for preprocessing steps that take place on SageMaker or EMR | | | |
| Pre-training Processing | | | Ingestion -> Optional preprocessing -> Data Wrangler -> SageMaker Feature Store -> Training/Inference  SageMaker Feature Store:  - Take processed data and centrally store and managed derived features  - Use offline storage to store historical training data sets  - Use online storage for features used in real-time inference  - Reuse data in multiple ML apps, track data lineage and centralize feature security | | | |
| Cleaning Dirty Data | | Missing data, duplicated, extreme outliers, inconsistent data types or formats  Causes: Human error, duplicate entries, data integration from diff sources, lack of validation  DataBrew and Data Wrangler can deduplicate data | | | | |
| - Missing At Random (MAR): Missingness can be explained by values of other (observed) features in dataset, but not by missing values. (e.g. student absent for test could be due to age/health)  - Missing Completely At Random (MCAR): Independent of both observed and unobserved values, can be considered as randomly distributed. (e.g. technical glitch cause drop out of survey)  - Missing Not At Random (MNAR): Prob of data missing depends on value of variable that is missing. (e.g. individual w higher income less likely to report their income, so income is missing)  SageMaker Autopilot provide methods to fill missing values | | | | |
| Outliers | | | Use scatter plot to visually detect data. Can drop outliers or normalize data  Amazon Managed Service for Apache Flink uses Random Cut Forest to detect outliers (randomly cut data, the more cuts that isolate a data point = more likely point is an outlier)  Legitimate outliers should be included in cleaned data, but can affect algos sensitive to outliers  - Standardizing and normalizing data into normal distribution can reduce impact of outliers | | | |
| Feature Engineering | | | | - Turn categorical data into numerical data using techniques that preserve richness of data  - Engineer numerical features to mitigate outliers or reduce complexity of computation | | |
| Feature Normalization: E.g. Min max scaling values to 0 to 1 (relative distance not affected)  Feature Standardization: Mean = 0, s.d. = 1  Binning: put a few data points into the same bin. Compress continuous scale to discrete (e.g. quantile binning = each bin ≈ same num of data points)  Log transformation: normalize skewed numeric data | | |
| 1) One-hot encoding: each category map to a num (not ordered, create multiple cols)  2) Binary encoding: convert to 0 and 1  3) Feature splitting: split data into smaller parts (dates, address)  4) Label encoding: map category to a num (ordered by maintaining relative r/s)  5) Tokenization: break word/phrase into tokens | | |
| Bag-of-words: track freq of each word  N-gram: builds off of bag-of-work by producing a group of words of n size  Temporal data: from date -> day, month, year, season | | |
| Selecting Features | | | Feature splitting, Feature combining  PCA: Reduce dimensionality of feature set. Improves training time  - Reduce weight or eliminates redundant features which correlate strongly w other features | | | |
| Types of Viz | | Methods: r/s analysis (correlations), distribution analysis, comparison, composition (identify biases by visualizing proportions, percentages and counts within the data)  Categorical data: use bar charts (comparison analysis), pie charts (composition), heat maps (r/s)  Numerical: scatter plots (r/s), histogram (dist), density plots (dist), box plots (dist)  Histogram: Used to visualize freq dist over a continuous variable (“binning” a continuous var)  Density Plot: Visualize a dist of a variable over a continuous range (“smooth” histogram) | | | | |
| Automating Data Transforma-tion | | | | SageMaker Data Wrangler Flow: if need to transform data w multiple steps (read from multiple sources, apply transformations, joins, output to S3 bucket)  Step Functions: orchestrate data pre-processing workflows that integrate many AWS services  SageMaker Pipelines: use any events (new obj event or scheduled event) to trigger pipeline (data prep, data validation, feature definition, model training, model testing, model deploy) | | |
| Transforming Streaming Data | | | | | Use Firehose + Lambda when applied transformations are small and simple  Use AWS Glue Streaming jobs when workload is inconsistent or unknown  Use Amazon EMR for huge amt of data in continuous or predictable workloads | |
| Creating & Storing Features | | | Amazon SageMaker Feature Store: store & manage features for ML training and inference  1) Define feature groups that contain a logical grouping of features  2) Access and ingest records w simple API calls  3) Control access to feature groups for ML workloads | | | |
| Types of Feature Group Store Configurations:  1) Offline Store: store feature groups in S3 for batch processing such as model training  2) Online Store: provides low-latency access to real-time feature data for ML inference | | | |
| Labelling Data | | | Amazon Mechanical Turk: leverage an external pool of workers for scalable data labelling, text annotation, data collection, data cleanup, transcription  SageMaker Ground Truth: manage human-in-the-loop tasks in your ML lifecycles  - Assign internal teams or commission Mechanical Turk for data labelling  SageMaker Ground Truth Plus: create a private workforce in Amazon Augmented AI (Amazon A2I), where you can add own employees as workers.  - More for production labelling workflows, sensitive data, more AI models to assist in labelling | | | |
| Data Imbalance | | | Bias: any situation where results are skewed for or against an outcome for a particular class  - Originates in training data. – Mitigate by observing bias metrics | | | |
| 1) Class Imbalance (CI): compare categories/facets to determine if a given facet is over/under-represented in data  - Class Imbalance = 0 = equal representation. – CI value near -1 or 1 = very imbalanced  - E.g. 10% cat A, 90% cat B. CI = 0.1 – 0.9 = -0.8  2) Difference in Proportions of Labels (DPL): imbalance of +ve outcomes btw diff facet values  - If ratio of +ve outcomes for these facets differs significantly in training data = could have bias  - E.g. Cat A (70% +ve, 30% -ve). Cat B (20% +ve, 80% -ve). DPL = 0.7 – 0.2 = 0.5  3) Total Variation Distance (TVD): compares the diff in outcome distribution for 2 facets in scenarios where outcomes are binary, multicategory, or continuous  - Produce a value from 0 to 1 that indicates the ratio of facet A outcomes that would have to change for facet A to match facet B (0 = identical, 1 = disjoint dist)  4) Kullback-Leibler Divergence (KL)/relative entropy: measure divergence of label distributn btw 2 facets  - Values near 0 indicate similar dist, large values indicate larger divergence | | | |
| Validate Data | | | Glue Data Quality: set data quality rules, automated scheduling, dashboards  Glue DataBrew: visual data prep tool, data profiling, transformations | | | |
| Detecting Bias | | | SageMaker Clarify: detect bias and explain ML and GenAI models  - Provides features to help build less biased and more explainable models  - Generates model governance reports for analysis by external teams or auditors  - Bias Analysis by running a SageMaker Clarify processing job on data  - Monitor and detects data drift, outlier detection, generate data insights | | | |
| Mitigating Class Imbalance | | | 1) Synthetic Minority Oversampling Technique (SMOTE): Synthesize instances of minority class  - Only choose this if actual class-balanced data not available  - Superior to random undersampling techniques as no loss of data  - Adds more variance than random oversampling (reduce chance of overfitting)  2) Generative Adversarial Network (GAN) Data Augmentation (for images)  - generate image variations based on training data  - 2 NN “compete”, one trying to trick the other w synthesized images | | | |
| Securing Sensitive Data | | PII (Personally identifiable information). PHI (protected health info): subset of PII  Data residency: geographical local where data must be stored and processed | | | | |
| Amazon Comprehend: NLP service used for sentiment analysis, entity recognition  - Can be used to find & redact PII. – Configure as an asynchronous batch job / real-time analysis  Amazon Macie: continuously scans S3 buckets in all accounts for sensitive data  - Define rules to determine severity of identified sensitive data  AWS Lake Formation: mask sensitive data to prevent human access on granular level  EBS: enable EBS volume encryption. RDS: customer managed keys or AWS managed keys | | | | |
| Encryption | | | S3 Server-side Encryption (encryption at rest):  1) S3 Managed Keys (SSE-S3): Enabled by default. Use 256 bit Advanced Encryption Standard  2) KMS Keys (AWS or customer managed) (SSE-KMS):  - View and edit control policies for individual keys. - Track key usage in AWS CloudTrail  3) Customer-provided Keys (SSE-C): Full control of encryption Keys  - Keys can be managed on prem  Redshift Encryption: KMS AWS-Managed Key OR KMS Customer Managed Key  ElastiCache: encryption for data at rest. SageMaker: AWS KMS to encrypt data at rest | | | |
| To access KMS encrypted data: Read/Write Permissions + Encrypt/Decrypt Permissions | | | |
| Preparing data for training | | | 1) Shuffling Data: - random permutation  - epoch-based shuffling: divide dataset into epochs (complete passes through dataset) and shuffling data btw each epoch  - mini-batch shuffling: shuffle data points within each mini-batch before giving batch to model  2) Splitting data: training, validation, testing (typically 80% for training, 20% for validation + test)  - Randomized splits shuffle data w no regard to feature dist  - Ordered splits: take sequential portion of records sorted by attributes like time  - Split by key: selects ≥ 1 attributes and ensures only unique combinations occur in ea data set (e.g. by user ID, ID in training dataset will NOT appear in validation set)  - Stratified splits: ensures similar distribution of a given attribute in training and validation sets  3) Augment Data: depends on composition and bias metrics of training data  - To address class imbalance and label imbalance  - image: flip, rotate, scale. – text: synonyms, paraphrase. – scaling time axis, jitter | | | |
| Can use SageMaker Data Wrangler Flows: data -> Data Wrangler random or stratified split -> Training data + Validation data -> SMOTE data augmentation -> Augmented training data | | | |
| Loading Data from Training Data Source | | Training data sources: S3, EFS, FSx for Lustre  1) S3 File Mode: loads all data into EC2 instance filesystem  - Good if dataset is small and can easily be stored on instance’s storage space  - Or if random access to data is required  2) S3 Pipe Mode: allows for faster startup times and throughput than File Mode  - Only need enough disk space to store the final model artifacts  3) S3 Fast File Mode: initializes by identifying the files for training, but don’t download them  - Files are streamed on-demand for faster start-up (better than Pipe mode)  - As files are streamed, instance volume don’t have to accommodate the whole data set  - S3 -> Stream -> FUSE process -> Instance filesystem <- Training script  4) FSx for Lustre: provides a mount point for EC2 instances for fast delivery of frequently accessed data  - For less frequently accessed data, it reads through from S3  - Can handle hundreds of GBs of throughput & millions of IOPS for super-low-latency file retrieval  - Requires additional setup, as need to connect to VPC where file system resides  5) EFS: mount EFS to EC2 training instance  - Training data must already reside in EFS to mount file system to training instance for training  - Supports data parallelism across thousands of EC2, but not as performant as FSx for Lustre  - OR can use DataSync to move data from EFS to S3, then use File Mode or Fast File Mode | | | | |
| Most SageMaker algo support training w data in CSV or RecordIO-protobuf (for images) | | | | |

