Date:

Kubernetes CK8
└ Fireship

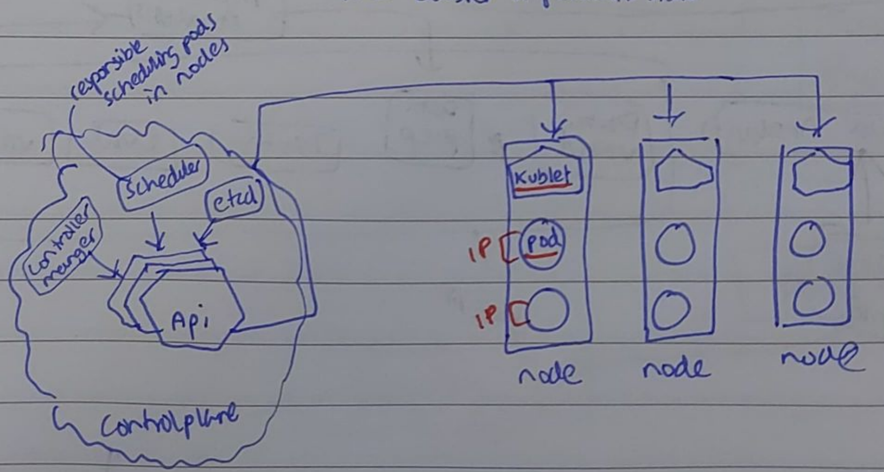
managing containerized workloads in cloud

e.g Kubernetes for binance – market closed
– market open → scale containers
└ replace container if one fail

A system deployed on Kubernetes is called a Cluster

Brain of operation is the control plane which exposes a api server which can handle internal external req to manage cluster

also contains it key/value database called ETCD – store info about running the cluster or persistant state



→ control plane manages worker machines called Nodes

→ each Node runs a Kublet
└ tiny application to communicate back with the main control plane mothership

→ each node contains multiple Pods
└ smallest deployable unit in K8s
  └ group of containers / 1 or more
└ provides share storage / network for containers
└ pods ephemeral = die/spawn new IP
  └ each pod has own IP address

Node – machine ~~Bspace~~ where containers run | Pod – etc – containers

Pods
→ Pod die/respawn new IP to be created so
Service used
internal — within cluster
external — accessible outside
├ Static/Permanent IP address for each Pod
└ lifecycle of Pod, service not connected

Ingress
└ service is https://124.89.101.2:8080
└ convert to https://myapp.com
└ ingress

e.g ~~K8~~ Cluster

```
|
|—— Node 1 (worker)
|        ├── Pod A → runs redis
|        └── Pod B → runs nginx
|
|—— Node 2 (worker)
|        |—— Pod C → runs backend app
|        |—— Pod D → run database
|
|
|__ . Node 3 (control/master)
```

→ workload increases then K8 increase no of ~~more~~ Nodes &
└ also takes care of
- networking
- secret managment
- persistent storage ...

→ K8 is designed for High Availability
└ way it achieves by maintaining replica set:
└ set of running pods ready to go at any time

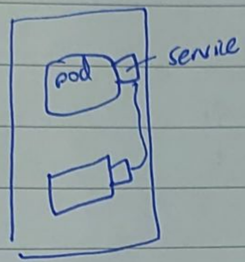
→ You describe Objects in yaml
└ define state of cluster

```
apiVersion: apps/v1
Kind: Deployment
metadata:
    name: nginx-deployment
spec:                              //describe behavior in spec field
    replicas: 3                        └ replicas
    containers:                          container
    - name: ngnix
      image: ngnix:1.14.2
      ports:
      - containersPort: 80
      volumes:
      - name: cool-volume
```

Date:

★ configmap

if pod changed the endpoint for communication b/w pods change and thus you have to rebuild / ~~the~~ image, make pod



solution: configmap
└ outside of image just rename new pod's name with endpoint
DB-url = mongo-db-service
endpoint · pod-name

★ Secrets
└ alike configmap but it user, psswrd of a database
new pod = diff user, psswrd so use secret to store these
stored in base64
use 3rd party encryption for further security

★ Volume
└ ~~same~~ stores persistent data
└ local
└ remotely
└ outside of pod
└ referenced ~~to~~ the storage.

Date:

★ Deployment — replicas of pod
└ abstraction above pods
└ if pod dies substitute with replica

deployment config file. (prev page)

metadata:

spec:

status: //running status etc.

- Problem: we cant replicate ~~DB~~ Db pod to ensure data consistency
so use statefulset

DVC data versioning control

git cant store large dataset, model in ML
so we use dvc (tool) for track this
- dvc ~~makes~~ / tracks real dataset / mode
- git tracks metadata files of dataset / model

• git + dvc used in collaboration
  └ links to an external drive (Gdrive, S3

Base project → Example workflow

```
ml-project/
├── data/
│   └── train.csv
├── model/
│   └── r
├── src/
│   ├── preprocessing.py
│   └── train.py
├── params.yml
├── readme.md
└── runmodel.py
```

1) git init //makes .git folder
   dvc init // makes .dvc folder

~~#Track dataset~~
2) dvc add data/train.csv // track dataset

∘ metadata file appears → train.csv.dvc

```
data/
├── train.csv
└── train.csv.dvc
```

