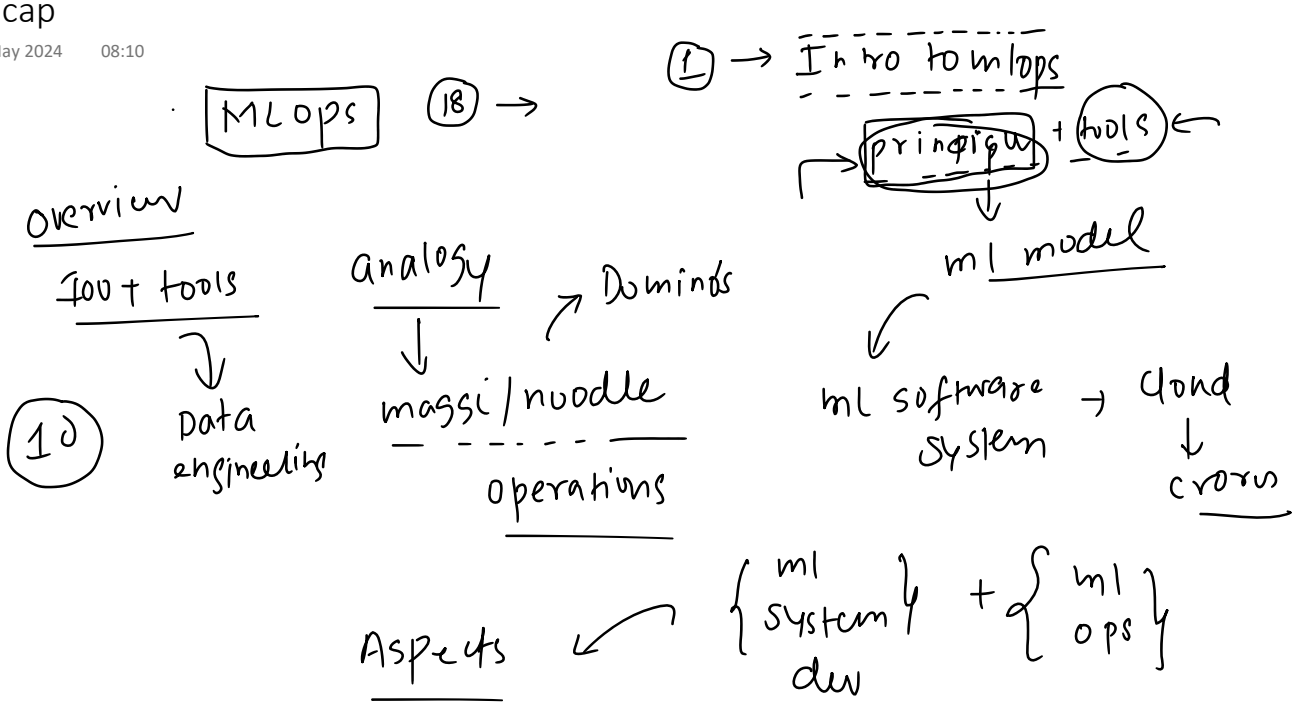
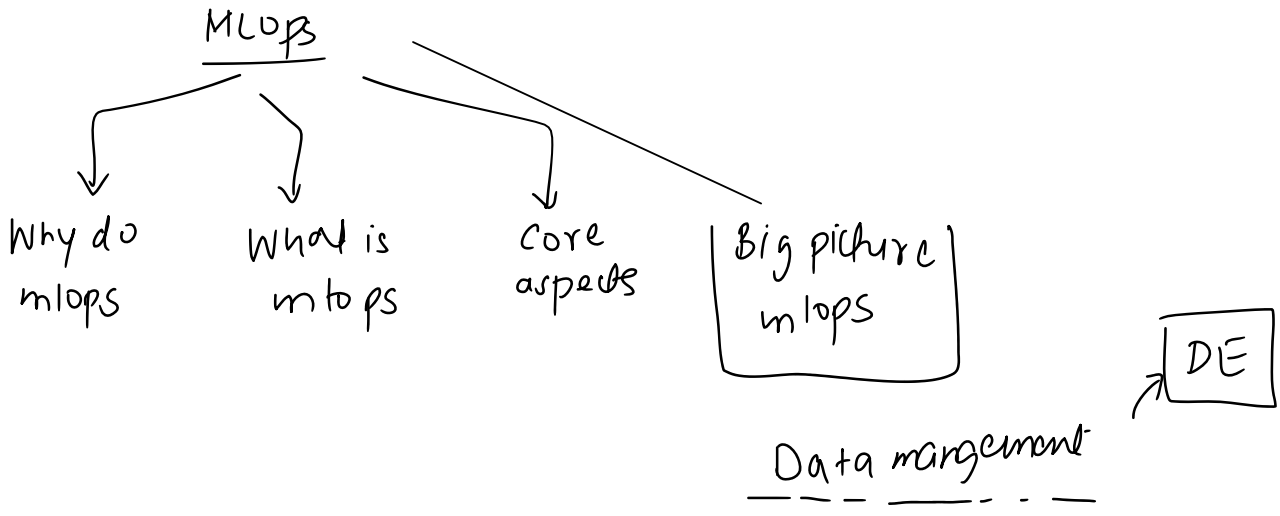
Recap

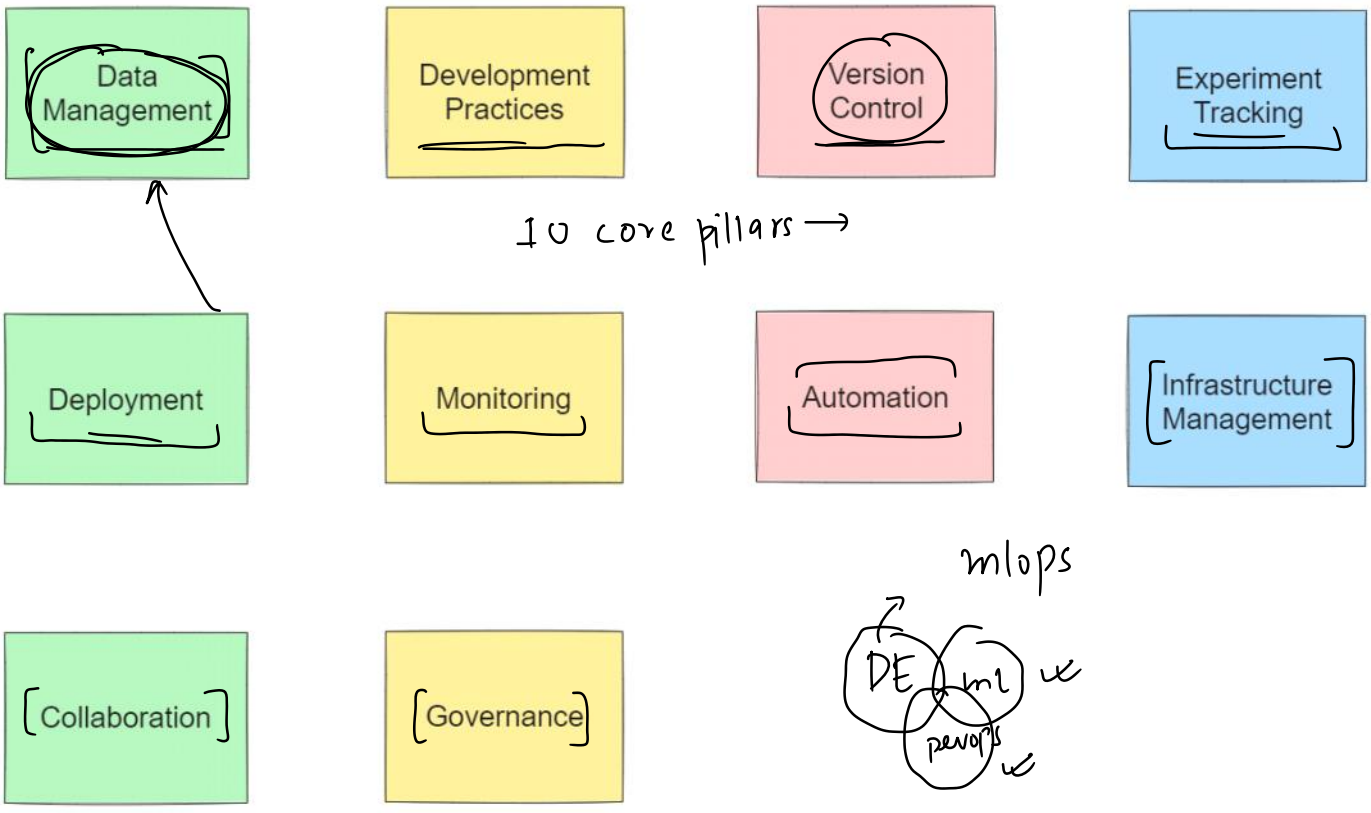
13 May 2024 08:10





Core Aspects [Revision]

13 May 2024 08:11



Benefits of MLOps

13 May 2024 16:06

1. Scalability
2. Improved performance
3. Reproducibility
4. Collaboration and efficiency
5. Risk reduction
6. Cost Savings
7. Faster time to market
8. Better compliance and governance

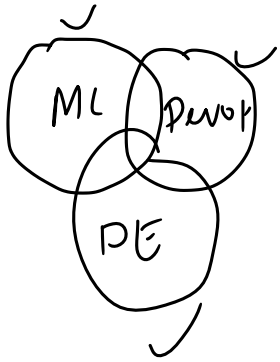
Swiggy

1. Data Management
2. Development Practices
3. Version Control
4. Experiment Tracking/Model Registry
5. Model Serving and CI/CD
6. Automation
7. Monitoring and Retraining
8. Infrastructure Management
9. Collaboration and Operations
10. Governance and Ethics

Challenges

13 May 2024 16:06

1. Complexity of ml models [variability, black box nature]
2. Multitude of models
3. Quality of data
4. Cost and resource constraints
5. Handling scale
6. Security risks
7. Compliance and regulatory concerns
8. Integration with existing systems
9. Limited Expertise/Skill gap



