HSLU Lucerne University of Applied Sciences and Arts

Containerization

Florian Bär
MLOps

Information Technology
16.10.2024

FH Zentralschweiz

Why?

- Any ideas?


works on
my
machine


cameo
cameo
cameo

Portability, Consistency, Scalability

Different machines have different

- Hardware
- Operating system
- Installed software

Goal: Application should run the same way on different machines

- Locally (for development)
- On-premise/private cloud
- Public cloud (different providers)

Overview

- Dependency management
- Containerization
  - Docker, Podman, …
- Managing containers
  - Kubernetes, Openshift, …
- Building pipelines
  - Airflow, Kubeflow, Metaflow, TFX

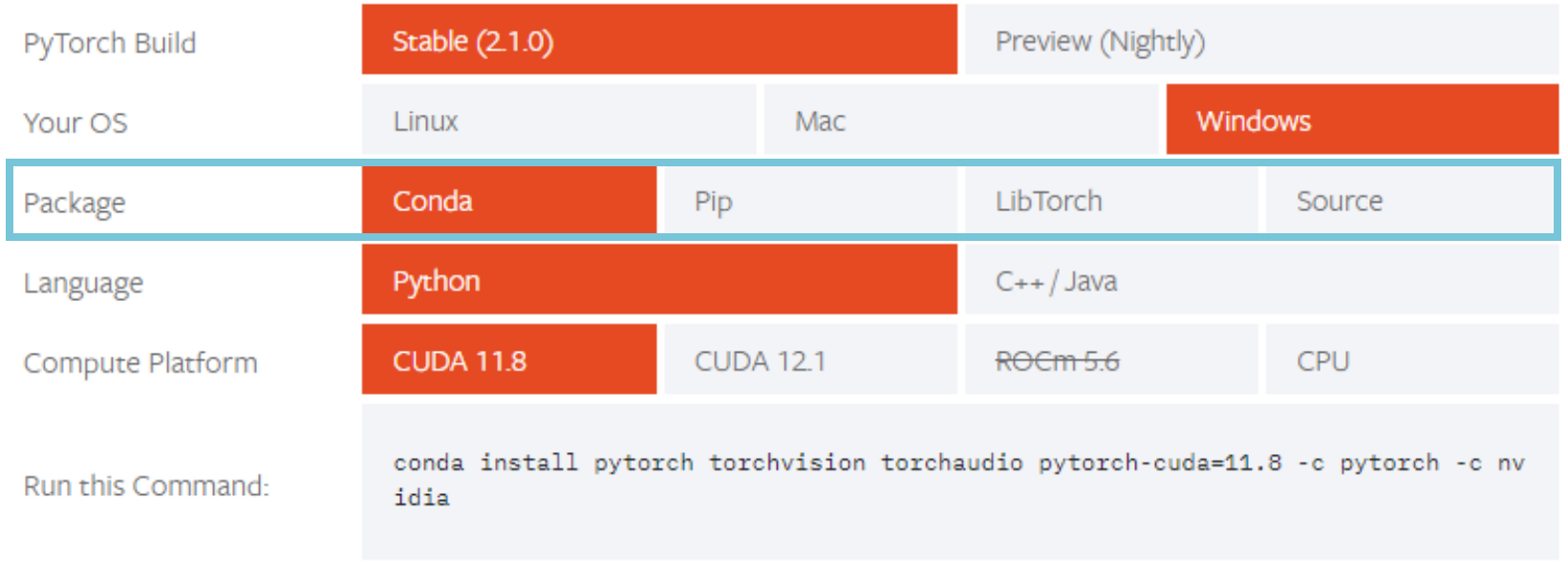
Dependency Management

- Libraries help us develop code faster and safer
  - Reuse existing functionality
  - Rely on thoroughly tested and maintained code
- But: Introduces additional complexity
  - Interfaces, Adapters etc. needed.
  - Everyone needs to use the same library versions in a project
  - in Python?!

- Best practice: Set up an environment (a set of all libraries and their versions) for each project

Python Dependency Management

- Install Python libraries from
  - Pip (package installer for Python)
  - Conda (multiple languages)
  - Source



Python Dependency Management

- Install Python libraries from
  - Pip (package installer for Python)
  - Conda
  - Source
- Environments
  - Machine environment (you don't want to go down this road)
    - Needs root access
  - User environment (neither this one)
    - `pip install --user torch`
  - Virtual environment (does the job, but not always – Win, Mac, Lin specific)
    - virtualenv/venv, pipenv
    - Conda (Anaconda/Miniconda)/mamba
    - Poetry
    - uv

Sharing Dependencies

- pip
  - One configuration file per project specifies the environment:
  - requirements.txt
    - Contains library name and version (torch==2.0.1)
    - `pip freeze > requirements.txt` (save)
    - `pip install -r requirements.txt` (recreate)
    - But: only includes libraries installed with pip
- Poetry, uv
  - Same, but with lock file
  - Fixed versions with hashes