~~∘ Data copied to .dvc/ cache~~

git add data/train.csv.dvc .gitignore
git commit

dvc config core.autostage true // automatically track file changes
└ auto ~~adds~~, runs $\boxed{\text{dvc add}}$

Date:

* Tracking changes
  1. change data : a script like get-data.py changes data
  2. check status :
     - Git doesn't find changes
     - but dvc does dvc status
  3. Add change to dvc dvc add data.csv
  4. Git with note data.csv.dvc has changed add it
  5. Commit

* Go back to previous data
  dvc logs // find prev version
  git checkout
  dvc checkout

* Remote storage setup

Add remote : dvc remote add g-drive-remote gdrive :// (folder-URL)
This command changes config file so git add , commit
do get client-ID , secret from google

To push: dvc push -r gdrive-remote
dvc pull

Date:

file automate
getdata, preprocess, train

Automation : DVC pipeline , dvc.yml

dvc → used to automate pipeline to retrain model - when new data comes

dvc repro // reruns pipeline
dvc metrics show
dvc params show
dvc metrics diff // show differences in metrics

How it knows to rerun

dvc checks
1- scripts (deps)
2- datafile (outs)
3- parameters (params.yml)
for changes
if found
dvc repro

dvc.yml e.g

```
stages:
    get-data:
        cmd: python src/get_data.py
        deps:
          - src/get-data.py
        outs:
          - data/raw.csv

    preprocess:
        ...
    train ....
```

Automate commits or CICD using Github Actions

1- Run dvc repro
2- Add updated files
3- commit, push

create .github/workflows/auto-train.yml file

~~pede~~

container
mlflow

Date:

```
name : Auto train pipeline

on :
    push :
        branches:
            - main
        pull request :

jobs :
    run-pipline:
    runs-on : ubunto-latest

    steps :
        - names : Set-up python
          uses : actions/setup-pytho
          with :
              python-version : 3.10

        - name : Install dependence
          run : pip install requirement.txt

        - name : Pull data
          runs : dvc pull --force

        - name : rerun pipeline
          run : dvc repro
```

Date:

MLflow

└ tool/library in python to manage ML experiments

helps in

- Tracks experiments
- Reproduce training runs anywhere
- Save / Load models easily
- version & manage models

→ 1

→ performance scores (e.g accuracy)

→ files (plots, save models

☆ Components	Purpose	e.g
Tracking	log experiments (parameter, metrics, artifacts)	compare model / tuning results
Projects	Package code so can run anywhere	Run training code anywhere
models	Save models in standard format	deploy model anywhere
Model Registry	version Manage models	promote best model to production

1 Tracking

```
import mlflow
import mlflow.sklearn

with mlflow.start_run():
        mlflow.log_param("learning_rate", 0.01)
          "      .log_metric
          "      .log_artifact
        mlflow.sklearn.log_model(model)

$ mlflow ui        // start mlflow ui, go to localhost:5000
$ mlflow.autolog()  // autodetects model & logs every param
```

Workflow

└ 1. install mlflow

2. import mlflow
3. initilize (set server IP)
3. train model as normal
4. start run
   └ log

Tip

hyperparameter tuning using gridsearchcv

└ method to auto test multiple combination of params to find best

└ you give
- model
- dictionary of params

it returns
best model + best params

grid = GridSearch(model, param_dictionary, cv=3 ...)
grid.fit(x, y)

Date:

2) Projects

requires Git repo

This file

create MLproject file

```
name: my-ml-project
conda-env: conda.yml
entry point:
  main:
    parameters:
      n_estimators: {type: int, default: 100}
      max-depth: { ... - - - }
    command: "python train.py --n-estimators {n_estimator} . . . . "
```

function that can be run

optional: conda.yml file for dependency

* Running the project

$ mlflow run .   // default param

$ mlflow run . -P n-estimator=200 -P max-depth=8 //

3) Models

when you save a model using mlflow it creates a repo like this

~~saved model~~

model/
- MLmodel //metadata of model
- model.pkl //actual model
- conda.yml //dependencies

* model flavours

pytorch	mlflow.sklearn
scikit-learn	. pytorch
XG-boost	. "
Tensorflow	. "

example

```
import mlflow.sklearn
with mlflow.start_run():
    model = RandomForest
    model.sklearn.log_model(model)
```

saved in

mlruns/
 <experiment_id>/
  <run_id>/
   artifacts/
    model/
     model

mlflow saves model in standard format so

can be
- loaded into python
- used in webapp
- Deployed as Rest API

mlflow models serve -m runs:/<run_id>/model -p 1234

serve as API, now send request at localhost:1234

Date:

4) Model Registry

- Hub of models
  - track model version (v1, v2 ...
  - Promotes model : Staging → Production
  - stores notes, metadata, description

result = mlflow.register_model ("runs:/<run-id>/model", "HousePriceModel");

mlflow.transition_model_version_stage(
    name = "HouseP..."
    version=1
    stage = "Production"
)

1- python tracking.py — ran

2- mlflow run -e tracking --env manager local

→ didn't run

→ as cmd requires more info

→ we defined experiment name in tracking.py but didn't tell same name in cmd so it created another experiment & confused with the name

→ Add "--experiment-name Basic_Tracking_Demo" at end of cmd 2