EMR: primary node (manages cluster, jobs), core node (coordinate data storage), task nodes (compute; can handle interruptions)

- Primary & core nodes: on-demand instance. Task node: spot instance

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ML Model Development: 1) Choose a modelling approach. 2) Train & refine models. 3) Analyze model perf | | | | | | | |
| Intro | | | | | Out of the box -> More control: AI services -> ML services -> ML framework and infrastructure | | |
| AI services | | 1) Vision: Amazon Rekognition (only obj detection & classification)  2) Speech: Amazon Polly (text to speech). Amazon Transcribe (speech to text)  3) Text: Amazon Comprehend (NLP to extract insight from docs). Amazon Translate. Amazon Textract (OCR)  4) Search: Amazon Kendra (NLP to return answers to search questions from your data)  5) Chatbots: Amazon Lex V2 (NLU: natural language understanding + ASR: auto speech recognitn)  6) Personalization: Amazon Personalize (personalized recommendation for users)  7) Forecasting: SageMaker Canvas (create time-series forecasting models)  8) Fraud: Amazon Fraud Detector  9) Development: Amazon CodeGuru Security (detect security vulnerabilities in code base)  10) Contact Centers: Amazon Connect Contact Lens (call center monitoring)  11) Generative AI: Amazon Q (suite of gen ai services), Amazon Bedrock (provide foundational models through a unified API)  - Amazon Q Business, Amazon Q Developer, Amazon Q in QuickSight, Amazon Q in Connect, Amazon Q in AWS Supply Chain) | | | | | |
| ML services - SageMaker | | | | | SageMaker Studio: IDE for ML (JupyterLab, Code Editor, RStudio)  SageMaker Notebook instances: TensorFlow, PyTorch, mxnet  - A SageMaker Jupyter notebook instance is a ML compute instance running the Jupyter Notebook app. SageMaker manages creating the instance and related resources  - SageMaker notebook instances initiate Jupyter servers on EC2 and provide preconfigured kernels w packages (Boto3, AWS CLI, Conda, Pandas, Deep learning libraries, …) | | |
| |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | |  | Notebooks | ML training | | | | Deep Learning | Inference | | Instance Family | t family | m family | r | c | p | g | Elastic inference | | Workload | Short jobs and notebooks | Standard CPU to memory ratio | Memory optimized | Compute optimized | Accelerated computing training & inference | Accelerated inference, smaller training jobs | Cost-effective inference accelerators | | | | | | | | |
|  | CPU for smaller models, GPU for computationally intensive tasks involving matrix operations  Trainium instances: cost effective solution for training large scale model | | | | | | |
| Model Types | 1) Generalized linear models: linear/logistic regression, SVM  2) Tree-based models: boosted trees, random forests, decision trees  3) Neural-networks: model nonlinear r/s, more diff to interpret  4) Clustering: k-means, DBSCAN. 5) Matrix factorization: PCA  6) Forecasting: ARIMA (autoregressive integrated moving average). 7) CV. 8) Seq models: RNN, LSTM  SageMaker model registry: organize & manage models for reproducibility, governance & compliance  Amazon SageMaker Neo compiles models for faster inference | | | | | | |
| SageMaker built-in algos | | | | 1) Supervised learning: linear learner algos, XGBoost, KNN, Factorization Machines Algo (capture interactions btw features), Object2Vec algo (for embeddings)  2) Unsupervised learning: K-Means, Latent Dirichlet Allocation (LDA; find topics in doc) algo, Random Cut Forest (RCF; find outliers) algo, IP insights (associate IPv4 addr w entities), PCA  3) Text/speech: BlazingText (optimized Word2vec), Seq-to-Seq (translation), Neural Topic Model (NTM; topic modeling)  4) Images, video, timeseries: ResNet (image classification), Single Shot multibox Detector (SSD; object detection), Semantic Segmentation algo, DeepAR Forecasting algo (time series)  5) Reinforcement Learning (RL): require DL framework (TensorFlow, MXNet) + RL toolkit (manages interaction btw agent & environment; Coach and Ray RLlib toolkits) + RL env | | | |
| SageMaker Autopilot | | | | | Part of SageMaker Canvas: Automate process of building and deploying ML models (AutoML)  - data analysis & processing, model selection, hyperparameter optimization, model training & evaluation, model deployment | | |
| SageMaker JumpStart | | | | | ML Hub w foundation models, built-in algos, and prebuilt ML solutions to deploy w a few clicks. Can evaluate FM quickly based on pre-defined quality metrics  - Pre-trained models from model hubs (TensorFlow Hub, PyTorch Hub, Hugging Face, MxNet) | | |
| JumpStart industry-specific solution: - Demand forecasting. – Credit rating prediction. – Fraud detection. – CV. – OCR. – Predictive maintenance. – Churn prediction. – Personalized recommendation. – Healthcare. – Financial pricing. – Causal inference | | |
| Amazon BedRock | | | | | Fully managed service offering FMs (more hands off than JumpStart)  - Amazon Bedrock Agents: automate complex tasks and orchestrations  - Bedrock Knowledge Bases: integrate company data w query (create S3 bucket -> create knowledge base in Bedrock & select S3 as data source -> Ingest data to knowledge base)  - Bedrock Guardrails: additional layer of safeguards on top of FM  - Can further fine tune FMs  - Can use using AWS Console, Bedrock API, Custom Model Import feature | | |
| Model Selection Considera-tions | | | Interpretability + performance: Linear regression -> Decision tree -> KNN -> RF -> SVM -> NN | | | | |
| Interpretability strategies: 1) interpretable model approximation (use simpler model to approx). 2) Local interpretability (LIME, Shapley). 3) Modular design (decompose complex model into smaller components. E.g. simple model for feature selection, complex model for prediction). 4) Ensemble mtds | | | | |
| Cost: CPUs: more cost-effective, for tasks not requiring intensive computation power.  - use for data preprocessing, feature engineering, simple models  GPUs: parallel processing & tasks involving matrix operations like DL. – Higher cost | | | | |
| ML training | | | | | Loss functions: RMSE. Log-Likehood loss / cross-entropy loss = -(y log p + (1-y) log(1 -p))  Optimization technique: Gradient descent, stochastic GD, Mini-batch GD | | |
| Compute Environments | | | | | | | Traditional ML Training & Inference: M, C, R family  Deep Learning Inference: Inf1, G family. DL Training: P, D family, Trainium instances |
| Container Services | | | | | ECS: container orchestration services  EKS: fully managed Kubernetes control plane and integration w other AWS services  Fargate: serverless compute engine, simplifies deployment and management  ECR: store container images | | |
| SageMaker managed container images: script mode = use managed images, but use own training algo  Customer-managed container images: build own custom container | | |
| SageMaker Script Mode | | | | | Provides flexibility to develop custom training and inference code while using ML frameworks  SageMaker built-in algo less flexibility  Bring Your Own Container if SageMaker provided ML frameworks not enough | | |
| 1) Use local machine w SageMaker python SDK.  2) Write training script (import libraries, data preprocessing, model definition, training fn, hardware reservations and resource allocation, hyperparam tuning, logging)  3) Create a SageMaker estimator obj, specifying training script, instance type, other configs  4) Call *fit* mtd on estimator, passing in training and validation data channels  5) SageMaker takes care of the rest. Pulls images from ECR and load on managed infra  6) Monitor training job and retrieve trained model artifacts once job completed | | |
| AWS managed pre-built containers: already available in ECR. Includes SageMaker Training Toolkit, TensorFlow, MXNet, PyTorch, Chainer | | |
| Reduce training time | | | | | 1) Early stopping: stops training when performance on validation stops improving. To avoid overfitting  a) Evaluate metric after each epoch. b) Compare to running median of previous training jobs  c) Stop current job if performing worse than median  2) Distributed training: scale training across multiple compute resources  a) Data parallelism (most common): split training set into mini-batches evenly distributed across nodes. Each nodes only trains model on a fraction of the total dataset  b) Model parallelism: split model up btw multiple instances | | |
| Data parallelism: each device gets a replicated copy of the model and compute forward and backward passes on its assigned data batches. After each batch, resulting gradients are aggregated across devices before updating model weight  Components: - Global batch size: total batch size used = sum of per-replica batch size  - Per-replica batch size: batch size assigned to each model replica  - AllReduce: operation to aggregate and average gradients from all devices  - SageMaker distributed data parallelism (SMDPP) library | | |
| Model parallelism: diff parts of the model params or weights are assigned to diff devices.  - Techniques like tensor parallelism split individual large matrix multiplications across devices by splitting the tensors (weights / activations) of those layers  - SageMaker model parallelism library v2 (SMP v2) implement model parallelism  -- Hybrid sharded data parallelism: combines data parallelism and model parallelism  -- Tensor parallelism: diff parts of a single tensor distributed across multiple devices  -- Activation checkpointing: saves memory by recomputing activations during backward pass instead of storing all activations during forward pass  -- Activation offloading: saves memory by offloading activations from GPU memory to CPU memory or disk during forward pass. Loads activations back onto GPU during backward pass  - SMPv2 is compatible w PyTorch Fully Sharded Data Parallelism (FSDP) | | |
| Integrating External Models into SageMaker | | | | | External models can be purchased in AWS Marketplace or trained yourself  AWS Marketplace: - task-specific ML resources, - Model subscriptions, - Pre-trained models for deployment, - different pricing packages | | |
| To use own model w SageMaker, need to build a model package  Package defines a Docker container including: - framework to run model, - location of model artifact, - code for performing inference w model, - other libraries and dependencies  Deployment options: 1) Pre-built containers (SageMaker provided)  2) Create custom containers (need to be SageMaker compatible)  Steps: 1) Upload trained model to S3. 2) In inference.py: need load the model artifact  3) Create a model folder with the inference.py, requirements.txt, model and then save in .tar and compress with gzip. So final folder is model.tar.gz | | |
| Bias-Variance Tradeoff | | | | | High model bias causes: simple model, incorrect modelling/feature engineering, inherited bias from training dataset  High model variance: complex model, too much irrelevant data in training dataset, overfit | | |
| Prevent overfit & underfit | | | | | 1) Early stopping. 2) Pruning (remove redundant features). 3) Regularization (dropout, L1 – weights can become 0, L2 – weights reduced). 4) Data augmentation. 5) Model simplification  To prevent underfit: 1) Train for enough time. 2) Use enough data. 3) Increase model flexibility (add new features, add cartesian products/interaction effect, change feature engineering, regularization) | | |
| Model Combination | | | | | 1) Boosting: ensemble technique to train models sequentially, w each new model attempting to correct the errors of previous one  - Final prediction combine outputs of all model. - Help mitigate under and overfitting  - Adaptive Boosting (AdaBoost): for classification. Assigns weight to data that is misclassified  - Gradient Boosting (GB): classification & regression. Optimize the loss function in ea iteration  - Extreme Gradient Boosting (XGBoost): optimized version of GB. Use multiple cores on CPU for parallel training. Also do cache optimization, and out-of-core processing  2) Bagging/bootstrap aggregation: combines models trained on diff datasets  - Random forest: combines bagging w random feature selection  3) Stacking: predictions generated by multiple base models, aka level-0 models. These predictions then fed to a level-1 model/meta-model, which combines the predictions | | |
| Hyperparam tuning | | | | | Gradient descent: learning rate, batch size (num of data used in each iteration of GD before model weights updated), epochs (num of passes through entire dataset)  NN: num of layers, num of neurons in ea layer, activation fn, regularization techniques  Decision tree: max depth of tree, min samples to split a node, criterion (Gini impurity = likelihood that data could be misclassified, Entropy = randomness of data) | | |
| 1) Manual selection: based on intuition, domain knowledge, prior experience  2) Grid search: evaluate all possible hyperparam values  3) Random search: randomly select sets of hyperparam values  4) Bayesian optimization: uses performance of previous hyperparam selections to predict which subsequent values likely to yield best results  - Faster than random search. But only works sequentially; hard to scale  5) Hyperband: dynamically allocates resources to well-performing configs & stops underperforming ones early  - Exploration phase: models trained within a range of hyperparam configs using fixed num of epochs. Exploration stops when: -- Specified num of epochs reached  -- performance improvement is observed. Stop at this point to prevent overfitting  - Exploitation phase: validation metric used to identify worst-performing configs. Half of the initial configs discarded, and remaining configs allocated additional epochs for next round  - Can train multiple models in parallel. More efficient than grid or random search  - But should only be used for iterative algos like NN, where intermediate results after ea epoch can be used to guide the tuning process | | |
| SageMaker Automatic Model Tuning (AMT): perform hyperparameter tuning | | |
| Model Size Reduction Techniques | | | | | 1) Pruning: remove least important params or nodes from a model  2) Quantization: change representation of weights to more space-efficient (to 8-bit)  3) Knowledge distilling: larger teacher model transfers knowledge to a smaller student model. Student model trained on same dataset as teacher, & also trained on teacher model’s knowledge of data | | |
| Fine-tuning pre-trained models | | | | | Fine tuning is a subset of transfer learning. Fine-tuning approaches:  1) Domain adaption: adapt FM to specific tasks by using limited domain-specific data  2) Instruction-based: Use labelled examples formatted as prompt-response pairs and phrased as instructions to improve performance of FM on a specific task  Can fine tune using SageMaker JumpStart or Amazon Bedrock | | |
| Catastrophic Forgetting Prevention | | | | | Occurs when model is trained on new data, and it forgets previously learned knowledge.  To detect: 1) Plot model performance over time. 2) Check validation sets are representative of historic patterns in data that are still relevant  To prevent: 1) Elastic weight consolidation (EWC): regularization technique that predicts which weights are impt to previously learned tasks. Adds a penalty term to loss fn that protects these weights  2) Rehearsal: include samples from original training set during fine-tuning  3) Model design: Design model w appropriate complexity to learn and retain patterns  4) Renate: open source Python library for auto model re-training of NN (incrementally train) | | |
| SageMaker Model Registry | | | | | Manage model versions, control approval status of models within ML pipeline  1) Register model version on Registry. 2) Approve model for deployment  3) Model Groups: contain diff versions of the same model. 4) Collection: higher level organizational units containing multiple Model Groups  5) Deploy model to production. Can automate w CI/CD pipeline | | |
| Performance Baselines | | | | | 1) Establish appropriate evaluation metric. 2) Use simple model performance as baseline  3) Ensure evaluation datasets are representative and don’t contain any leakage or bias | | |
| Performance Metrics | | | | | Validation set to identify where to improve model. Test set to predict model performance  Regression metrics: MSE, RMSE, R-Squared, Adjusted R-squared  Classification metrics: Accuracy, Precision = TP/(TP+FP) (when cost of FP is high),  Recall/Sensitivity/TPR = TP/(TP+FN) (when cost of FN is high), F1 = 2\*P\*R/(P+R), AUC-ROC (uses sensitivity and specificity = TNR; higher better; plots outcome at diff threshold)  Specificity = TN/(TN+FP) | | |
| Convergence Issues | | | | | | Vanishing or exploding gradients, over or underfitting, local minima or saddle points, …  Amazon SageMaker Automatic Model Tuning (AMT) can help mitigate convergence issues, and auto tune hyperparam.  Amazon SageMaker Training Compiler: optimize model training on GPU instances | |
| Debug Model Convergence | | | | | | SageMaker Debugger: monitor and debug ML models during training and deployment  - improve model performance, explainability and bias detection, alerts, debugging  - Can stop training jobs when non-converging conditions detected | |
| SageMaker Clarify | | | | | | Monitor, explain and improve fairness, bias and transparency of ML models  Data bias metrics: Class Imbalance, Facet Imbalance, Facet Correlation  Model bias metrics: Differential validity (diff in model performance across diff facet groups), Differential prediction bias (diff in predicted outcomes for diff facet groups, given same input features), Differential feature importance  Model explainability metrics: SHAP (Shapley Additive exPlanations = contribution of each feature in a prediction), Feature Attribution, Partial Dependence Plots (PDPs = diff in predicted outcome as an input feature changes)  Data quality metrics: Missing data, duplicate data, data drift | |
| SageMaker Experiments | | | | | | Organize, view and perform reproducible experiments. Streamline ML experimentation work  - Track inputs, params, configs, results, artifacts of iterations as runs | |