Prerequisites

13 May 2024 16:07

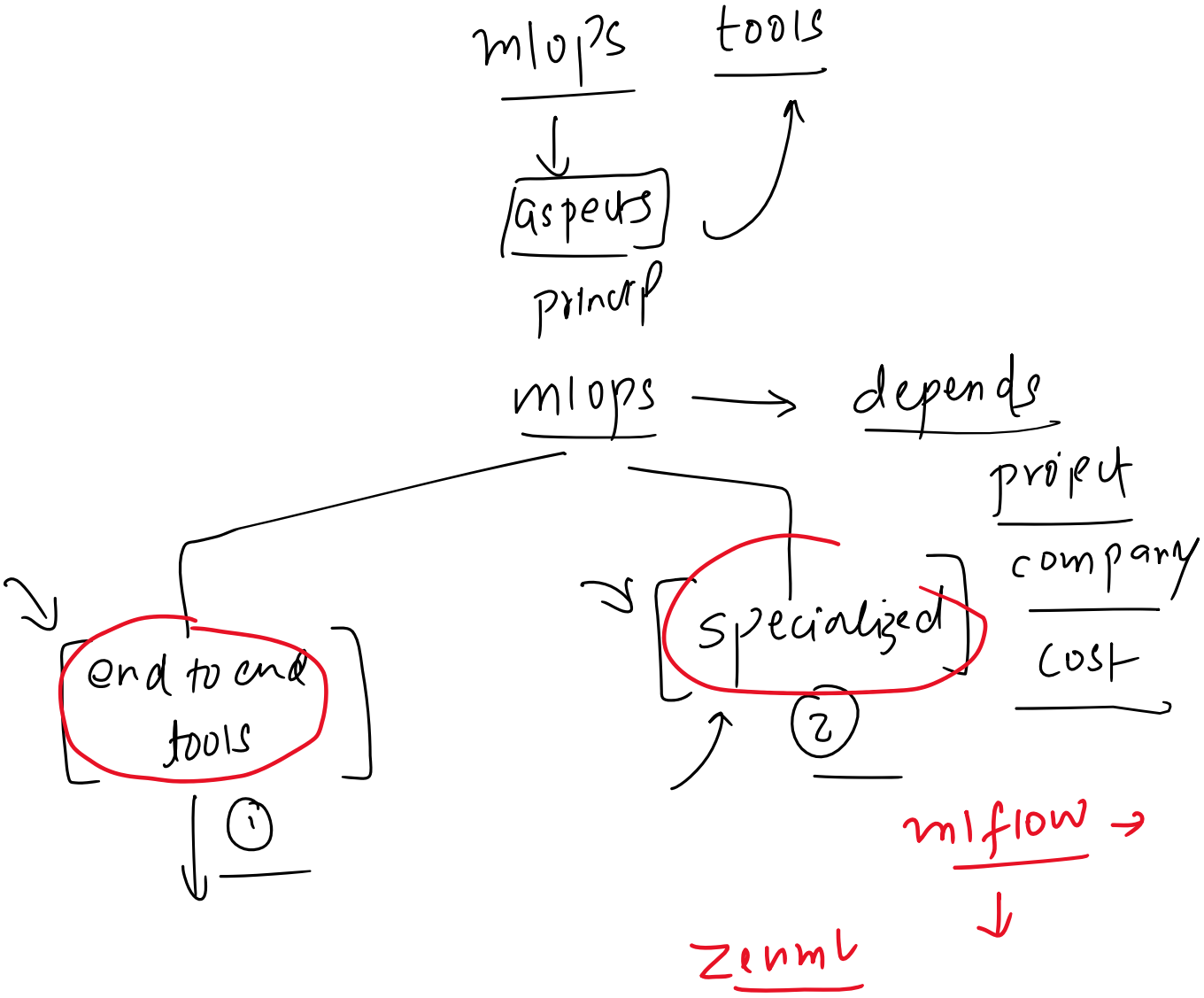
1. Basic understanding of ML
    a. Cleaning and preprocessing
    b. Feature engineering
    c. Model building
2. Software development skills
    a. Python
    b. Git
    c. Software development best practices [OOP, Design Patterns]
    d. Linux
    e. Venv
    f. Yaml(infrastructure as a code)
3. Data Engineering
    a. SQL
    b. Big Data Tech [Spark, Kafka]
    c. Data Storage Solutions [Databases, Data Warehouses, Data lakes]
4. DevOps Principles and Tools
    a. CI/CD Pipeline
    b. Automation
5. Familiarity with cloud platforms
    a. AWS, GCP and Azure
6. Containerization technologies
    a. Docker
    b. Kubernetes
7. Networking Principles
    a. Distributed computing
8. Security Fundamentals
    a. Cybersecurity fundamentals
9. Soft Skills

MLOps Tools Stack

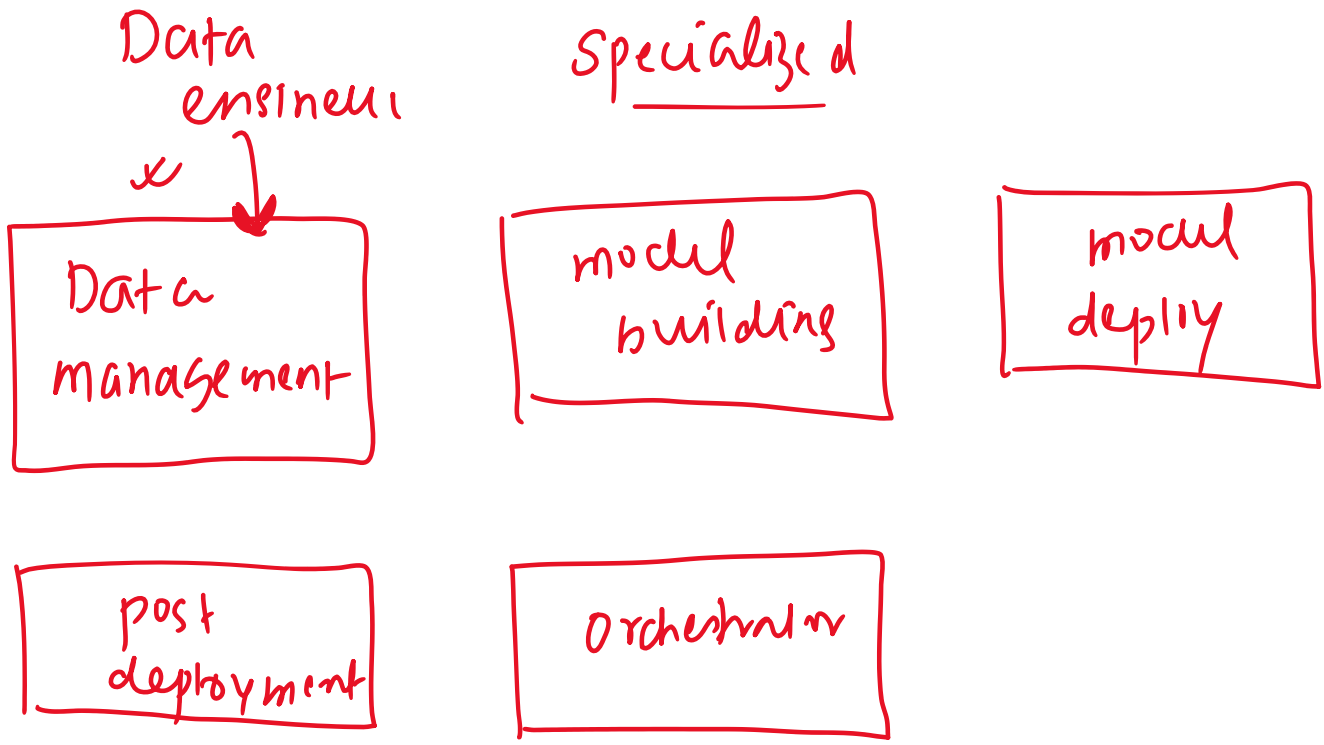
14 May 2024 09:23

full stack web dev
↳ backend + frontend

An MLOps tool stack refers to the set of tools and technologies used together to facilitate the practice of Machine Learning Operations (MLOps). This involves managing the lifecycle of machine learning models from development through deployment and maintenance, incorporating principles from DevOps in the machine learning context. The goal of an MLOps tool stack is to streamline the process of turning data into actionable insights and models into reliable, scalable, and maintainable production systems.

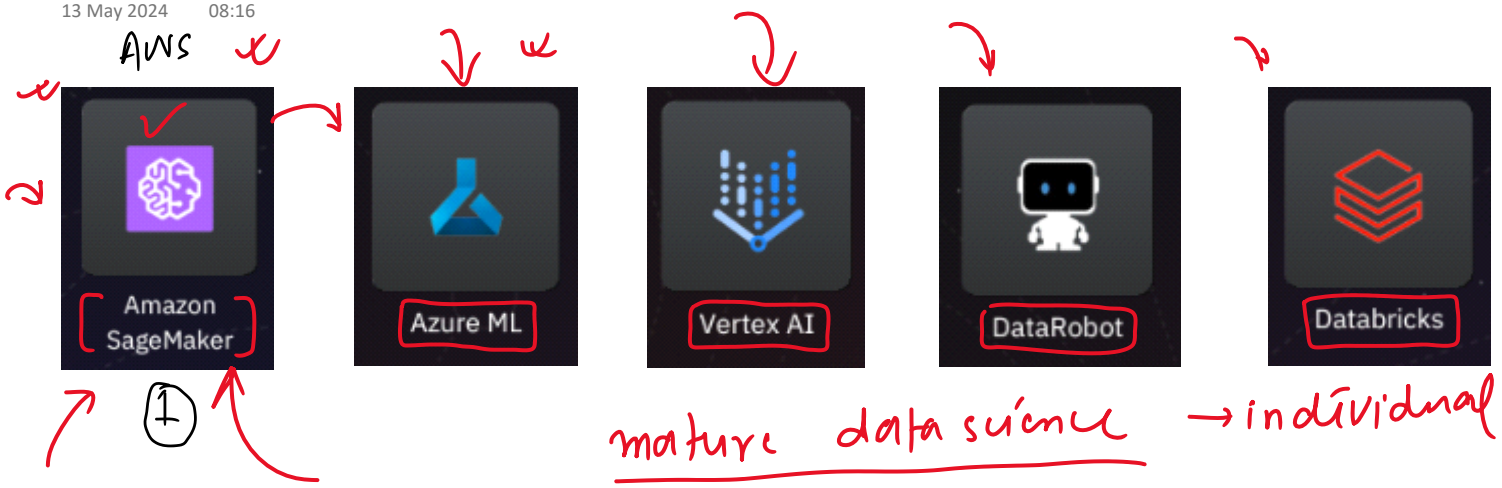


Data Specialized


Data engineer
Specialized
Data management
model building
model deploy
post deployment
Orchestrator

1. End to End MLOps Platforms

13 May 2024 08:16



Advantages

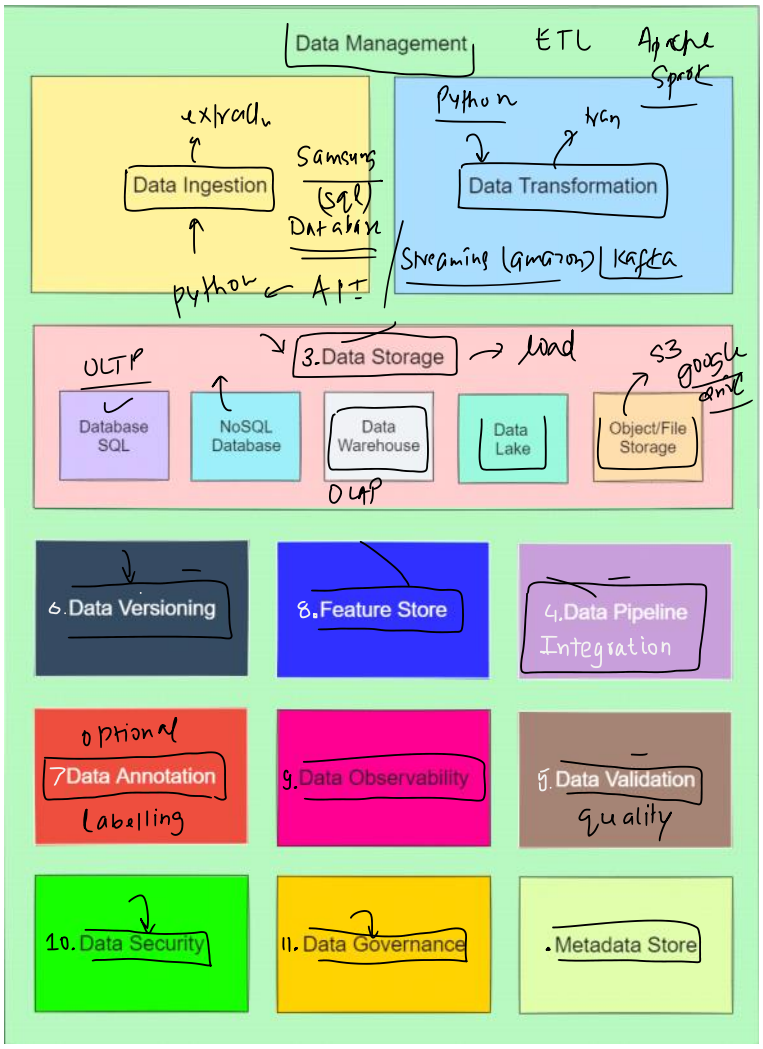
- Easy to setup and use
- Standardization and consistency across projects
- Good for quick experimentation
- Reduced IT overheads
- Enhanced security
- Better support

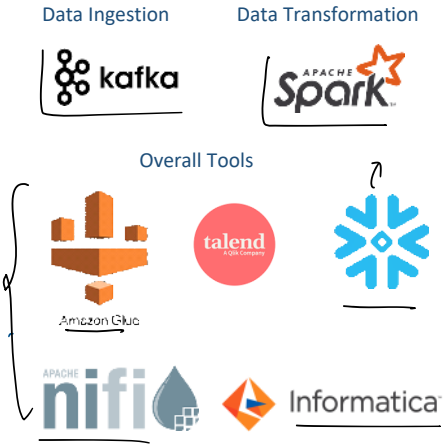
Disadvantages

- Cost
- Vendor lock-in
- Limited options to customize
- Privacy concerns

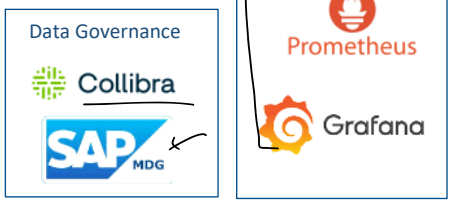
2. Data Management Tools

13 May 2024 18:23





spandas
h ge
is data
distribute



- **Task 1**: Trigger data extraction jobs from the sales and inventory systems.
- **Task 2**: Run data cleaning and transformation scripts.
- **Task 3**: Load the cleaned and transformed data into the data warehouse.
- **Task 4**: Execute SQL queries in the data warehouse to generate analytical reports.
- **Task 5**: Distribute reports via email to the inventory management and marketing teams.

1. Extraction Phase (ingestion)

- **Pre-Extraction Validation**: Before extracting data from the source systems, initial validation checks can be performed to ensure that the source systems are accessible, and that the data is available and in an expected format. This might include checking file sizes, timestamps, or the presence of specific files or tables.

2. Transformation Phase

- **During Transformation**: This is where the bulk of data validation occurs. As data is transformed, various checks are implemented to ensure data quality:
  - **Data Type Checks**: Ensuring that data types are consistent with expected formats (e.g., dates, numeric values, strings).
  - **Range Checks**: Validating that data falls within acceptable ranges (e.g., age, prices, quantities).
  - **Consistency Checks**: Ensuring data across different fields or records is consistent (e.g., a state field matches the corresponding ZIP code).
  - **Completeness Checks**: Verifying that essential data fields are not missing values.
  - **Uniqueness Checks**: Ensuring that data meant to be unique (like transaction IDs or user IDs) does not have duplicates.
  - **Business Rule Validation**: Applying domain-specific rules to ensure the data adheres to business logic (e.g., a customer's credit limit must not be exceeded).

3. Loading Phase

- **Pre-Loading Validation**: Before loading the transformed data into the target system (like a data warehouse or database), final validations ensure the data is ready for integration. This might include:
  - **Referential Integrity Checks**: Ensuring foreign keys correctly link to primary keys in related tables.
  - **Data Formatting Checks**: Confirming that data formatting aligns with the schema of the destination system.
  - **Aggregation Checks**: Verifying sums, averages, counts, and other aggregates are calculated correctly and match expected values.

Data Observability

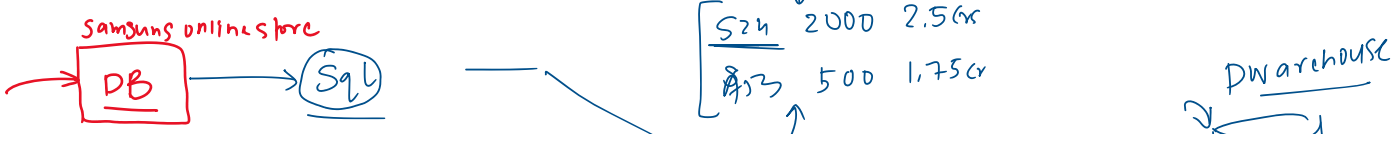
1. Data drift
2. Pipeline failure
3. Monitoring resource usage
4. Throughput
5. Latency

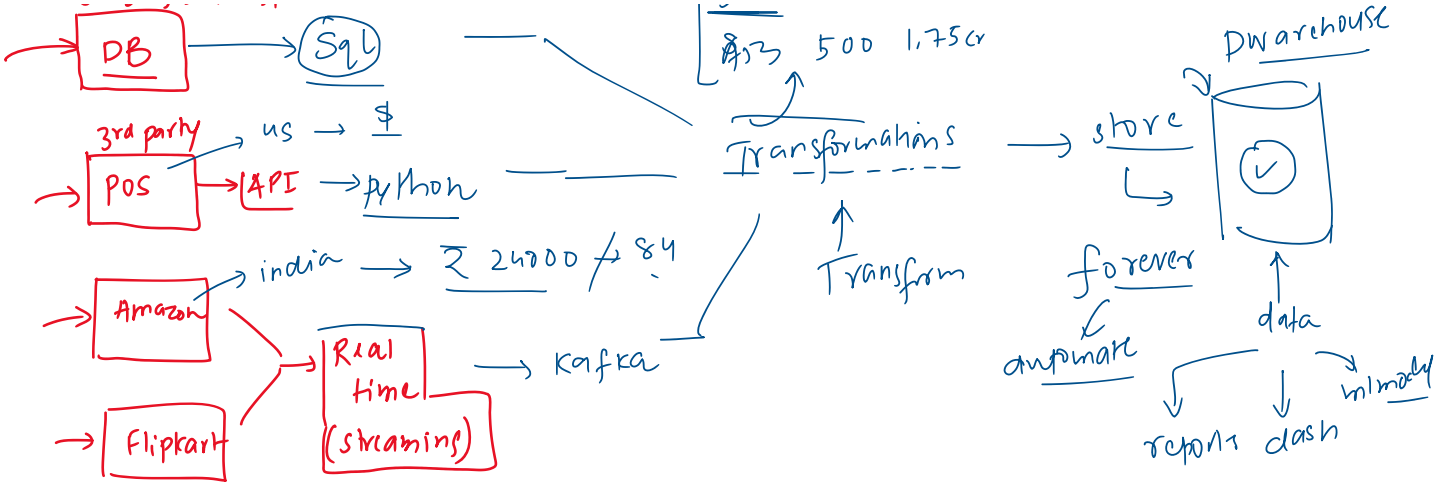
Data Security

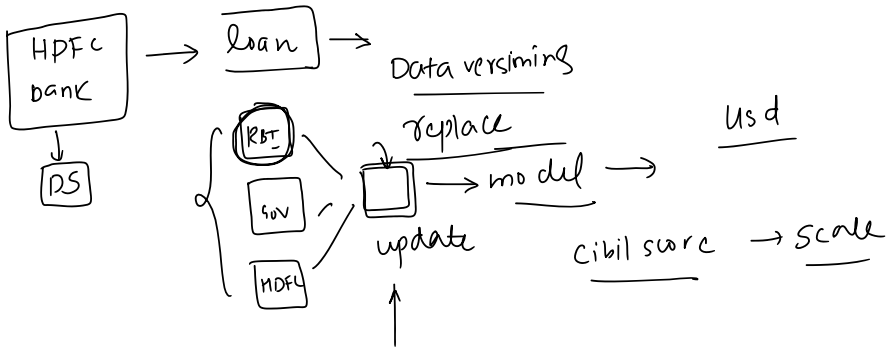
1. Data Encryption at rest, in transit and in use
2. Role based access control
3. Data masking and anonymization
4. Audit Trails - Keeping detailed logs of all data access and changes to track how data is used and by whom.
5. Backup and Disaster recovery

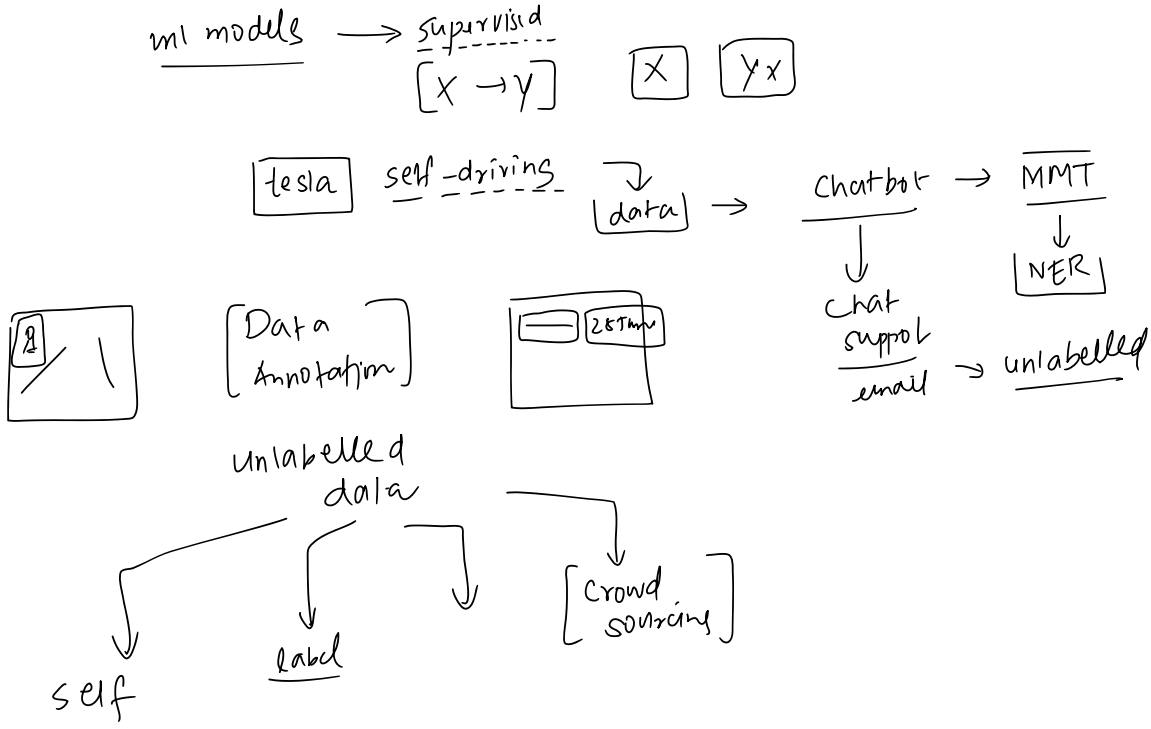
Data Governance

- Data Cataloguing: Helps organizations organize and access their data assets.
- Policy Management: Allows the creation and enforcement of data governance policies.
- Automated Data Lineage: Tracks the origin, movement, and transformation of data.
- Privacy and Security: Helps organizations comply with data protection regulations like GDPR.