SageMaker Warm Pools: retain & reuse provisioned infra after completion of a training job to reduce latency/start up times for repetitive workloads. Training jobs w same params can run on the retained warm pool infra

Temperature: randomness of output (higher = more random, lower = more consistent)

Top\_k: number of top probable tokens to consider when generating response (lower = less tokens to consider = more consistent)

Top P: max cumulative probability threshold to sample tokens from. If < 1, model choose most probable tokens and ignores uncommon tokens

SageMaker TensorBoard: visualization tool to monitor loss and gradient. SageMaker Debugger: monitor vanishing/exploding gradient, underutilized GPU, overfitting

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Deployment and Orchestration of ML Workflows:  1) Select deployment infrastructure based on existing architecture and requirements  2) Create and script infrastructure based on existing architecture and requirements  3) Use automated orchestration tools to set up CI/CD pipelines | | | | | | | |
| Deployment Infra | | | 1) Model building and deployment infra: data processing, feature engineering, model training, validation, optimization. – Requires GPUs for quickly training complex models  - Output final trained model artifacts that will be used for inference  2) Inference infra: receive new unseen data, run it through model, return predictions  - Focuses on low latency, high throughput inference serving and scales to handle high query volumes w/o affecting latency. – Can be hosted on cloud, on prem or edge locations | | | | |
| Challenges: 1) Data quality and preprocessing. 2) Model monitoring and maintenance  3) Scalability & performance. 4) Infra and DevOps. 5) Continuous delivery and updates | | | | |
| MLOps framework | | 1) Version control. 2) CI/CD: ensure changes to codebase is auto tested, validated and deployed to production env. – AWS CodePipeline orchestrates build, test, and deployment process. – AWS CodeBuild provides fully managed build service to compile code, run tests & package the artifacts  3) ML model builder: use SageMaker Training Jobs  4) ML model deployment: containerize, integrate w app or service. SageMaker to host model  5) Monitoring: performance of model. SageMaker Clarify  6) Workflow security: IAM policies. 7) Model registry: SageMaker model registry  8) Testing and evaluation: unit test, integration test, performance test + evaluation of model performance. Can use Lambda to run unit/integration/performance tests as part of CI/CD | | | | | |
| Orchestrating ML workflows | | | | 1) SageMaker Pipelines: create, automate, and manage end-to-end ML workflows at scale  - Integrates w SageMaker Feature Store, ECR, SageMaker Training, SageMaker Processing (DataWrangler, Clarify), SageMaker Model Registry  2) AWS Step Functions: serverless way to orchestrate pipelines, including ML pipelines  - Can add steps to use Lambda to test endpoint status or model performance  3) Amazon MWAA: orchestrates workflows using DAGs.  - Python-based tool, DAG workflow management, Extensibility  4) 3rd-party tools: - MLflow (tracking, projects, models, registry)  - Kubernetes (AWS Controllers for Kubernetes (ACK) & SageMaker Components for Kubeflow Pipelines) | | | |
| Deployment Considera-tions | | | Best practices: Auto scaling, Test out new model variant on a subset of production traffic before fully replacing original model, Use batch for large datasets for inference | | | | |
| Deployment targets: 1) SageMaker endpoints: fully managed, easy to deploy and monitor, minimal operational overhead. But not as customizable, potentially higher cost  2) EKS: container orchestration platform to deploy & manage ML models as containerized app  - high scalable and flexible/ - But higher operational overhead, steeper learning curve  3) ECS: fully managed container orchestration service. Easy deployment of Docker containers  - convenient to scale, integrate well w other AWS services. – But limited advanced features compared to Kubernetes, Vendor lock-in  4) AWS Lambda: - serverless, auto scale, low operational overhead, pay-per-use  - But limited runtime, cold starts affects latency, not suited for long-running/complex models | | | | |
| Model Inference Strategy | 1) Real-time inference: low latency, high throughput, fully managed, auto scaling  - Create SageMaker model by specifying model artifact and inference container image  - Define SageMaker endpoint config. – Create SageMaker endpoint  2) Serverless inference: intermittent traffic w/o need to manage infra. No need to specify infra  - On-demand SageMaker Serverless Inference: workloads w idle periods & can tolerate cold start  - Provisioned Concurrency w SageMaker Serverless: predictable bursts in traffic, can define min amt of inference capacity to keep endpoints warm  3) Asynchronous inference: queues requests and handles large payload (up to 1GB), long processing times (up to 1 hour), auto scaling  - Create async inference endpoint by specifying AsyncInferenceConfig object  - To call endpoint: provide a S3 URL in payload as part of InvokeEndpointAsync request  - SageMaker use SQS to queue request for processing and returns an identifier and output location as response. – SageMaker place the result in S3. – Can choose to receive success or error notifications w SNS  4) Batch transform: processes large offline datasets, serve predictions on scheduled basis  - Provide path to S3 where data to be transformed is stored. – Compute resource to use for transform job. – Path to S3 for output. – Name of SageMaker model to use to create inference | | | | | | |
| Multi-model deployment: uses a shared serving container, reduce hosting cost  - check utilization metrics (CPU, GPU or others) in CloudWatch for underutilization  - A/B testing: host diff model versions and compare performance  - Ensemble model: 1 container to host an ensemble of diff models, endpoint can then route requests to best-suited model  - Dynamic model loading: if have large num of models rarely used, load specific models needed  - Multi-tenant apps: diff customers require own model, serve multiple cust from same endpoint  Multi-container deployment: uses multiple frameworks, uses direct invocation  - Data preprocessing: run a data pre-processing container alongside primary model container  - Post-processing and interpretation: performs add tasks like generating explanations, formatting output, or combining results from multiple models  - Multi-modal inference: if ingest and process diff data modalities, each have specialized containers for each data type | | | | | | |
| Container & Instance Types for Inference | | For container: 1) SageMaker managed container images: - auto handle setup, model hosting, inference processing (has pre-built code). – If using SageMaker managed container image to train model, it might already include the inference logic  2) Custom inference code: - extend SageMaker containers OR Bring your own container (BYOC) | | | | | |
| Deployment w Edge Computing | | | AWS IoT Greengrass: performs inference locally on connected IoT devices  - multiple IoT devices -> AWS IoT Greengrass core device (Gateway device, IoT apps, AWS IoT Greengrass client software) -> AWS Cloud (AWS Greengrass cloud service)  - AWS IoT Greengrass core devices, AWS IoT Device SDK enabled devices, FreeRTOS devices can be configured to communicate w one another.  - If IoT Greengrass core device loses connectivity to cloud, connected devices can communicate w ea other over local network  - Manage and operate device fleets in field locally or remotely using MQTT or other protocols | | | | |
| Amazon SageMaker Neo: auto optimize ML models for inference on cloud instances and edge devices to run faster  - can convert compiled models btw frameworks or diff hardware  - compress model size: quantizes and prunes models to reduce size | | | | |
| Deployment Best Practices | | | AWS Well-Architected Framework: security, reliability, performance efficiency, cost optimization, sustainability, operational excellence (automation, repeatable code) | | | | |
| Automatic resource provisioning: reduce human error, release time & maintain compliance | | | | |
| Infrastructure as Code (IaC) | | | | | Components: 1) Infra definitions: specify components of infra, configs.  2) Shell scripts: automate infra tasks like install software, configure settings. 3) App code | | |
| Approaches to IaC:  1) Declarative: describe resources and setting that make up end state of desired system  2) Imperative: describe all the steps to set up resources and get to desired state  - Necessary for complex infra deployments, like when order of events is critical  - E.g. provisioning a SageMaker endpoint, first create ECR repo for Container image, S3, compute resources for hosting inference containers | | |
| 1) CloudFormation: model, provision & manage AWS and 3rd party resources  - Define and provision AWS resources using templates written in JSON or YAML  2) AWS CDK: define and provision AWS resources using TypeScript, Python, Java, C#. Serves as an abstraction layer to efficiently produce CloudFormation templates  3) Terraform: supports wide range of cloud providers. Can manage infra across multiple platforms using a consistent workflow and language  4) Pulumi: Supports TypeScript, JavaScript, Python, Go, Java, .NET, YAML. Can provision, update, manage cloud infra using a downloadable CLI, runtime, libraries & hosted service | | |
| AWS Cloud-Formation (CF/CFN) | CloudFormation Template to describe infra -> CFN provision resources -> Resources managed as a CFN stack (all resources must be provisioned completely before CFN reports build as successful) | | | | | | |
| Template: 1) Format version: “AWSTemplateFormatVersion”: “2010-09-09”  2) Description. 3) Metadata. 4) Parameters (values passed to template when you create or update a stack. Can refer to params from Resources and Outputs sections)  5) Rules (validate param values). 6) Mappings (specify conditional params)  7) Conditions (when certain resource are created). 8) Transform (for serverless apps, transform specifies version of AWS Serverless Application Model (SAM) to use)  9) Resources (compulsory). 10) Outputs (values to return when you view stack’s properties)  - Resources can refer to 1 another using the **Fn::GetAtt** function, or its short form, **!GetAtt** | | | | | | |