DB
Sql
3rd party
POS
API
python
us → $
Amazon
Flipkart
india → ₹ 24000 / 84
Real time (streaming)
kafka
500 1.75cr
Transformations
Transform
store
Dwarehouse
forever
archive
data
reports
dash
inmodel


HPFC bank
loan
DS
RBI
GOV
HDFC
Data versioning
replace
update
model
Usd
Cibil score
scale


ml models
supervised
[X → Y]
X
Yx
tesla
self-driving
data
chatbot
MMT
NER
chat support email
unlabelled
Data Annotation
Unlabelled data
Crowd sourcing
label
self

Feature Store

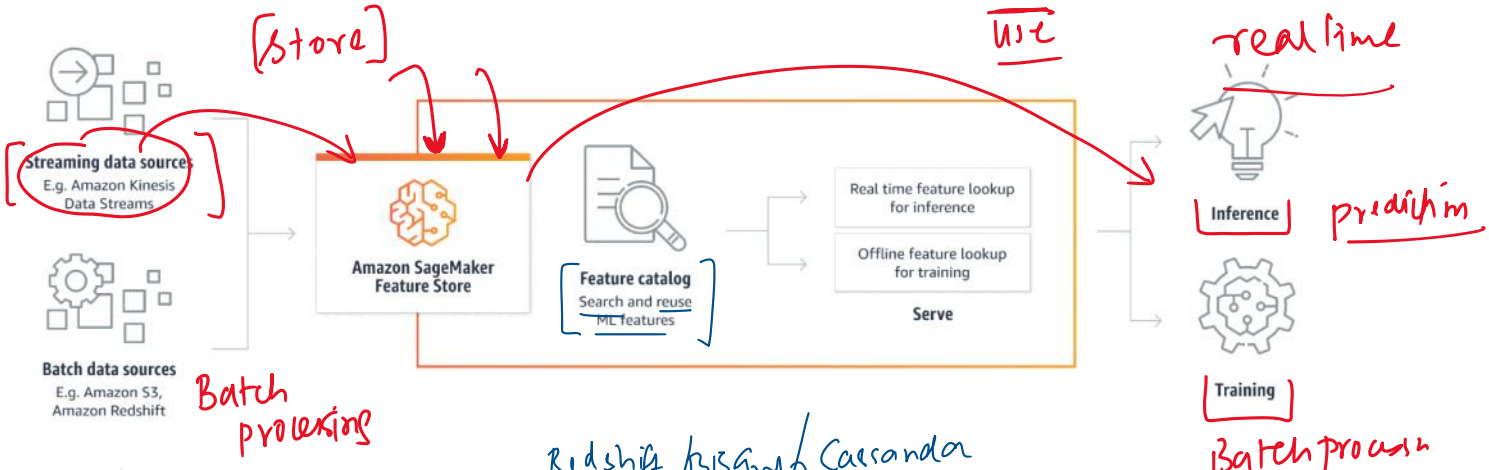
19 May 2024 14:12

A feature store in the context of Machine Learning Operations (MLOps) is a centralized repository for managing, storing, and accessing features (i.e., individual measurable properties or characteristics used in the construction of machine learning models) across various machine learning projects and teams. It plays a critical role in bridging the gap between data engineering and machine learning by ensuring that features used for training models are consistent with those used for making predictions in production environments.

Why use Feature Store?

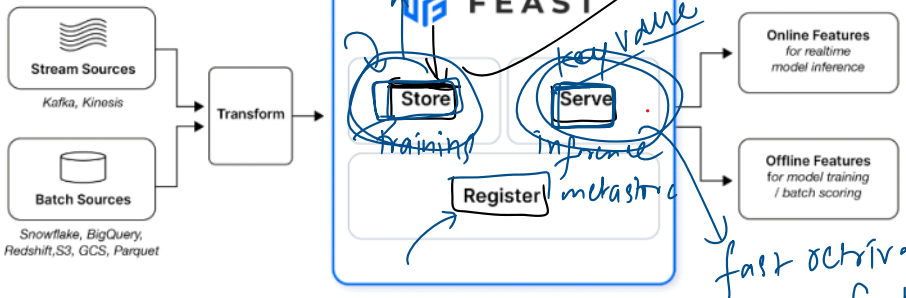
1. Reusability
2. Transfer of Expertise
3. Data Quality
4. Standardization

General Idea



Examples - Feast, AWS Feature Store

Architecture



Key Functions of a Feature Store

1. **Feature Management**: Manages the lifecycle of features, including their creation, versioning, and retirement. This ensures that features are kept up-to-date and remain consistent across different models and applications.
2. **Feature Sharing and Reuse**: Facilitates the sharing and reuse of features among multiple data science teams and projects, reducing duplication of effort and ensuring that improvements to features are propagated across all models that use them.

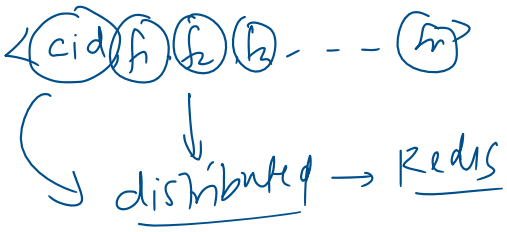
science teams and projects, reducing duplication of effort and ensuring that improvements to features are propagated across all models that use them.

3. **Feature Consistency**: Ensures consistency between training and inference stages. Features used to train models are exactly the same as those used during model predictions, which helps in maintaining model accuracy and performance in production.

4. **Feature Serving**: Provides low-latency access to feature data for real-time model predictions. This can involve serving features directly to online applications or through APIs that applications can query to get the necessary features for making predictions.

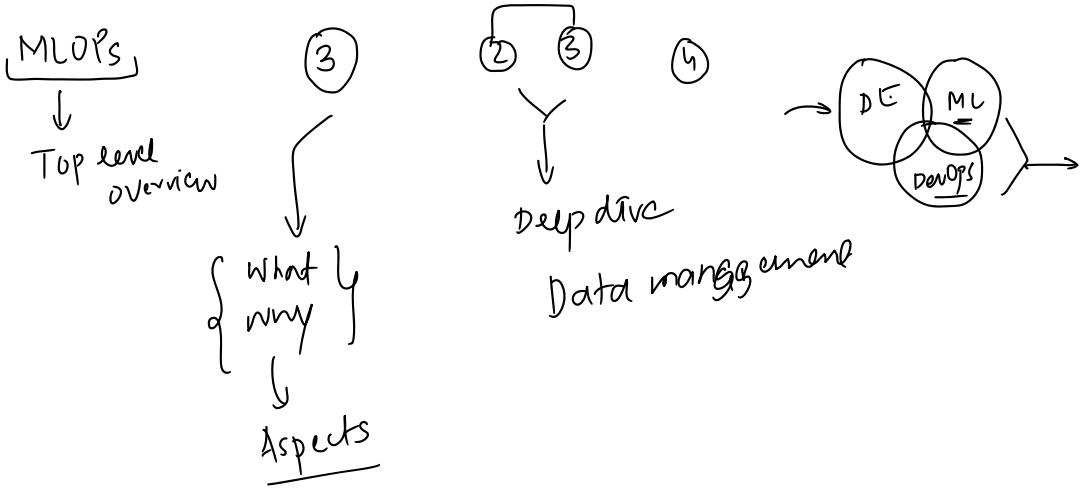
5. **Feature Engineering**: Some feature stores also provide tools to assist in the process of feature engineering, such as transformation functions and pipelines that can be applied to raw data to generate features automatically.

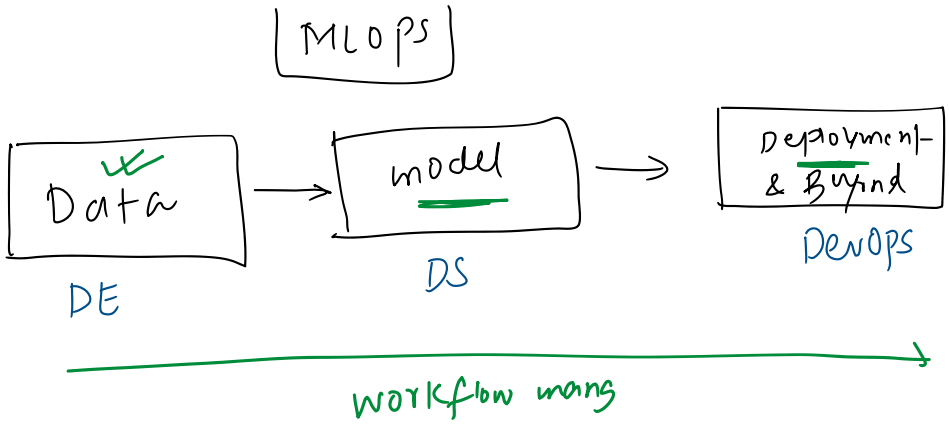
6. **Monitoring and Governance**: Monitors the usage and performance of features and provides tools for governance, such as tracking data lineage, managing metadata, and ensuring that data privacy regulations are followed.



Recap

21 May 2024 14:58

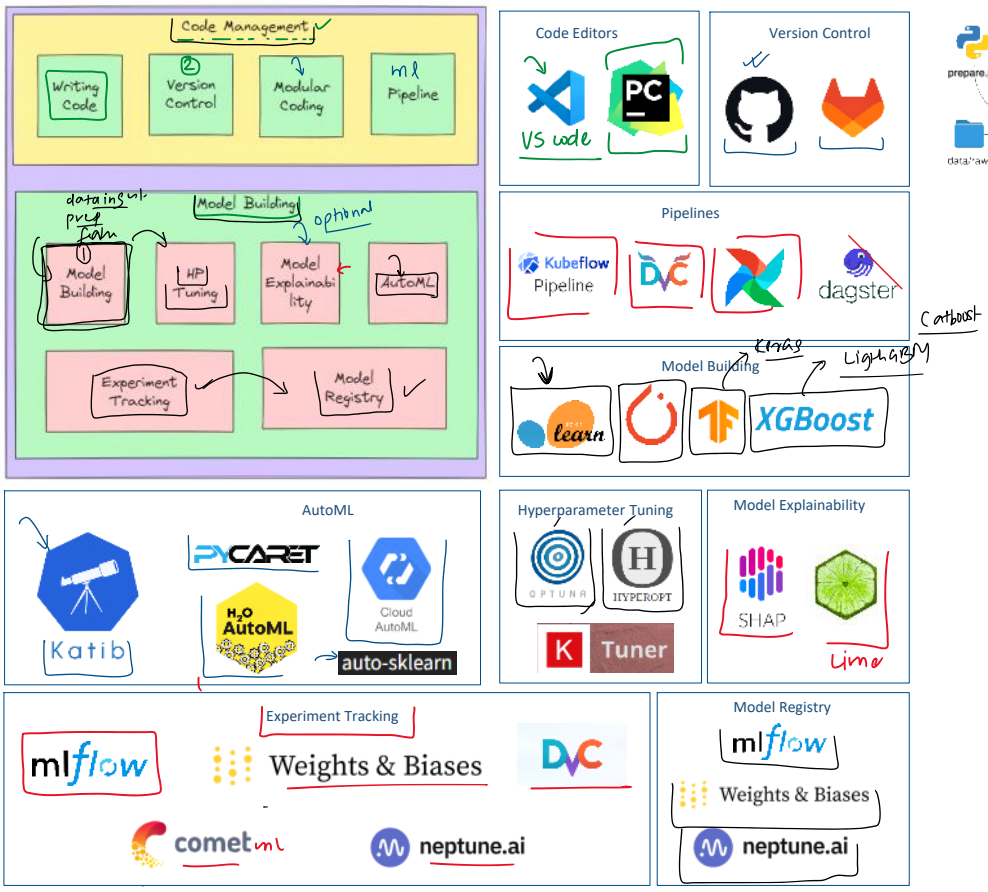


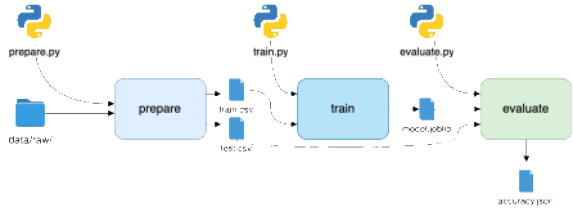


3. Model Building

13 May 2024 22:59

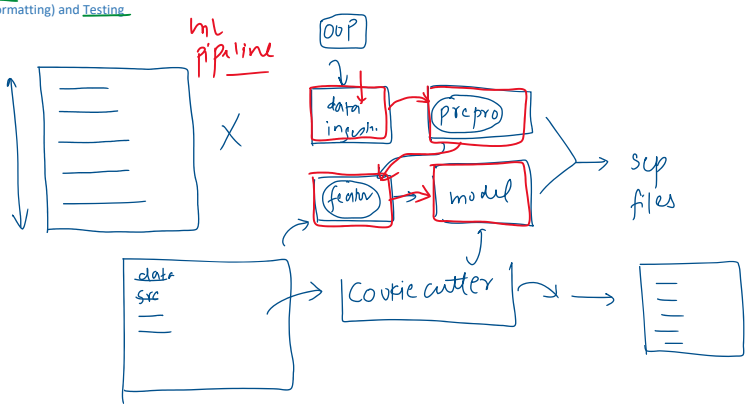
Code management- model building





Reasons to choose VSCode

1. Code Debugging ✓
2. Extensions -> extension for powerful mlops tools like DVC, Docker, Kubernetes etc.
3. Support for Jupyter notebook
4. Git Integration
5. Integrated command prompt
6. Code Quality(linting and formatting) and Testing



When Model Explainability is Essential:

1. **High-Stakes Decisions:**
   - In areas such as healthcare, finance, legal, and autonomous systems, where decisions can significantly impact lives, explainability is crucial for ensuring trust and safety.
2. **Regulatory Compliance:**
   - Many industries have regulations that require transparency and explainability in automated decision-making systems (e.g., GDPR, HIPAA).
3. **Ethics and Fairness:**
   - To ensure that models do not perpetuate or exacerbate biases, explainability helps identify and mitigate unfair biases in the model.
4. **User Trust:**
   - For products or services where user trust is important, providing explainable models can increase user confidence and satisfaction.
5. **Model Debugging:**

5. **Model Debugging:**
   - Explainability aids in understanding why a model is making certain predictions, which is essential for debugging and improving model performance.
6. **Stakeholder Communication:**
   - When presenting models to non-technical stakeholders (e.g., business executives, customers), explainability helps bridge the gap between technical and non-technical understanding.

When to use AutoML

1. Limited expertise
2. Simpler ML problems
3. Rapid Prototyping
4. Baseline model creation

When not to use AutoML

1. Complex ML problems
2. Data quality issues
3. Projects that need complex feature engineering
4. Scalability issues
5. Interpretability is important
6. Suboptimal hyperparameter tuning

Experiment tracking in MLOps refers to the process of systematically recording, organizing, and managing all aspects of machine learning experiments. This includes logging details about the datasets, preprocessing steps, model configurations, hyperparameters, performance metrics, and other relevant artifacts. The goal is to ensure reproducibility, facilitate comparison, and provide transparency throughout the machine learning development lifecycle.

Why Experiment Tracking

1. Comparison
2. Reproducibility
3. Collaboration

What exactly is tracked?

1. Parameter (learning_rate, batch_size, regularization parameters)
2. Metrics (accuracy, precision, roc-auc score)
3. Artifacts (serialized model file, plots)
4. Source code
5. Environment (python version, library dependencies)
6. Version control information (version control state of the database)
7. Unique experiment ID

A model registry in MLOps is a centralized repository or system that manages the storage, versioning, and lifecycle of machine learning models. It serves as a catalog where models are tracked from development to deployment, ensuring that the best-performing models are readily available for production. The model registry is essential for managing the complexity of deploying and maintaining models in a reliable and reproducible manner.

1. Model Versioning
   - Definition: Track different versions of a model as it evolves over time.
   - Tasks:
     - Assign unique version identifiers to each model version.
     - Maintain a history of changes and updates for each model.
     - Support rollback to previous versions if needed.

2.Metadata Management

   - Definition: Store and manage metadata associated with each model.
   - Tasks:
     - Record model-related information such as author, creation date, description, and tags.
     - Store performance metrics, hyperparameters, and training data references.
     - Capture details about the environment in which the model was trained.

3. Model Lineage Tracking

   - Definition: Track the lineage and provenance of models to understand their development history.
   - Tasks:
     - Document the data preprocessing steps, feature engineering, and model training processes.
     - Link models to the specific experiments and runs that produced them.
     - Provide traceability from raw data to the final model.

4. Stage Management

   - Definition: Manage the lifecycle stages of a model (e.g., development, staging, production).
   - Tasks:
     - Define and manage different stages such as "development," "staging," "production," and "archived."
     - Support model promotion and demotion between stages based on validation and approval processes.
     - Enforce stage-specific policies and controls.

5. Model Storage and Retrieval

   - Definition: Store and provide access to model artifacts.
   - Tasks:
     - Securely store model files and artifacts in a centralized repository.
     - Ensure efficient retrieval and loading of models for inference and

5. Model Storage and Retrieval

- Definition: Store and provide access to model artifacts.
- Tasks:
  - Securely store model files and artifacts in a centralized repository.
  - Ensure efficient retrieval and loading of models for inference and further training.
  - Provide access controls and permissions to manage who can view and retrieve models.

6. Integration with CI/CD Pipelines

- Definition: Integrate with continuous integration and continuous deployment (CI/CD) pipelines.
- Tasks:
  - Automate the process of registering new models as part of CI/CD workflows.
  - Trigger model validation, testing, and deployment steps based on CI/CD pipeline stages.
  - Ensure seamless integration with tools like Jenkins, GitHub Actions, GitLab CI/CD, and others.

Recap

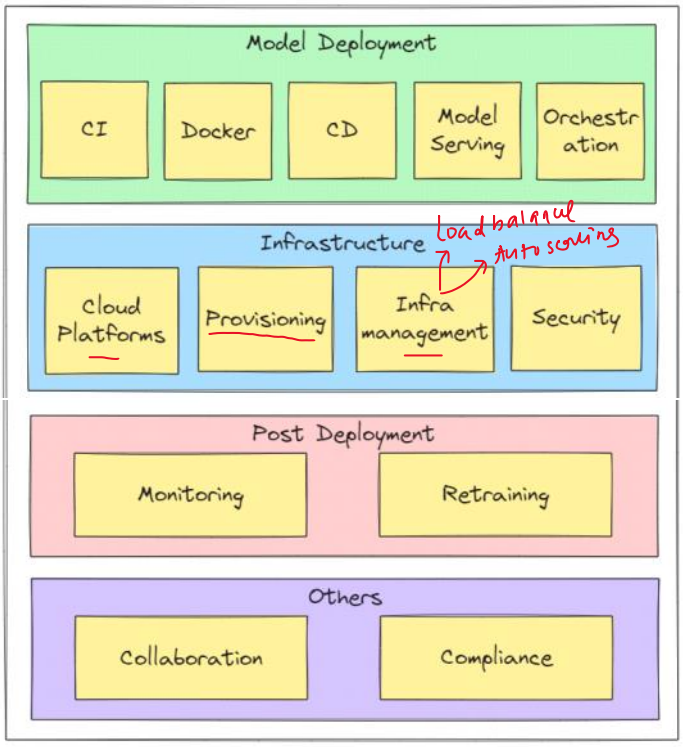
07 June 2024 11:26

4. Deployment & Monitoring

13 May 2024 23:56

CI/CD

CI → message → docker image



CI/CD

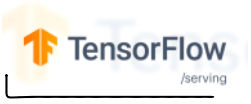


Containerization

Orchestration

Security

Model Serving



Monitoring

General Tools

Cloud Native Tools

ML Specific

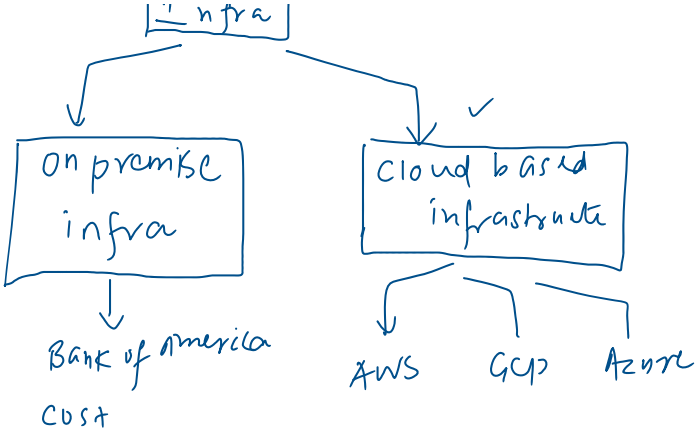
Cloud Platforms


Provisioning

Infrastructure Management

Collaboration

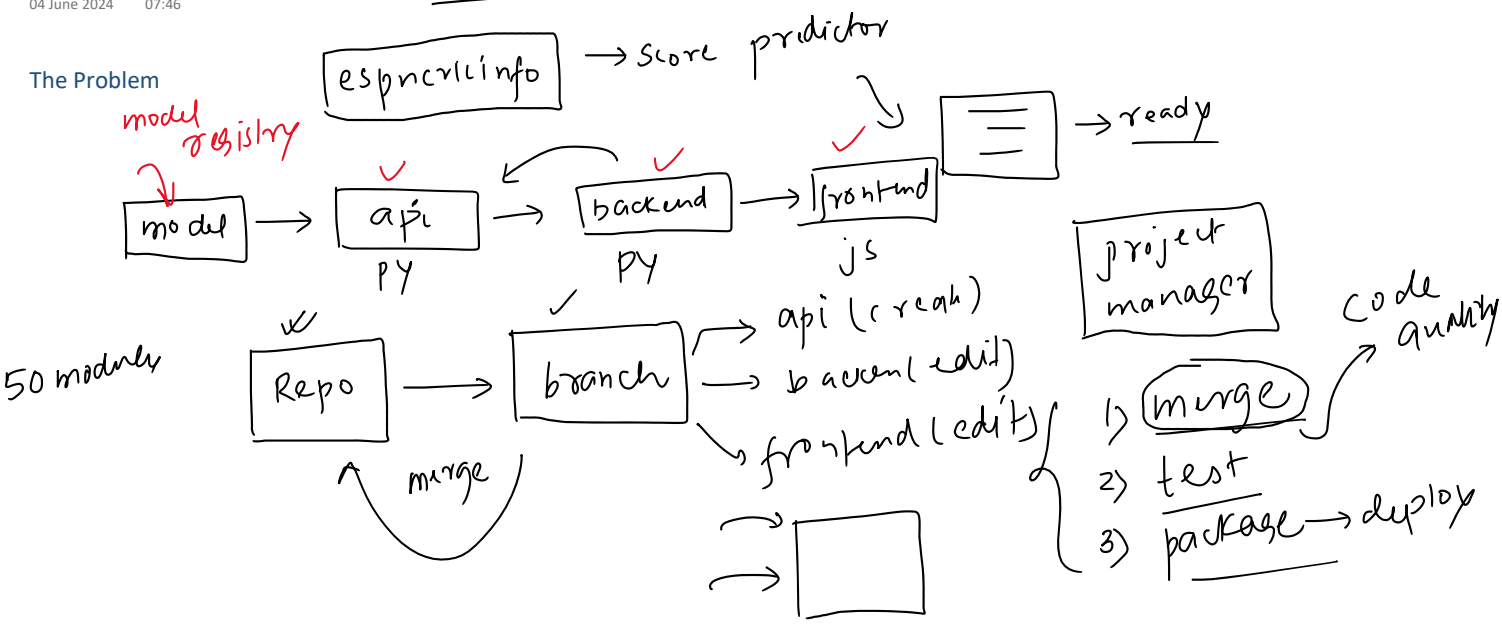
Infra


Infra
On premise infra
Cloud based infrastructure
Bank of America
AWS
GCP
Azure
Cost