| In a CFN stack: if a resource can’t be created, CFN rolls back stack and deletes any resources created. If a resource can’t be deleted, CFN retains any remaining resources until entire stack can be deleted  To update: Updated template -> CFN -> Change set -> Updated stack  - Need to review change set to assess impact on running resources and approve it | | | | | | |
| Layered architecture: Ea layer has dependency on layer directly below it. Can have ≥ 1 stacks in each layer, but in each layer, stacks should include have w similar lifecycles & ownership  1) Identity layer: define IAM users, groups, roles. 2) Base network layer: VPC, Internet gateways, VPN, NAT gateways. 3) Shared resources layer: DB, common monitoring, subnets, security groups. 4) Back-end layer: model endpoints, Lambda, app servers. 5) Front-end | | | | | | |
| Communication btw stacks: Can share outputs from 1 stack w another stack  Network stack: “Outputs”:{“Subnet”:{“Export”:{“Name”:{“Fn::Sub”:”${AWS::StackName}-SubnetID}}}}  Front-end: “NetworkInterfaces”:[{“SubnetId}:{“Fn::ImportValue”:{“Fn::Sub”:”${NetworkStackName}-SubnetID”}}}] | | | | | | |
| Provisioning stacks: can do through AWS Console, or through AWS CLI  *aws cloudformation create-stack –stack-name teststack –template-body template.json \*  *-- parameters ParameterKey=Param1,ParameterValue=Value1 ParameterKey=Param2,…* | | | | | | |
| AWS CDK | 1) AWS CDK Construct Library: collection of pre-written modular and reusable code called constructs  - Construct represent infra resources and collections of infra resources  2) AWS CDK Toolkit: CLI tool for CDK apps. Use to create, manage and deploy CDK projects | | | | | | |
| A project in CDK is referred to as an *app*. It holds the diff groups of resources, or *stacks* that comprise your application. Add *constructs* = diff AWS resources to each *stack*. Lastly, synthesize your *app* to create CF templates for each stack | | | | | | |
| A construct = a component and encapsulates everything CF needs to create it. There are diff levels: 1) Lower-level constructs (L1 constructs/CFN resources): single resource from a single AWS service  - Use if need configure all resource properties to same level of granularity as a CFN template  2) Higher-level constructs (L2, L3): more complex component consisting of multiple AWS resources  - Represent individual AWS resources, but provide a level of abstraction  - Incorporate defaults, boilerplate, and glue logic needed when writing a CFN resource construct  - Provide methods to work w resources, like *Bucket.grant\_read(lambdafn)*  L3 constructs = patterns to declare multiple resources | | | | | | |
| 1) To begin CDK project, create a dir for it, run **cdk init**, and specify programming language used  2) Run **cdk bootstrap** to prepare the environments into which the stacks will be deployed  3) Synthesize CFN templates using **cdk synth**. To update existing stack, run **cdk diff**  4) Run **cdk deploy** to have CFN provision resources defined in the synthesized templates | | | | | | |
| CFN vs CDK | | | CFN simple to write but can be verbose.  Templates are declarative  Harder to debug (CFN own errors) Complex to write modular stacks  More community support | | | CDK can use own programming language  Imperative approach to generate declarative CFN templates  Easier to debug in your programming language  Easier to create modular IaC code blocks  Newer so less community support | |
| SageMaker Python SDK | | In Increasing abstraction: AWS API -> AWS SDK for Python (Boto3) -> SageMaker Python SDK  - More user-friendly to create and manage training jobs, deploy models, orchestrate workflow  - SageMaker Python SDK also manages most of config and provisioning of infra resources | | | | | |
| E.g. SageMaker pipeline. step\_process = ProcessingStep(..), step\_train = TrainingStep(..)  pipeline = **Pipeline**(name=”pipeline”, **parameters**=[input\_data, processing\_instance\_type, …],  **steps**=[step\_process, step\_train, step\_evaluate, step\_conditional])  from sagemaker.sklearn.processing import SKLearnProcessor  sklearn\_processor = SKLearnProcessor(framework\_version=…, role=IAM\_role, instance\_type=…)  step\_process = ProcessingStep(name=””, step\_args=sklearn\_processor.run(  code=”preprocessing.py”,  inputs=[ProcessingInput(source=”s3://bucket/data”, destination=”ml/processing/input”)],  outputs=[ProcessingOutput(output\_name=”train\_data”, source=”ml/processing/input”),  ProcessingOutput(output\_name=”test\_data”, source=…)],  arguments=[“—train-test-split-ratio”, “0.2”])),  - Use parameters to customize pipeline runs. – Use steps to orchestrate workflow | | | | | |
| For training models: instantiate an **estimator** class, then call **.fit()**  To deploy: either predictor = **estimator.deploy()** or **model.deploy()** (existing model). **predictor.predict()** | | | | | |
| Building & Maintaining Containers | | Training containers directory:  /opt/ml # Top level of directory  code/ # Training/preprocessing script  input/ # Folders for training input  config/ # JSON files for Hyperparams  data/ # Input data channels (S3)  checkpoints/ # Checkpoints buckets (S3)  output/  data/ # output.tar.gz  model/ # model.tar.gz | | | | | Inference containers:  - /opt/ml/model # Import model artifact and store in this folder  - Container must be configured to run as an executable. i.e. Dockerfile should have ENTRYPOINT  - Container web server must listen on port 8080  - Container must accept POST requests to the **/invocations** and **/ping** real time endpoints. Request send to these endpoints must return within 60 sec, and have size ≥ 6MB |
| For registry: use ECR  For orchestration tool: 1) SageMaker: for model training, can specify training container image and EC2 instance type. For SageMaker endpoint, can specify model to use in SageMaker Model Registry and inference container image  2) ECS: container orchestration service w a more managed model for deploying containers. Also have integration w other AWS services  3) EKS: managed service to run Kubernetes container orchestration on AWS  For container hosting: to decide which compute resource orchestration tool will use to host containers  1) AWS Fargate: serverless hosting service that auto allocates CPU and memory to support containers.  2) EC2: launch containers on diff instance types. More cost effective and more control over instance type | | | | | |
| Auto Scaling Inference infra | | | SageMaker model auto scaling: auto distributes instances across AZs. Auto scaling methods:  1) Target tracking scaling policy: E.g. use target metric **CPUUtilization** and target value of **50** percent  - Auto scale num of EC2 instances to maintain an average CPU utilization of 50% across all instances  2) Step scaling policy: choose scaling metrics and threshold values for CloudWatch alarms that invoke the scaling process. Define how target should be scaled when threshold is breached for a specified num of evaluation periods. Allows more configuration when demand reaches certain levels  - E.g. if invocations > 50 per instance -> add 1 more EC2. If invocations > 80 -> add 2 more EC2  3) Scheduled scaling policy: if know demand follows a particular schedule. Can specify a 1 time schedule or recurring schedule, or specify cron expressions  - Need specify start time when scaling should start. New min, max and desired size for scaling action  4) On-demand scaling: incr or decr num of instances manually  - New product launch, special promos or ad campaigns | | | | |
| DevOps | | | Code -> Build -> Test -> Provision -> Deploy -> Monitor. Continuous Integration (Code, Build, Test). Continuous Delivery (Code, Built, Test, Provision + Manual approval to continue Deploy and monitor)  Continuous Deployment (Full pipeline w/o manual approval to deploy) | | | | |
| MLOps | | | CI/CD (continuous deployment): Data pipeline (data prep) -> Building and testing pipeline (model build, evaluate, model selection) -> Deployment pipeline (deployment) -> Monitoring pipeline (monitoring)  Roles: DE, Data scientist, ML engineer  ML system requirements: consistency (minimal variance btw diff env like dev, prod), flexibility (accommodate a wide range of ML frameworks and tech to adapt to changing requirements), reproducibility (past experiments give same results), reusability (components that can be reused across diff ML projects), scalability, auditability, explainability | | | | |
| Automated Tests | | 1) Unit tests: validate smaller components like individual functions or methods  2) Integration tests: check that pipeline stages (data ingestion, training, deployment) work tgt correctly  3) Regression: re-running same tests to ensure something that used to work was not broken by a change | | | | | |
| 1) Implement automated tests: using frameworks or AWS pre-built tests to integrate w the pipeline and run automatically after code commits or model updates  2) Integrate w CI/CD: Use CodePipeline and Jenkins to automate tests as part of pipeline  3) Monitor and analyze results | | | | | |
| Use SageMaker Project template to create a SageMaker project which setup infra, select code repo, workflow automation tools like CodePipeline or Jenkins, and pipeline stages for model training, deploy... | | | | | |
| Version Control | | Git repo -> Git repo service. AWS Code Pipelines (Git repo service + AWS CodeBuild)  AWS CodePipeline: Integrates w Git repo service and CodeBuild (compile code, run tests, build artifacts) | | | | | |
| Continuous Deployment Flow Structures | | | Set of processes, tools, and practices that help w deployment of ML models to production  1) Model training and versioning. 2) Model packaging and containerization  3) Continuous integration (CI): automated build, test and validate ML model and its components, ensuring new changes don’t introduce regressions or errors  4) Deployment automation. 5) Monitoring and observability  6) Rollback and rollforward strategies: rollback in event of issues or performance degradation | | | | |