Continuous Integration

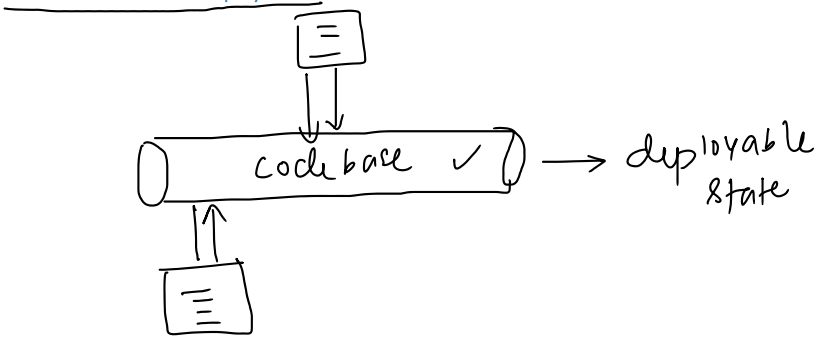
04 June 2024 07:46

The Problem

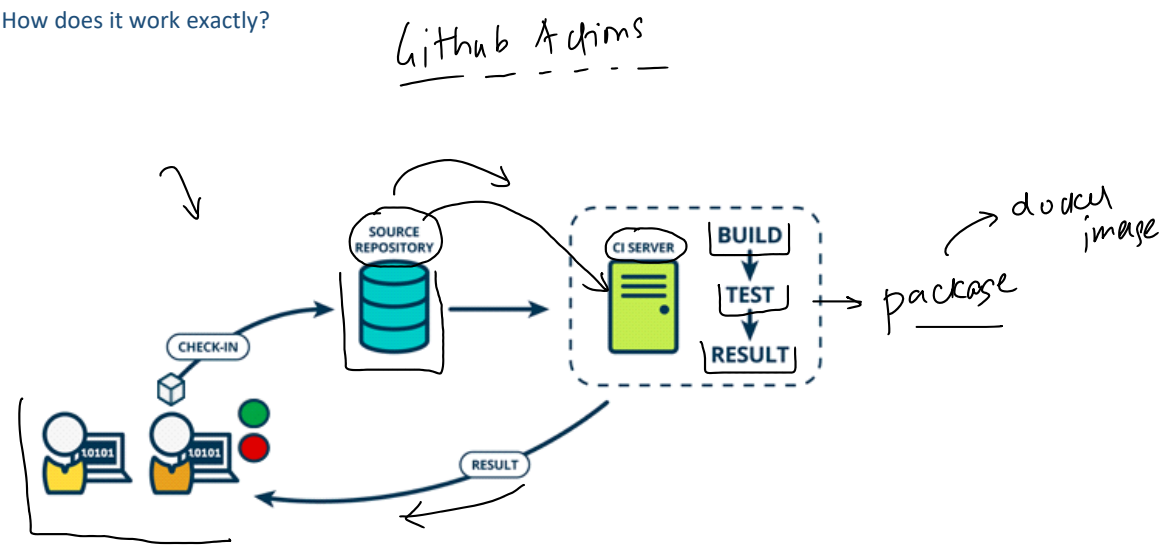


The Solution Continuous Integration

Continuous Integration (CI) is a software development practice where developers frequently integrate their code changes into a shared repository, typically multiple times a day. Each integration is automatically verified by running tests to detect integration errors as quickly as possible. The primary goals of CI are to improve software quality, reduce the time taken to deliver software, and ensure that the codebase remains in a deployable state.



How does it work exactly?



Exact steps executed during the CI workflow

1. Checkout Code

- Retrieve the latest code from the version control system (e.g., Git).
- Example tool: GitHub Actions, Jenkins, CircleCI.

2. Set Up Environment

- Configure the environment needed for the build and tests.
- This includes setting up the programming language runtime, dependencies, and environment variables.

3. Install Dependencies

- Install all the dependencies required for the application.
- This step ensures that the application has all the necessary libraries and tools to build and run.

4. Build the Application

- Compile the source code and prepare build artifacts.
- This step is essential for languages that require compilation (e.g., Java, C++).

5. Run Static Code Analysis

- Analyze the source code for potential errors, code smells, and adherence to coding standards.
- Tools like pylint, SonarQube, or ESLint can be used.

6. Run Automated Tests

- Execute unit tests, integration tests, and possibly end-to-end tests.
- This step ensures that the code changes do not break existing functionality.

7. Generate Test Reports

- Generate reports from the test execution.
- This often includes test coverage reports, which indicate the percentage of code covered by tests.

8. Package the Application (Optional)

- Package the application into a deployable format (e.g., Docker image, JAR file).
- This step is sometimes included in CI workflows if it's part of the build process.

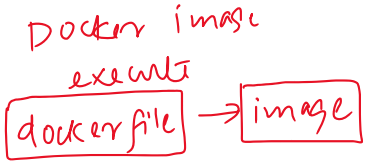
9. Upload Artifacts (Optional)

- Upload build artifacts to a central repository or storage.
- This step is useful for sharing build artifacts with other stages of the CI/CD pipeline.

10. Notify Developers

- Send notifications about the build status to the development team.
- This can be done via email, Slack, or other communication tools.

```yaml
name: CI Workflow

on:
  push:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: '3.8'

  - name: Install dependencies
    run: |
      python -m venv venv
      . venv/bin/activate
      pip install -r requirements.txt

  - name: Run static analysis
    run: |
      . venv/bin/activate
      pylint mymodule/

  - name: Run tests
    run: |
      . venv/bin/activate
      pytest --junitxml=report.xml

  - name: Generate coverage report
    run: |
      . venv/bin/activate
      pytest --cov=mymodule --cov-report=xml

  - name: Build Docker image
    run: docker build -t myapp:latest .

  - name: Push Docker image
    run: |
      echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u "${{
      docker push myapp:latest

  - name: Notify
    if: always()
    uses: actions/slack@v2
    with:
      status: ${{ job.status }}
```

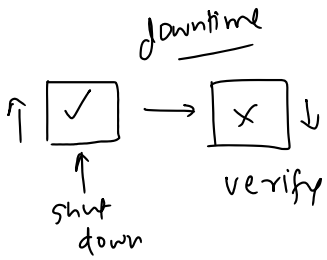


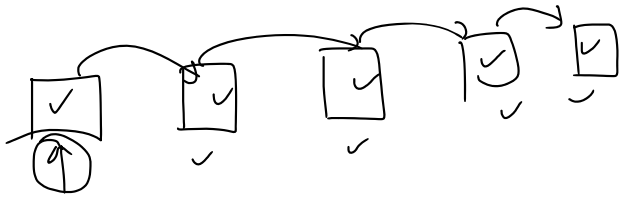
Continuous Deployment

CI/CD

04 June 2024 10:13

The Problem





The manual deployment process

1. Manual integration
2. Manual building
3. Manual testing
4. Manual deployment
5. Manual verification
6. Manual rollback
7. Difficulty in executing different deployment strategies → canary deployment

Disadvantages

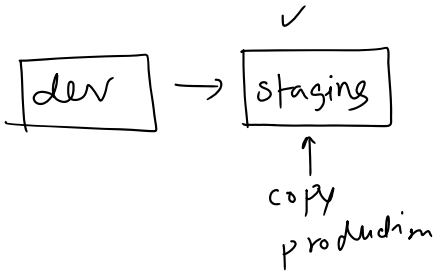
1. Time consuming
2. Labor intensive
3. Error prone
4. Deployment downtime

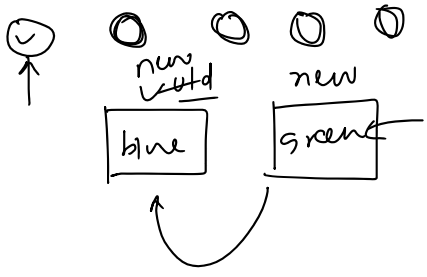
What is CD

Continuous Deployment (CD) is a software development practice where code changes are automatically deployed to production as soon as they pass predefined automated tests. CD extends the principles of Continuous Integration (CI) by automating the release process, ensuring that software is always in a deployable state and can be released quickly and reliably.

Continuous Deployment Workflow

1. Code Commit:
   - Developers commit code changes to the version control system (e.g., Git).
2. Continuous Integration (CI):
   - The CI pipeline is triggered, which checks out the latest code, runs automated tests, and builds the application.
3. Build and Package:
   - The application is built and packaged into deployable artifacts (e.g., Docker images, binaries).
4. Development to Staging:
   - The new version is deployed to a staging environment where further automated tests and validations are performed.
5. Approval (Optional):
   - Some workflows include a manual approval step before deploying to production, especially for critical applications.
6. Deployment to Production:
   - The new version is automatically deployed to the production environment using deployment strategies like rolling updates, blue/green deployments, or canary releases.
7. Monitoring and Rollback:
   - The production environment is monitored for any issues. Automated rollback mechanisms are in place to revert to the previous version if necessary.





Sample Workflow

```yaml
name: CI/CD Pipeline

on:
  push:
    branches:
      - main

jobs:
  build-and-test:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.8'
```

```yaml
      - name: Install dependencies
        run: |
          python -m venv venv
          . venv/bin/activate
          pip install -r requirements.txt

      - name: Run static analysis
        run: |
          . venv/bin/activate
          pylint mymodule/

      - name: Run tests
        run: |
          . venv/bin/activate
          pytest --junitxml=report.xml

      - name: Build Docker image
        run: docker build -t user/myapp:latest .
```

```yaml
      - name: Push Docker image to Docker Hub
        run: |
          echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u "${{ secrets.DOCKER_USER
          docker push user/myapp:latest

  deploy:
    runs-on: ubuntu-latest
    needs: build-and-test

    steps:
      - name: Set up SSH
        uses: webfactory/ssh-agent@v0.5.3
        with:
          ssh-private-key: ${{ secrets.SSH_PRIVATE_KEY }}
```

```yaml
      - name: Deploy to Production
        run: |
          ssh -o StrictHostKeyChecking=no ec2-user@your-ec2-public-dns << 'EOF'
          docker pull user/myapp:latest
          if [ $(docker ps -q -f name=myapp) ]; then
              docker stop myapp
              docker rm myapp
          fi
          docker run -d --name myapp -p 80:5000 user/myapp:latest
          EOF
```

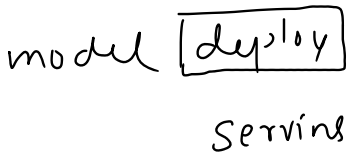
Continuous Deployment vs Continuous Delivery

Continuous Deployment vs Continuous Delivery

Model Serving

04 June 2024 11:58

Model serving is the process of making the deployed model accessible for inference (i.e., predictions) by exposing it through an API or other interface so that applications can send data to the model and receive predictions.