| Gitflow: Git branching model providing structured & standardized way to manage dev and deployment  Branches: 1) Main: production ready codebase. Latest stable, released version  2) Hotfix: quickly address and fix critical issues in production env. Branched off from main and merged back into both main and develop branch  3) Release: to prepare a new prod release & can include final touches (bug fix) before merging into main  4) Develop: active development and integration of new features. Central hub where all completed features are merged  5) Feature: short-lived branches created for development of new features. Typically branched off from develop branch and merged back after feature is completed | | | | |
| 1) Create new feature branch from develop branch. Develop and commit changes to feature branch  2) When feature completed, merge feature branch back to develop branch  3) When time to create new release, create a release branch from develop branch  4) Perform any necessary final adjustments (bug fixes) on the release branch  5) Merge release branch into main branch and tag w version num  6) Merge release branch into develop branch to incorporate changes  7) If issue found in prod env, create hotfix branch from main, fix issue, merge back to main & develop | | | | |
| GitHub Flow: branch-based workflow. Create branch -> Commit changes -> Open PR -> Review -> Deploy to prod | | | | |
| Continuous flow structures: organize and run a series of tasks in a pipeline fashion  1) Identify tasks. 2) Determine task seq. 3) Choose a continuous flow framework (Airflow, Luigi, Prefect). 4) Define pipeline. 5) Implement tasks. 6) Configure pipeline. 7) Test and validate. 8) Deploy and monitor. 9) Iterate and optimize | | | | |
| Continuous Delivery Services | | | Git repo -> AWS CodeBuild or 3rd party software -> AWS CloudFormation -> AWS CodeDeploy  Whole pipeline defines in AWS CodePipeline (integrates w CodeBuild, CodeDeploy, Elastic Beanstalk, ECS, Lambda, GitHub, Bitbucket, Jenkins).  - Can configure manual approval gates in pipeline to control release process  - CodePipeline provides visibility into status of pipelines, w logs and notifications (can monitor directly from CodePipeline console, AWS CLI, EventBridge, CloudTrail)  - Supports resource-level permissions (give diff permission to diff stages in pipeline) | | | | |
| CodeBuild: fully managed CI service that compiles code, run tests, and produce software packages that are ready to deploy. Supports both managed and custom build env using Docker images  - Configure build by using YAML buildspec file that defines build steps | | | | |
| CodeDeploy: automated deployments, flexible deployment strategies, rollback capabilities | | | | |
| CodePipeline limits: typically 1000 pipelines for ea AWS acct in a single AWS Region  - Num of actions in a single pipeline ≤ 500. – Size of input artifact for a single action ≤ 1GB (when stored on GitHub). – Num of custom actions for ea AWS Region in an AWS acct = 50  - Max num of webhooks for ea AWS Region in an AWS acct = 300  CodeBuild: - max num of concurrently running builds for env like ARM Lambda/10GB, and Linux Lambda compute types is 1 for ea AWS Region  - Num of security groups/subnets for VPC config for ea Region = 5 / 16  - Max num of build projects for ea Region = 5000. – Max build timeout = 2160 mins = 36 hr for ea Region  - Concurrent num of compute fleets = 10. – Concurrently running instances = 1  CodeDeploy: - max num of hours Lambda deployment can run = 50  - max num of apps associated w an acct in a Region = 1000  - max num of EC2 Auto Scaling groups in a deployment grp = 10 for ea Region  - Max num of deployment groups associated w a single app = 1000 for ea Region  - Max num of minutes a blue/green deployment can wait after successful deployment before terminating instances from original deployment = 2800  - Max num of minutes btw first and last traffic shift during a Lambda canary or linear deployment = 2880  - Max num of deployment groups that can be associated w an ECS service = 1 for ea Region | | | | |
| Trouble-shooting | | | Check IAM permissions, CloudWatch Logs (if set up), inspect source code or configs, check service limits, check network issues | | | | |
| Data Integration | | | Integration btw data ingestion and ML pipeline can be achieved using EventBridge to initiate Step Functions state machines when new data becomes available  Lambda fns act as bridges btw diff services in pipeline | | | | |
| Step Functions and CodePipeline | | | Step Functions: visual workflow service to build complex, distributed apps by automating and orchestrating various components.  - Each step aka as state – specific tasks, and entire workflow = state machine  - Can create workflows that process & publish ML models and build ETL pipelines  CodePipeline can be set up to invoke the MLOps pipeline when certain events occur (new model version committed to repo)  - CodePipeline can have stages for source code management, building, testing, deploying model  - When pipeline invoked, it can start the Step Functions state machine to initiate the MLOps workflow  - CodePipeline can pass input data or params to Step Functions state machine to configure running of the MLOps workflow | | | | |
| Deployment Strategies | | | 1) Blue/green deployment: maintain 2 identical production env, 1 blue (existing) and 1 green (new), and gradually shifting traffic btw them  - Facilitates safe rollback if new deployment fails. SageMaker can manage env and traffic routing  - Traffic shifting modes: All at once, Canary, Linear  - All at once: shifts all endpoint traffic from existing blue fleet to green 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 blue fleet after baking period  - Canary: Traffic shift to green in 2 steps. First, a small portion of traffic (the canary) is shifted to new fleet and monitored during a baking period. If canary succeeds, rest of traffic shifted from blue to green, and blue fleet terminated  - Linear: traffic shifted gradually in a pre-specified num of equal incremental steps. Num of steps and % of traffic shifted at each step can be customized.  - Baking period: set time to monitor green fleet’s performance before full transition. | | | | |
| 2) Canary deployment: roll out new model version to a small portion of users.  - Performance of new version is monitored in baking period. If performs well, traffic to new version is incr. SageMaker supports canary deployment through use of Endpoint Configuration & Traffic Routing | | | | |
| 3) Rolling deployment: replace previous deployment of model versions w new version by updating endpoint in a configurable batch size  - Similar to linear traffic shifting mode in blue/green deployment. But have benefit of reduced capacity requirements compared to blue/green (only some instance have new version)  - Fewer instances are active at a time. More control over how many instances to update in new fleet | | | | |
| Code repo w Pipelines | | | 1) Commit or Push event. 2) Invoke Event: code repo detects commit or push event and invokes the associated pipeline. Event usually configured in repo settings or pipeline definition  3) Pipeline processing: pipeline config stored in aws-codepipeline.yml or Jenkinsfile in code repo  4) Source stage: pipeline retrieve latest code from repo (git clone + git checkout)  5) Build code: pip install + docker build. 6) Run tests: pytest tests + python evaluate\_model.py  7) Deploy (to SageMaker, EKS, …): aws sagemaker create-model + aws sagemaker create-endpoint-config + aws sagemaker create-endpoint | | | | |
| Retraining Integration | | | 1) Automated retraining pipeline: new data is available, CodePipeline orchestrate pipeline  2) Schedule retraining: retrain at regular intervals. CloudWatch Events can be used to invoke training job  3) Drift detection: SageMaker Model Monitor can be used to detect model drift, then trigger retraining  4) Incremental learning: update model w new data w/o complete retraining from scratch (SageMaker supported algo include XGBoost & Linear Learner)  5) Experimentation and A/B testing: compare performance of diff model versions or config. SageMaker and Amazon Personalize can be used to deploy and manage these experiments | | | | |
| To prevent catastrophic forgetting, several methods:  1) Regularization-based: - Elastic Weight Consolidation (EWC) = assign diff importance to model params based on relevance to previous tasks, preventing model from forgetting impt params  - Synaptic Intelligence = tracks importance of each param during training and use this info to selectively update each param  2) Replay-based: - Experience Replay = stores a small subset of data from previous tasks and replays it during training of current tasks  - Generative Replay = no actual data stored. A generative model synthesizes data from previous tasks and uses it for training current task  3) Architectural: - Progressive NN = growing model’s capacity as new tasks are learned, supporting model to retain knowledge from previous tasks w/o overwriting it  - Modular Networks = model divided into diff modules, each module responsible for a specific task  4) Rehearsal-based: - Gradient Episodic Memory = stores a small subset of data from previous tasks and interleaves it w data for current task during training  - Exemplar Replay = stores a small set of representative examples from previous tasks and use during training of current task | | | | |
| EventBridge rules | | | Events from SageMaker delivered to EventBridge almost in real time. Guidelines for rules:  1) Rule composition: organize EventBridge rules into logical groups based on type of events they process, services they interact w, or specific use case they address  2) Event filtering: use precise event filtering criteria in rules  3) Event replay and replay retention: to analyze and troubleshoot past events  4) Error handling: implement error handling mechanisms in EventBridge rules, such as error notifications, to capture and handle failures or exceptions that might occur during rule processing  5) Monitoring and alerting | | | | |