Main tasks

1. API development - Develop RESTful or gRPC APIs to expose the model.
2. Load the model into the serving environment, ensuring it's ready for inference.
3. Accept and validate incoming data, perform inference, and return predictions.
4. Handling scalability
5. Handling latency
6. Security

Cloud Platforms - AWS vs GCP vs Azure

22 May 2024 08:12

AWS Service	Description	Example Use Case	GCP Service	Azure Service
Amazon SageMaker	Comprehensive service for building, training, and deploying ML models.	Automating the entire ML workflow from data preparation to model deployment.	Google AI Platform	Azure Machine Learning
AWS Lambda	Run code without provisioning or managing servers.	Running a preprocessing function for incoming data before it is stored.	Google Cloud Functions	Azure Functions
AWS Glue	ETL (Extract, Transform, Load) service for data preparation.	Preparing raw data for analysis by transforming and cleaning it.	Google Cloud Dataflow	Azure Data Factory
Amazon EMR	Big data processing with Hadoop and Spark.	Processing large datasets for training ML models.	Google Cloud Dataproc	Azure HDInsight
Amazon Redshift	Data warehousing service.	Storing and querying large amounts of structured data for analytics.	BigQuery	Azure Synapse Analytics
Amazon S3	Object storage service.	Storing training data,	Google Cloud	Azure Blob
AWS Step Functions	Orchestrate workflows and handle model pipelines.	Coordinating different stages of the ML pipeline, from data preprocessing to model deployment.	Google Cloud Composer	Azure Logic Apps
Amazon Kinesis	Real-time data streaming and analytics.	Ingesting and processing real-time data streams for live predictions.	Google Cloud Dataflow	Azure Stream Analytics
AWS Batch	Run batch computing workloads.	Batch processing of large volumes of data for model training.	Google Cloud Batch	Azure Batch
Amazon	Monitoring and	Monitoring the	Google Cloud	Azure Monitor

		for model training.		
Amazon CloudWatch	Monitoring and observability service.	Monitoring the performance and health of ML models in production.	Google Cloud Monitoring	Azure Monitor
AWS CodePipeline	CI/CD service to automate code deployments.	Automating the deployment of ML models and updates.	Google Cloud Build	Azure DevOps Pipelines
AWS CodeBuild	Service to compile and test code.	Building and testing ML model code before deployment.	Google Cloud Build	Azure DevOps Pipelines
AWS CodeDeploy	Service to automate application deployments.	Automating the deployment of ML models to different environments.	Google Cloud Build	Azure DevOps Pipelines
AWS CodeCommit	Source control service for managing code repositories.	Managing version control for ML model code and scripts.	Google Cloud Source Repositories	Azure Repos
AWS CloudFormation	Infrastructure as code service for provisioning AWS resources.	Deploying and managing infrastructure needed for ML workflows.	Google Cloud Deployment Manager	Azure Resource Manager
Amazon RDS	Managed relational database service.	Storing structured data used for training and validating models.	Google Cloud SQL	Azure SQL Database
Amazon DynamoDB	Managed NoSQL database service.	Storing large volumes of unstructured data for model training.	Google Cloud Firestore	Azure Cosmos DB
AWS Identity and Access Management	Service for managing user access and encryption keys.	Controlling access to ML models and data.	Google Cloud IAM	Azure Active Directory

AWS Data Pipeline	Orchestration service for data-driven workflows.	Moving data between different AWS services for ML processing.	Google Cloud Data Fusion	Azure Data Factory
AWS Fargate	Serverless compute engine for containers.	Running containerized ML inference tasks without managing servers.	Google Cloud Run	Azure Container Instances
Amazon ECR	Docker container registry.	Storing and managing Docker images for ML applications.	Google Cloud Container Registry	Azure Container Registry
AWS App Mesh	Service mesh to manage microservices.	Managing communication between microservices in an ML application.	Istio	Azure Service Fabric

Amazon EC2	Scalable virtual server service.	Running compute-intensive ML training jobs.	Google Compute Engine	Azure Virtual Machines
Amazon ECS	Container orchestration service.	Orchestrating Docker containers for ML applications.	Google Kubernetes Engine	Azure Kubernetes Service
Amazon EKS	Kubernetes management service.	Managing Kubernetes clusters for ML workloads.	Google Kubernetes Engine	Azure Kubernetes Service
AWS Elastic Beanstalk	Platform as a service for web applications.	Deploying and scaling ML web applications.	Google App Engine	Azure App Service

Amazon Forecast	Service to generate time-series forecasts.	Building predictive models for time-series data.	Google Cloud AI Platform	Azure Machine Learning
Amazon Comprehend	Natural language processing (NLP) service.	Extracting insights from text data for ML models.	Google Cloud Natural Language	Azure Cognitive Services
Amazon Polly	Text-to-speech service.	Converting text data to speech for interactive ML applications.	Google Cloud Text-to-Speech	Azure Cognitive Services
Amazon Rekognition	Image and video analysis service.	Analyzing images and videos for ML models.	Google Cloud Vision	Azure Cognitive Services
Amazon Textract	Extract text and data from documents.	Extracting text from documents for ML processing.	Google Cloud Document AI	Azure Form Recognizer
Amazon Transcribe	Automatic speech recognition service.	Converting speech to text for ML models.	Google Cloud Speech-to-Text	Azure Cognitive Services
Amazon Translate	Language translation service.	Translating text data for multilingual ML applications.	Google Cloud Translation	Azure Cognitive Services

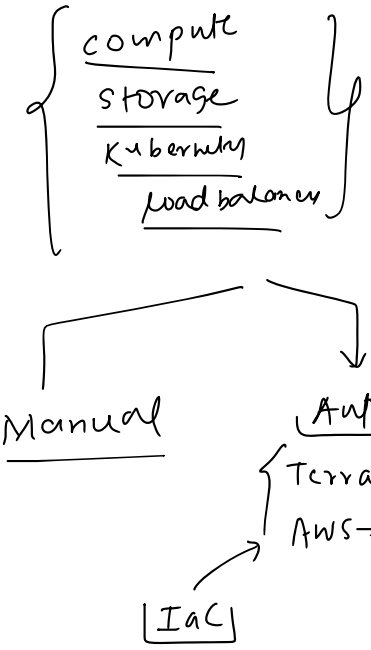
Provisioning

06 June 2024 10:30

What is provisioning
Food Example
Provisioning in MLOps
Steps in provisioning
- Requirements
- Configure ✓
- Deploy services ✓
- Networking and security ✓

Types of provisioning
- Manual
- Automated
  - IaC
  - Declarative syntax
- Benefits of Automated
  - Fast
  - Less error prone
  - reproducibility

Provisioning in MLOps involves creating, configuring and managing the necessary infrastructure and resources required to support the lifecycle of machine learning models, from development to deployment and serving. This process can be automated and managed using various tools and platforms to ensure efficiency, scalability, and reliability.



Define Requirements

Objective: Determine the computational, storage, and networking needs based on the specific ML workload and use case.

- Compute Requirements: Decide on the types and number of CPUs, GPUs, memory, and disk space needed.
- Storage Requirements: Identify the storage solutions for data, model artifacts, and logs.
- Network Requirements: Plan for networking components to ensure secure and efficient communication between different parts of the infrastructure.

Select Infrastructure

Objective: Choose the appropriate infrastructure to meet the defined requirements, considering options like cloud, on-premises, or hybrid setups.

- Cloud Providers: AWS, Google Cloud Platform (GCP), Microsoft Azure.
- On-Premises: Using local data centers with physical servers.
- Hybrid Solutions: Combining cloud and on-premises resources.

Provision Resources

Objective: Set up the required infrastructure components using automated tools to ensure consistency and repeatability.

- Infrastructure as Code (IaC): Use IaC tools to automate the provisioning process.
  - Terraform: Platform-agnostic tool for provisioning infrastructure across multiple providers.
  - AWS CloudFormation: For provisioning AWS resources.
  - Azure Resource Manager: For provisioning Azure resources.
  - Google Cloud Deployment Manager: For provisioning GCP resources.

Example Terraform Script:

```hcl
provider "aws" {
  region = "us-west-2"
}

resource "aws_instance" "ml_server" {
  ami           = "ami-0abcdef1234567890"
  instance_type = "t2.medium"

  tags = {
    Name = "MLServer"
  }
}
```

Configure Environments

Objective: Install and configure necessary software, libraries, and dependencies in the provisioned infrastructure.

- Operating Systems: Ensure the correct OS is installed and configured (e.g., Ubuntu, CentOS).
- Dependencies: Install required libraries and frameworks (e.g., TensorFlow, PyTorch, Scikit-learn).
- Environment Management: Use tools like Conda or virtual environments to manage dependencies.

Example Bash Script:

```bash
#!/bin/bash
# Update and install dependencies
sudo apt-get update
sudo apt-get install -y python3-pip
pip3 install tensorflow scikit-learn
```

Deploy Services

Objective: Deploy essential services needed for MLOps, including databases, model registries, and CI/CD pipelines.

- Databases: Set up databases for storing data (e.g., PostgreSQL, MySQL).
- Model Registry: Use a model registry to manage and version models (e.g., MLflow, DVC).
- CI/CD Pipelines: Set up CI/CD pipelines for automating model training, testing, and deployment (e.g., Jenkins, GitLab CI, GitHub Actions).

Set Up Networking

Objective: Configure networking to ensure secure and efficient communication between different components.

- VPCs and Subnets: Configure Virtual Private Clouds (VPCs) and subnets to isolate resources.
- Load Balancers: Set up load balancers to distribute traffic across multiple instances.
- Firewalls and Security Groups: Configure firewalls and security groups to control access.

Example AWS VPC Configuration:

```hcl
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_subnet" "subnet1" {
  vpc_id            = aws_vpc.main.id
  cidr_block        = "10.0.1.0/24"
  availability_zone = "us-west-2a"
}

resource "aws_security_group" "allow_http" {
  vpc_id = aws_vpc.main.id

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

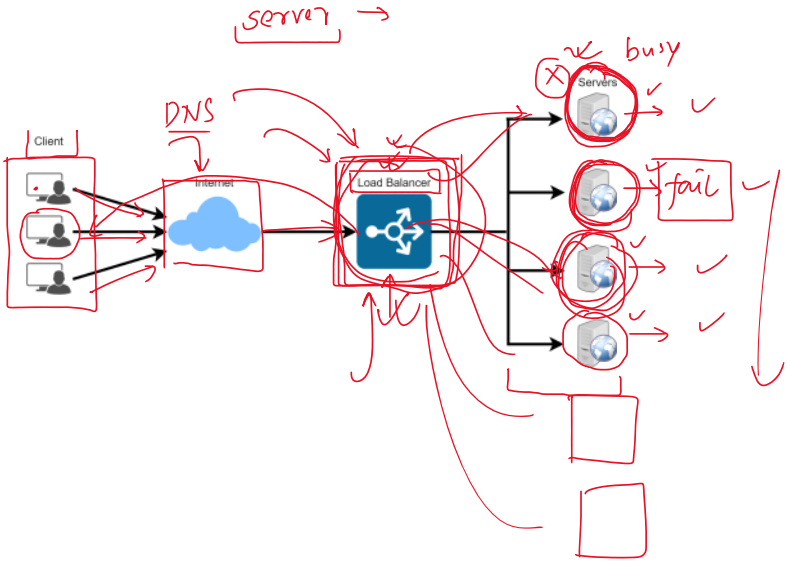
Implement Security Measures

Objective: Ensure the infrastructure is secure and complies with relevant regulations.

- Access Control: Set up IAM roles and policies for access management.
- Encryption: Implement encryption for data at rest and in transit (e.g., SSL/TLS).

Load Balancers

06 June 2024 14:57



What is a Load Balancer?

A load balancer is a device or software that distributes network or application traffic across multiple servers. By evenly distributing incoming traffic, load balancers ensure that no single server becomes overwhelmed, which helps in achieving high availability and reliability for applications.

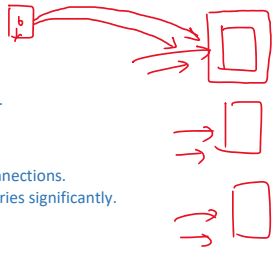
Why Do We Need a Load Balancer?

1. Improved Availability and Reliability:
   - Ensures that applications remain accessible even if one or more servers fail.
   - Redirects traffic from failing servers to healthy ones.
2. Scalability:
   - Facilitates horizontal scaling by adding more servers to handle increased traffic.
   - Automatically distributes traffic to new servers as they are added.
3. Optimized Resource Utilization:
   - Balances the load to prevent any single server from becoming a bottleneck.
   - Enhances the overall performance by utilizing all available resources efficiently.
4. Maintenance and Updates:
   - Allows servers to be taken offline for maintenance without disrupting service.
   - Can direct traffic to servers running the latest updates or versions.
5. Security:
   - Can provide a single point of entry, making it easier to enforce security policies.
   - Often integrates with SSL/TLS to encrypt traffic, ensuring secure communication.

How Exactly Does a Load Balancer Work?

1. Traffic Distribution:
   - Incoming Requests: The load balancer receives incoming client requests.
   - Traffic Routing: It uses various algorithms to determine which backend server should handle each request.
2. Health Checks:
   - Regularly checks the health of backend servers to ensure they can handle requests.
   - Redirects traffic away from unhealthy or down servers.
3. Session Persistence:
   - Ensures that requests from a particular client are directed to the same backend server for the duration of the session, if needed.
   - Useful for applications that require session affinity (e.g., shopping carts).
4. SSL Termination:
   - Offloads SSL decryption/encryption from backend servers, improving their performance.
   - Provides a single point to manage SSL certificates.
5. Types of Load Balancers:
   - Hardware Load Balancers: Physical devices deployed in data centers.
   - Software Load Balancers: Installed on servers and can be cloud-based.
   - Cloud Load Balancers: Managed services offered by cloud providers (e.g., AWS Elastic Load Balancer, Google Cloud Load Balancer).

Load Balancing Algorithms



1. Round Robin:
   - Distributes requests sequentially across all servers.
   - Simple and effective for evenly distributed workloads.
2. Least Connections:
   - Directs traffic to the server with the fewest active connections.
   - Ideal for environments where connection duration varies significantly.
3. IP Hash:
   - Distributes requests based on the client's IP address.
   - Ensures that a client is consistently routed to the same server.
4. Weighted Round Robin:
   - Distributes requests based on server weight, allowing more powerful servers to handle more traffic.
5. Random:
   - Distributes traffic randomly among servers.
   - Can be effective in evenly distributed environments.

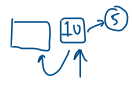
Example Workflow

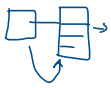
1. Client Request:
   - A client sends a request to access a web application.

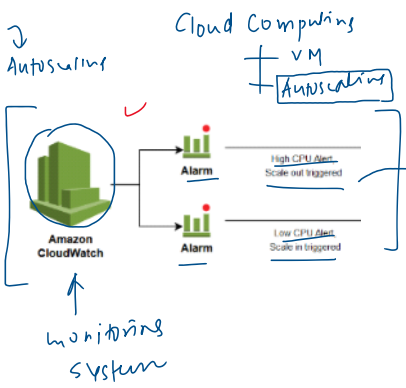
2. DNS Resolution:
    - The client's DNS query is resolved to the load balancer's IP address.
3. Load Balancer Receives Request:
    - The load balancer receives the incoming request.
4. Health Check:
    - The load balancer checks the health of the backend servers.
5. Routing Decision:
    - Based on the load balancing algorithm, the load balancer selects a healthy server to handle the request.
6. Forwarding the Request:
    - The load balancer forwards the client's request to the selected server.
7. Server Response:
    - The backend server processes the request and sends the response back to the load balancer.
8. Load Balancer Response:
    - The load balancer forwards the server's response to the client.

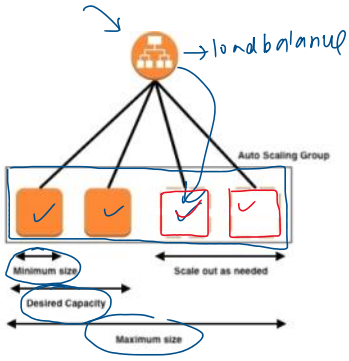
Auto Scaling

06 June 2024 14:28









Amazon Auto Scaling Groups (ASGs) are a fundamental feature in AWS that automatically adjust the number of Amazon EC2 instances in response to changing demand. This ensures that the right number of EC2 instances are running to handle the load for your application. Here's an in-depth look at how ASGs work in AWS, including details on scaling policies, monitoring with CloudWatch, and cost considerations.

Key Components of Auto Scaling Groups

1. Launch Configuration:
   - Launch Configuration: Specifies the EC2 instance configuration, including the instance type, Amazon Machine Image (AMI), key pair, security groups, and other instance settings.
2. Auto Scaling Group:
   - Defines the minimum, maximum, and desired number of EC2 instances.
   - Associates with one or more subnets for distributing instances across Availability Zones (AZs) to improve fault tolerance.
3. Scaling Policies:
   - Dynamic Scaling Policies: Automatically adjust the number of instances based on real-time metrics (e.g., CPU utilization, network traffic).
     - Target Tracking Scaling: Adjusts the number of instances to keep a specific metric (e.g., CPU utilization) at the target value.
   - Predictive Scaling Policies: Uses machine learning to predict future traffic and schedules scaling actions ahead of time.
   - Custom Policies:
     - Load on ELB: Scales instances based on the load metrics from an Elastic Load Balancer.
     - Scheduled Scaling: Allows you to schedule scaling actions to occur at specific times to handle predictable traffic patterns.

How ASGs Work

1. Configuration and Creation:
   - Define a launch configuration or launch template that specifies the EC2 instance settings.
   - Create an ASG and specify the desired, minimum, and maximum number of instances.
   - Attach the launch configuration or launch template to the ASG.
   - Define the subnets and Availability Zones where the instances will be launched.
   - Configure scaling policies to determine how the group should scale in and out.
2. Launching Instances:
   - The ASG ensures that the desired number of instances are running. If there are fewer instances than desired, it launches new instances based on the configuration.
3. Monitoring and Scaling:
   - The ASG continuously monitors the instances and the metrics specified in the scaling policies.
   - When a scaling policy is triggered (e.g., CPU utilization exceeds a target), the ASG adjusts the number of instances accordingly.
   - Scaling Out: Adds instances when the demand increases.
   - Scaling In: Terminates instances when the demand decreases.
4. Health Management:
   - ASGs perform regular health checks on instances. Health checks can be EC2 status checks or custom health checks from an Elastic Load Balancer (ELB).
   - If an instance is found to be unhealthy, the ASG terminates it and launches a new one to replace it.
5. Load Balancing Integration:
   - ASGs can be integrated with Elastic Load Balancers (ELB) to distribute incoming traffic across the instances in the group.
   - Ensures that traffic is directed to healthy instances and helps maintain high availability.

Monitoring with CloudWatch

Amazon CloudWatch plays a crucial role in monitoring the performance and health of resources within an ASG:

- Metrics Collection: CloudWatch collects and tracks metrics for EC2 instances, such as CPU utilization, network traffic, disk I/O, and more.
- Alarms: CloudWatch Alarms can be configured to trigger scaling actions based on specific metrics, ensuring that the ASG responds to changes in demand in real time.
- Logs: CloudWatch Logs can store and monitor log data from instances, providing deeper insights into application performance and issues.
- Dashboards: CloudWatch Dashboards offer a customizable view of metrics and alarms, allowing for comprehensive monitoring of the ASG and its instances.

Example Scenario

Suppose you have a web application with variable traffic. You can use an ASG to automatically scale the number of instances based on traffic load:

1. Create a Launch Template:

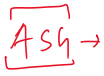
```python
import boto3

ec2_client = boto3.client('ec2')

response = ec2_client.create_launch_template(
    LaunchTemplateName='my-launch-template',
    VersionDescription='WebAppTemplate',
    LaunchTemplateData={
        'ImageId': 'ami-12345678',
        'InstanceType': 't2.micro',
        'KeyName': 'my-key-pair',
        'SecurityGroupIds': ['sg-12345678'],
        'UserData': 'base64-encoded-user-data-script'
    }
)
```

2. Create an Auto Scaling Group:

```python
autoscaling_client = boto3.client('autoscaling')

response = autoscaling_client.create_auto_scaling_group(
    AutoScalingGroupName='my-asg',
    LaunchTemplate={
        'LaunchTemplateName': 'my-launch-template',
        'Version': '$Latest'
    },
    MinSize=1,
    MaxSize=10,
    DesiredCapacity=2,
    VPCZoneIdentifier='subnet-12345678,subnet-87654321',
    HealthCheckType='EC2',
    HealthCheckGracePeriod=300
)
```

3. Attach Scaling Policies:

```python
response = autoscaling_client.put_scaling_policy(
    AutoScalingGroupName='my-asg',
    PolicyName='scale-out-policy',
    PolicyType='TargetTrackingScaling',
    TargetTrackingConfiguration={
        'PredefinedMetricSpecification': {
            'PredefinedMetricType': 'ASGAverageCPUUtilization'
        },
        'TargetValue': 50.0
    }
)
```



Cost Considerations

Using Auto Scaling Groups is a free service in AWS. However, you will be charged for the resources that the ASG provisions and uses:

- EC2 Instances: You pay for the EC2 instances launched by the ASG based on the instance type and usage.
- Storage: Costs associated with the storage of AMIs and EBS volumes.
- Data Transfer: Charges for data transfer between instances and other AWS services.

By leveraging ASGs, you can ensure that your application maintains high availability, handles varying levels of traffic efficiently, and operates cost-effectively, with the added benefit of integrated monitoring and security features.

5. Workflow Management

14 May 2024 01:42

Rollout
Rollback
Retraining

MLOps Maturity Levels

13 May 2024 16:06

How to select a MLOps tool?

13 May 2024 08:18