SageMaker pipeline processing steps: only invoke preprocessing scripts in SageMaker, cannot invoke external Glue jobs

- To invoke external Glue jobs: use Callback steps in SageMaker pipelines to pause the pipeline, trigger an external service (e.g. Glue job) and then resume pipeline when external service is done

Deployment strategy: Shadow Testing = send a copy of live prod traffic to a shadow variant of new model

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ML Solution Monitoring, Maintenance, and Security: 1) Monitor model inference  2) Monitor and optimize infrastructure and costs. 3) Secure AWS resources | | | | | | | |
| Detecting Drift in Monitoring | | | 1) Data Quality Drift: data in prod becomes diff from training data (data drift or incorrect data received)  2) Model Quality Drift: predictions by model differ from ground truth over time (due to data / concept drift)  3) Bias Drift: incr in bias and can be introduced by changes in live data distribution (training data too small, training data inherently have bias)  4) Feature Attribution Drift: aka concept drift (contribution of features to predictions differ) | | | | |
| Data quality monitoring: 1) Implement data validation checks (data type, range, missing values, outliers)  2) Calculate statistical metrics like mean, s.d., missing value counts, outlier counts on input data  3) Compare these metrics w baseline values or expected ranges established during training phase (significant deviations could mean data quality issues or distribution shifts)  4) Use techniques like data drift detection: Kolmogorov-Smirnov tests or Maximum Mean Discrepancy | | | | |
| Model quality monitoring:  1) Calculate relevant evaluation metrics (accuracy, precision, recall, F1-score, AUC-ROC)  2) Implement techniques like confidence thresholding or uncertainty estimation to identify potentially unreliable predictions (e.g. set threshold on model’s confidence score, or use techniques like Monte Carlo Dropout to estimate uncertainty of predictions)  3) Flag predictions w low confidence or high uncertainty for further review  4) Monitor model’s performance on diff subpopulations of data to detect performance degradation or bias | | | | |
| Model bias drift monitoring: 1) Calculate bias metrics (disparate impact, equal opportunity difference, average odds difference) for diff groups.  2) Compare bias metrics w baseline values or acceptable thresholds established during model training  3) Implement techniques like adversarial debiasing or calibrated equalized odds to mitigate bias during training or inference | | | | |
| Feature attribution drift monitoring: 1) Use interpretability techniques like SHAP to calculate feature attributions / importance scores for individual predictions  2) Calculate statistical metrics (mean, s.d.) on feature attributions  3) Compare metrics w baseline values established during model training phase  4) Identify features whose attributions have changed significantly over time (data or model drift) | | | | |
| SageMaker Model Monitoring | | | | Fully managed auto monitoring service for ML models  - Provides visibility into data quality, monitor model performance, alerts  - Use baseline of training dataset -> monitor incoming data w baseline stats (mean, s.d.) -> if issue detected, generate report and save in S3  - Able to monitor for data quality, model quality, model bias drift, feature attribution drift  - Can integrate w CloudWatch, CloudWatch Logs, EventBridge, CloudTrail for monitoring & auditing  - Can integrate w SageMaker Clarify to improve visibility into potential bias  - Can run a job using cron expression, or on demand | | | |
| After a monitoring job starts, can view the job status:  1) Completed status. 2) Complete w violations (constraint violations detected)  3) Failed status (monitoring job failed, likely due to client error, incorrect role permissions or infra issues) | | | |
| Types of output for Monitor jobs:  1) Statistics: output in statistics.json file  - statistical params calculated for baseline data and data being captured  - dataset-level statistics (num of rows or instances). – feature level statistics for each column  - other statistics like num of nulls, max, min, mean  2) Violations file list results of evaluating constraints (specified in constraints.json file) against analyzed dataset. Report is output as constraints\_violations.json  3) Metrics: per-feature metrics output to CloudWatch, where you can set up dashboards & alerts  - Summary metrics from CloudWatch visible in SageMaker Studio and further analyzed in a notebook  4) Logs: info about monitoring jobs like start time, end time, status can be viewed in CloudWatch console  - Requests and predictions asynchronously copied from disk to S3 | | | |
| SageMaker Model Monitoring – data quality | | | | 1) Initiate data capture on endpoint.  - Incoming prediction requests and responses captured (can specify %) and stored in S3.  - *data\_capture\_config = DataCaptureConfig(enable\_capture=True, sampling\_percentage=100)*  *- model.deploy(…, endpoint\_name = “”, data\_capture\_config=data\_capture\_config)*  2) Create baseline: get SageMaker to suggest a set of baseline constraints & generate descriptive stats  - start a baseline processing job w the suggest\_baseline mtd of the ModelQualityMonitor object using SageMaker Python SDK  - Will output “constraints.json” & “statistics.json”. For numeric (min, max), categorical (missing, distinct)  3) Define and schedule data quality monitoring jobs  - Monitoring job will capture new inference data from deployed endpoint & compare against baseline  4) Integrate data quality monitoring w CloudWatch to view data quality metrics  - Can use built-in SageMaker Model Monitor container  - Configure CloudWatch alarms to alert when data quality drifts beyond specified thresholds  - Can use alarms to initiate remediation actions (retraining model, updating training data)  5) Interpret results of monitoring job: output constraint\_violations.json file  - SageMaker Model Monitor prebuilt container checks for data\_type\_check, completeness\_check, baseline\_drift\_check, missing\_column\_check, extra\_column\_check, categorical\_values\_check | | | |
| SageMaker Model Monitor – model quality | | | | 1) Data capture config on endpoint -> Endpoint -> Requests + Predictions  2) Human in the loop -> Inference ground truth  - Can use SageMaker Ground Truth labels (ensure monitoring job only uses data for which labels are available. Can adjust ‘start\_time\_offset’ and ‘end\_time\_offset’ params to select data)  3) Merge job: combine ground truth data w predictions to perform comparison  4) Merge files: intermediate result of model quality drift monitoring execution  - Merged data save to JSONL file, where ea line = Valid JSON obj combining captured & ground truth data  5) Model quality Monitoring Job: report, stats and violations saved to S3  6) CloudWatch Metrics | | | |
| Binary classification metrics: confusion matrix, accuracy, recall, precision, TPR, TNR, FPR, FNR, AUC, PR (Precision-Recall) and ROC (Receiver Operating Characteristic) Curves, F1 score  Multiclass Classification metrics: confusion matrix, weighted precision/recall/accuracy/F1 score  Regression: MAE, MSE, R-squared | | | |
| SageMaker Clarify – Bias Drift | | | | Clarify detects potential bias and explain the predictions that models make (by using a feature attribution approach)  SageMaker Model Monitor integrates w SageMaker Clarify to monitor predictions for bias. Post-training bias metrics in SageMaker Clarify answers:  - Are all facet values represented at a similar rate in positive model predictions  - Does model have similar predictive performance for all facet values | | | |
| 1) Use SageMaker Clarify directly: configure a SageMaker Clarify processing job  - SageMaker Python SDK API provides high level wrapper to set up job (including retrieving SageMaker Clarify container image URI, and generating analysis config file)  2) Use SageMaker Model Monitor: create baseline specific to bias drift, then schedule monitoring job like other monitoring type  - Evaluate baseline constraints against analysis results of current monitoring job  - If violations detected, job puts them in constraint\_violations.json file  - Can also configure alerts in CloudWatch if bias breaches a threshold  3) Use SageMaker Data Wrangler: detect bias during data prep  - Specify input features, and SageMaker Clarify runs an analysis job to detect bias in these features | | | |
| Monitoring for Feature Attribution Drift | | | | Feature attribution = feature importance / feature relevance  SageMaker Model Monitor integrate w SageMaker Clarify to provide explainability tools for deployed models hosted in SageMaker.  1) Activate data capture. 2) Establish baseline (SHAP). 3) Store in S3. 4) Set up monitoring schedule or feature attribution monitoring job (model explainability monitor calculates a normalized global SHAP score). 5) CloudWatch metrics if global SHAP scores fall below a threshold + store results in S3 | | | |
| Monitoring Model Performance using A/B Testing | | | | | A/B testing = split testing or bucket testing: evaluate relative performance of 2 or more models simultaneously using live prod traffic by running multiple models in parallel and exposing them to a controlled subset of prod traffic  SageMaker allows testing of multiple models or versions behind the same endpoint and can specify how to route traffic  - Track CloudWatch metrics for latency, num of invocations, how it was distributed among the variants  - Enable SageMaker Model Monitor to track how model is performing | | |
| SageMaker Model Dashboard | | | | Provide consolidated view of models in SageMaker console.  If SageMaker Model Monitor is set up, can also track models’ performance as they make real-time predictions on live data.  1) Display alerts in CloudWatch. 2) Risk rating: user-specified param w a low, medium or high value of business impact. 3) Endpoint performance: CpuUtilization, MemoryUtilization, DiskUtilization  4) Most recent batch transform job. 5) Model lineage graphs (from data prep to. deployment)  6) Links to model details | | | |
| Automated Remediation | | | | | 1) Stakeholder notifications: Send notifications to relevant stakeholders  - Data scientist if data drift detected to analyze and assess need to retrain model  - Business analyst when monitoring metrics indicate changes that impact business KPI  2) Data analysis: - Data scientist if data drift  - Site reliability engineer if missing data to check data is not being lost  3) Model retraining: CloudWatch alarm setup to initiate model retraining pipeline if threshold breached  4) Autoscaling: scale out/up when infra resource utilization metrics low and scale in for costs | | |
| Model retraining strategies: 1) Event driven: CloudWatch to detect drift by creating alarms  2) On-demand. 3) Scheduled: historical monitoring data identify seasonal patterns and need retraining  - CloudWatch can also initiate an alarm state at a specific data and time to trigger retraining  - Training & deployment pipelines retrain the model, baseline the new model. Once automated testing completed, traffic redirected to new endpoint  - Manual intervention might still be needed due to: missing or unexpected data changes, regulatory or compliance concern (manual review to check data is anonymized), business changes or shifts in problem domain, revaluation of business KPIs or problem framing | | |
| Tools to help troubleshoot issues: 1) Model registry to manage model versions, associate metadata  2) Model cards: store model info to help centralize and standardize model documentations.  - Capture training details like training dataset, evaluation results  - Document business details and intended use  3) Lineage tracking: investigate pipelines and events that created an artifact  - Use SageMaker Lineage Tracking to locate resources that created that artifact | | |
| KPIs for ML Infra | | | | | | | Utilization, Cost, Throughput, Availability, Scalability, Fault tolerance |
| Monitoring | | | CloudWatch: monitors AWS resources in real time. Can collect and track metrics  - CloudWatch home page displays metrics on every AWS service use + resource utilization, app performance and operational health  CloudWatch alarms: alarms to monitor metrics and send notifications when threshold breached  CloudWatch dashboards: customizable home pages  - For multiple AWS accounts, can set up CloudWatch cross-account observability  - Can create dashboards from console or AWS CLI or PutDashboard API operation | | | | |
| SageMaker Monitor: more for monitoring ML infra (w CloudWatch Logs)  - CloudWatch dashboards more for system-wide view of resource utilization, app performance  - Can set continuous monitoring or on-schedule monitoring. – Can set alerts to notify decr in model quality  - Can automate retraining models or detect bias and quality issues  CloudWatch Logs collect log files of monitoring model status and stores the files in S3  - **Patterns** tab on **Logs** **Insights** page find recurring patterns in query results  - **Compare(On)** in time interval selector on **Logs Insights** page to compare query results for selected time range to a previous period.  - **Log anomalies** page in Logs section of navigation pane auto surfaces anomalies in logs | | | | |
| Monitoring & Observability | | | | | Monitoring: collecting, logging and process data on performance and health of model  Observability: can infer internal states of a model based on external outputs  - Observability achieved based on data collected during monitoring phase  - Observability allows knowing system’s behaviour, identify root causes of issues, reason about its overall health (for ML: model interpretability, data lineage, model versioning, distributed tracing [flow of data & requests in distributed system])  - Monitor for model accuracy, inference latency, resource utilization, input data distribution | | |
| Monitoring Tools for Performance and Latency | | | | | 1) X-Ray: provides trace info about responses and calls made by an instrumented app, including: Downstream AWS resources, Microservices, DBs, Web APIs  - Can analyze & debug performance of distributed apps  - Trace requests made to apps that span multiple AWS accts, Regions, AZs  - Works w EC2, ECS, Lambda, Elastic Beanstalk deployed apps written in Java, Node.js, .NET  - Identify performance bottleneck. X-Ray service maps to observe r/s btw services and resources in app in real time. Can detect high latencies and visualize node & edge latency distribution for services  - Traces user requests as they travel through your entire app. Aggregates data generated by app | | |
| 2) CloudWatch Lambda Insights: useful if model uses Lambda functions  - Provides performance monitoring and troubleshooting for Lambda functions | | |
| 3) CloudWatch Logs Insights: explore, analyze and visualize logs instantly  - Interactively analyze and query log data to identify patterns & anomalies  - Can correlate log data from diff sources (model, infra, related services) | | |
| Observability tools | | | | | 1) CloudTrail: logs, monitors & retains API call activities across your AWS acct and store in S3  - Track who initiated, when they occurred, which resources were affected  - Debugging & troubleshooting, & Auditing & compliance  - CloudTrail -> S3 -> Athena OR CloudTrail -> CloudWatch (set up alarms)  - Deliver 1 copy of management events to S3 free by creating a trail. But still have S3 charges  - Create either multi-Region or single Region trails | | |
| Use AWS Console or CLI to create CloudTrail. CLI can only create single-Region trail  Start logging: aws cloudtrail start-logging –name my-trail  View log files in JSON format | | |
| Setting up Dashboards | | | | QuickSight: BI dashboard. Can do anomaly detection, forecasting, autonarratives  EventBridge: receive and respond to events from various AWS services.  - Central hub for event ingestion and delivery, to build event-driven architectures | | | |
| EventBridge can monitor SageMaker events w change in: - model state, - training job state, - hyperparam tuning job state, - transform job state, - endpoint state, - feature group state, - model package state, - pipeline execution state, - pipeline step state, - processing job state, - SageMaker image state, - SageMaker image version state, - Endpoint deployment state, - Model card state | | | |
| Rightsizing Compute Infra | | | | Amazon SageMaker Inference Recommender: analyze ML model and workload patterns -> provide recommendations for best AWS compute resources and configs  - If computationally intensive workflow, might recommend AWS Inferentia chips  - If not as intensive, suggest CPU or GPU instances w specific instance types and configs  - Can also recommend batching, multi-model deployments, auto scaling  - Prospective instances feature: provide list of top 5 instance types optimized for cost, throughput, latency based on preliminary benchmarking  - Default job type are Inference recommendations. Advanced job type are endpoint recommendations (for custom load test) | | | |
| AWS Compute Optimizer: use the rightsizing recommendation preferences feature  - Feature allows adjust headroom & threshold of CPU utilization, headroom of memory utilization, configure specific lookback period, set instance family preferences at organization, acct or regional level  - Use to ensure ML instances are optimally sized for training jobs (EC2, ECS, Lambda) | | | |
| Cost | | Spot Instances: spare EC2 instances offered at significant discounts compared to On-Demand instances. Excellent cost-saving option for fault-tolerant & flexible workloads  Spot Fleets: auto bid on and launch Spot Instances across multiple instance types and AZs  - Can configure Spot Fleet for ML training, specifying target capacity and max bid price | | | | | |
| Cost Optimization tools: AWS Cost Explorer: analyze and visualize AWS costs  - Identify top spend areas and opportunities for cost optimization  AWS Trusted Advisor: provides recommendation for improving performance, cost optimization, security, fault tolerance based on your AWS env | | | | | |
| Service quotas (aka limits) is Region-specific. Can request increase for quota but not all can be increased  - Max supported ML model size per Region = 2GB | | | | | |
| Capacity Blocks for ML: reserve GPU instances on future data to support short-duration ML workloads  - Instances running inside a Capacity Block are placed closed tgt inside EC2 UltraClusters, for low-latency, petabit-scale, nonblocking networking | | | | | |
| Provisioned concurrency  SageMaker Serverless Inference = inference option to deploy and scale ML models w/o caring about infra  - Integrates w Lambda to offer high availability, fault tolerance, auto scaling  1) On-demand serverless inference: workloads w idle periods btw traffic spurts & can tolerate cold start  2) Provisioned Concurrency Serverless Inference to keep a warm pool of compute instances ready to handle incoming inference requests, reducing cold start latency  - Cold start latency: delay due to provisioning new instance and initialize to serve an instance request | | | | | |
| Auto scaling: workload incr, more instances, workload decr, less instance  1) Target tracking. 2) Step scaling. 3) Schedule scaling  - For on-demand serverless inference, SageMaker handle autoscaling, for Provisioned concurrency, can set target metric or schedule for scaling | | | | | |
| Cost Analysis tools | 1) AWS Cost Explorer: - see patterns in AWS spending over time. – Project future cost. – Identify areas that need further inquiry. – Observe Reserved Instance utilization. – Observe Reserved Instance coverage. – Receive Reserved Instance recommendations  2) AWS Trusted Advisor: get real-time identification of potential areas for optimization  3) AWS Budgets: set custom budgets that invoke alerts when cost or usage exceed (or forecasted to exceed) a budgeted amount. Budgets can be set based on tags, resource types, or accts  4) CloudWatch: collect & track metrics, monitor log files, set alarms, auto react to changes in resources  5) CloudTrail: log, monitor and retain acct activity related to actions across AWS infra at low cost  6) S3 analytics: analysis & viz of S3 storage patterns to decide when to shift data to diff storage class  4) AWS Cost and Usage Report: granular raw data files detailing hourly AWS usage across accounts used for DIY analysis. | | | | | | |
| For ML workloads specifically:  1) AWS Billing & Cost Management: centralized platform to view AWS costs across all services & regions.  - Offers cost allocation tags, consolidated billing, detailed billing reports  - By assigning cost allocation tags to ML resources (EC2, S3), can track and attribute costs to specific projects, teams or experiments  - Also generate reports to identify cost optimization opportunities (rightsizing instance, Spot instances)  2) AWS Budgets: set granular budgets for ML projects  - Can alert when costs deviate significantly from historical patterns or exceed threshold to identify spikes or anomalies in ML workloads  3) AWS Cost Explorer: predict future costs for ML workloads based on historical usage patterns  - Cost optimization analysis (instance types, storage configs)  4) AWS Trusted Advisor: instance rightsizing, identify idle resource, reserved instance recommendations | | | | | | |
| 1) Cost allocation tagging. 2) Tagging SageMaker Studio domains and users  3) Tagging SageMaker notebook instances. 4) Tagging SageMaker managed jobs & resources  - Can visualize tags in Cost Explorer, Cost & Usage Report, AWS Budgets | | | | | | |
| Purchasing options to optimize infra costs | | | | 1) Spot Instances: for workloads that can tolerate interruptions, cheaper than On-Demand  2) On-demand instances: short term, unpredictable workloads that cannot be interrupted  3) Reserved Instances: commitment to use instances for specific period (1 or 3 years). For predictable workloads. Cheaper than Spot instances  4) Capacity Blocks: reserve capacity for EC2 instances in AWS Outposts or AWS Wavelength Zones. Reserve capacity in advance and reduce risk of capacity constraints during peak demand period  5) Savings Plans for SageMaker: commit to specific amount of compute usage (in $ per hour) for 1 or 3 years. Cheaper than On-Demand  6) AWS Cost and Usage Report | | | |
| ML Savings Plans for SageMaker: - Long-term ML projects. – Large-scale training workloads. – Batch inference workloads. – Hosted model deployment  1) Compute Savings Plan: cover compute costs associated w SageMaker training jobs, hosted models, batch transform jobs  2) ML Services Savings Plan: cover costs of using AWS ML services (Rekognition, Translate, Comprehend) | | | |
| Shared Responsibility Model | | | | | | Identity & Access Management: Customer + AWS  Customer: Data, Platform, Apps, IAM, OS, Network, Firewall, Client-side data encryption & data integrity authentication, Server-side encryption (file system, data), Networking traffic protection  AWS: Software (Compute, Storage, DB, Networking), Hardware/Infra (Regions, AZs, Edge locations) | |
| IAM | 1) Users: individual identities w specific credential and permissions, created by account root user or admin  2) Groups: collection of IAM users. Grouping simplifies permission management  3) Roles: grant fine-grained permissions to entities (users, apps, services) that need to interact w resources  - Roles eliminate need to share long-term access keys, so more secure than traditional access keys | | | | | | |
| 1) User roles: specific permissions and responsibilities assigned to diff users based on their roles  - Data scientists: access to SageMaker Studio, S3, Athena. - DE: S3, Glue, EMR, Athena.  - MLOps engineer: SageMaker, CodePipeline, CodeBuild, CloudFormation, ECR, Lambda, Step Functions  2) Services roles: provide cloud services w necessary permissions to perform tasks on behalf of user  - SageMaker Notebook: SageMaker Execution Service Role  - SageMaker Processing/Training Job: SageMaker Processing Job/Training Service Role  - SageMaker Model: SageMaker Model Service Role  3) IAM Policies: JSON documents that define permissions for AWS resources  3a) Identity-based policies: attach policies to IAM users, groups or roles to define what actions they can perform on which resources  3b) Resource-based policies: S3 bucket policies or VPC endpoint permissions for an identity to access and interact w AWS resources | | | | | | |
| Principle of Least Privilege | | SageMaker Role Manager: build & manage persona-based IAM roles for DS or MLE  - define & manage granular permissions for SageMaker notebook instances | | | | | |
| Account Separation: use separate AWS accounts for diff env (dev, testing, prod) and diff teams or projects  - Implement access controls btw accounts using Organizations and Service Control Policies (SCP)  - SCP: define rules that all IAM entities (users, roles, groups) in the acct must adhere to  AWS Organizations: centrally manage & apply policies across multiple AWS accts  - Create an Organizational Unit (OU) specifically for ML workloads and accts  - Apply SCPs at the OU level to restrict ML service access & resource creation based on requirements  - SCPs enable allowing/denying specific ML services, resource types, Regions | | | | | |
| Network Access Controls | | | | VPC: isolate ML resources. Can create VPC w no internet access  - Create VPC w private subnets w/o internet access | | | |
| VPC endpoints: access certain AWS services from VPC  - For supported AWS services, can privately connect VPC w/o internet gateway, NAT device or VPN connection. Can further limit access to these services by using endpoint policies  - SageMaker Studio runs in a service VPC by default. When notebooks in studio env access AWS resources, VPC routes this traffic over internet by default. To route traffic over AWS network, launch SageMaker Studio & SageMaker notebook instances in a VPC of your choice  - When launch SageMaker Studio instances in a VPC of your choice, they communicate w the domain through the Elastic Network Interface (ENI) in a private subnet. ENI is protected by a Security Group (SG). Can then create interface VPC endpoints for AWS services & resources that SageMaker Studio will access through their own ENI in the same subnet  - VPC endpoints: can connect to SageMaker runtime, SageMaker API, CloudWatch Logs, S3 | | | |
| Endpoint Policies: resource-based policy that you attach to a VPC endpoint  - When deploy model to a SageMaker endpoint, can optionally configure an endpoint policy | | | |
| NAT Gateways for Egress Traffic: enable instances in private subnet to connect to internet or other AWS services but prevent internet from initiating connections w those instances  - NAT Gateway place in public subnet to allow outbound internet access for private ML instances (download libraries or docker image from public repos) | | | |
| Security Groups (SG): virtual firewalls that control inbound and outbound traffic to and from your ML resources within a VPC  - configure SG rules to allow traffic based on protocols, ports and src or dest IP addr | | | |
| Network Access Control List (NACL): additional layer of security that act as a firewall for controlling traffic in and out of subnets in your VPC  - NACL rules allow or deny traffic based on IP addrs, protocols and ports | | | |
| AWS Network Firewalls: managed service to deploy network protection across VPC  - Provides intrusion prevention & detection, web filtering and domain or content inspection features  - Private subnet -> NAT gateway -> AWS Network Firewall (public subnet) -> Internet Gateway -> Internet | | | |
| Security & Compliance Features | | | | | Data Privacy and Opt-Out: can opt out of metadata collection in SageMaker  Data Encryption: - Encryption at Rest (AWS KMS) for SSE, S3 Managed Keys or bring your own key  - Encryption in Transit: Data transmitted btw SageMaker and other AWS services is encrypted using TLS protocols  - IAM: granular control over who can access resources and perform specific actions  - Logging & Monitoring: CloudWatch, CloudTrail | | |
| Compliance & Governance Features | | | | | 1) AWS Artifact: self-service portal providing on-demand access to AWS compliance reports and select online agreements.  2) AWS Config: service to assess, audit and evaluate configs of you AWS resources  - Continuously monitors and records resource configs  3) Security Hub: comprehensive view of security alerts & security posture across AWS accounts  4) Audit Manager: continuously audit AWS usage to streamline how you assess risk and compliance w regulations & industry standards  5) Amazon Inspector: automated vulnerability management service that scans AWS works for software vulnerabilities and unintended network exposure (scan container images in ECR)  6) AWS Service Catalog: enterprise can create & manage catalogs of pre-approved SageMaker resources, configs, ML models | | |
| SageMaker governance tools:  1) SageMaker Role Manager: admin define min permissions needed for common ML activities  2) SageMaker Model Cards: document, retrieve, share essential model info (intended uses, risk ratings, training details from conception to deployment)  3) SageMaker Model Dashboard: pre-built visual overview of all models in acct. Keeps you informed on model behaviour in prod in 1 view  4) SageMaker Assets: New workflow to streamline ML governance helping users publish, share & subscribe to ML assets and data assets  5) Model Governance & Explainability: ensure data & workloads are protected and compliant w industry standards & regulations | | |
| Compliance certifications & regulatory Frameworks:  1) ISO 27001: Information Security Management System. 2) SOC 2: Service Organization Control  3) PCI-DSS: Payment Card Industry Data Security Standard  4) HIPAA: Health Insurance Portability & Accountability Act  5) FedRAMP: Federal Risk & Authorization Management Program | | |
| Security in CI/CD | | | | 1) Pre-commit: pre-commit hooks, IDE-level checks (linting, formatting, securing code)  2) Commit: Static App Security Testing (SAST) / static code analysis, involve tools that detect bugs by analyzing source code. Work backward by dissecting vulnerabilities to define potential attack methodologies & generate signatures as preventative measures  3) Build: Software Composition Analysis (SCA), identify open source packages used in code, vulnerabilities present, check licenses for each package, examine dependencies  - Also scan infra as code (IaC) manifest files for vulnerabilities in containerized env  4) Test: a) Dynamic App Security Testing (DAST) / black-box solutions = test app during operational lifecycle. Provides recommendations for potential vulnerabilities and compliance issues  - As DAST monitor app behaviour, tend to generate fewer false positives than SAST tools  - Can automate these security tests using Lambda or CodeBuild  b) Interactive App Security Testing (IAST) = combine advantages of SAST and DAST tools  - run dynamically to identify issues like a DAST tool while also running inside app server to evaluate code like SAST tool. IAST tools provide real-time findings and are useful for API testing  5) Deploy: penetration testing = ensure no vulnerabilities or non-compliant assets go unnoticed in end product. Both tool-based & human-based pen testing are used to find app vulnerabilities  - Multi-directional approach, navigate app to detect diff app behaviours and user experiences  6) Monitor: penetration testing + red/blue/purple teaming + event monitoring  b) Red/Blue/Purple teaming = security experts w diff roles simulating real-life cyber attacks  - To pinpoint system deficits, improve protection mechanisms and processes, maximize infra efficiency while minimizing risk  - Red teams = attackers, Blue = defenders, Purple = members from both red & blue teams to provide a multi-faceted approach & various security perspectives | | | |
| Security through Monitoring | | | | 1) CloudWatch: - CloudWatch Metrics. – CloudWatch Alarms. – CloudWatch Logs  2) CloudTrail: record of actions taken by users, roles, AWS services in SageMaker | | | |
| To troubleshoot & debug security issues:  1) CloudTrail logs: identify potential unauthorized API calls to SageMaker resources  2) Data event Logs: check if any unauthorized entities accessed model’s input or output data during training or inference. Also provide visibility into data processed by models  3) IAM policies: proper permissions granted for SageMaker resources & operations  4) VPC flow logs: monitor network traffic to and from SageMaker resources  5) Encryption settings: check data encryption enabled for SageMaker resources, both at rest & in-transit  - Check AWS KMS key configs to ensure keys are properly managed & rotated | | | |
| Use AWS PrivateLink to privately connect VPC to SageMaker | | | |

VPC peering connection: connect 2 VPCs within the same Region w/o passing through internet

AWS Direct Connect: private connection btw on-prem and AWS Cloud

By default, SageMaker training and deployed inference containers are internet-enabled

- To block internet access, enable network isolation mode