Sharing Dependencies

- Conda
  - environment.yml
    - Export environment from conda: `conda env export -f environment.yml`
    - Can recreate the same environment on a different machine: `conda create -f environment.yml`
    - Includes channels from where libraries were downloaded, and dependencies (pip and non-pip)
    - How to manage environments with conda

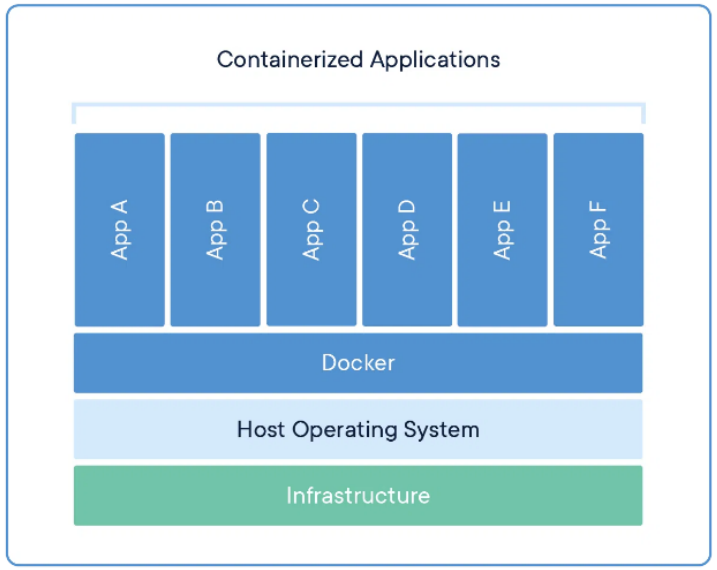
Conda Demo

Containerization

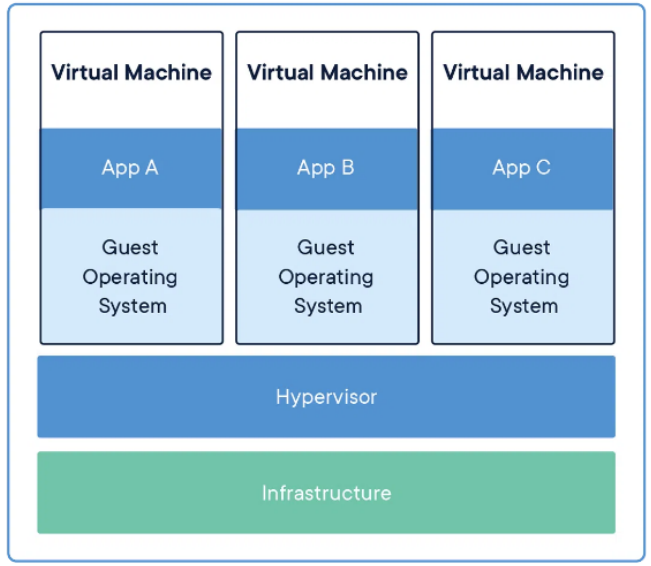
Docker

- Isolates the application from the hardware/OS
- Command-line interface (CLI) based
  - Good for automation/scripts
  - GUI: Docker Desktop
- More lightweight than virtual machines
  - Does not include the OS

Docker



Virtual machine



https://www.docker.com/resources/what-container/

Docker

- Container
  - Sandboxed process, isolated from all other processes on machine
- Image
  - Package of software used by the container
  - Must contain all dependencies, scripts, data

- Docker Hub: Publicly available Docker images
- Docker playground

Dockerfile

- The `Dockerfile` specifies all operations performed when starting the image
- Example:

```
# syntax=docker/dockerfile:1

FROM node:18-alpine
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
EXPOSE 3000
```

where to look for the Dockerfile

- Dockerfile reference
- Build the image with: `docker build -t tag .`

Running a Container

- Run an image with:
  `docker run -d -p 127.0.0.1:3000:3000 tag`
  - -d: detach (run in background)
  - -p: publish (port mapping between host and container)
    - Format is *host:container*
    - Necessary to access the application from the host

- To list all the containers: `docker ps (-a)`

Further Useful Commands

- `docker stop` *`container-id`* (stops running container)
- `docker rm` *`container-id`* (removes stopped container)
- `docker tag` *`old-name new-name`* (rename image)
- `docker push` *`new-name`* (push image to Docker Hub)
- `docker exec` *`container-id command`* (execute a command in a running container)
- `docker volume create volume-name` (mount a volume for persistent storage)
- `docker logs -f` *`container-id`* (see log of a container)

Communicating Between Containers

- Containers are isolated processes
- Can allow them to communicate by placing them on the same network: `docker network create` *`network-name`*
- Connect containers to network at startup or during runtime

Docker Compose

- Define and share multi-container applications
  - Defined in a YAML file

compose.yaml

```yaml
services:
  app:
    image: node:18-alpine
    command: sh -c "yarn install && yarn run dev"
    ports:
      - 127.0.0.1:3000:3000
    working_dir: /app
    volumes:
      - ./:/app
    environment:
      MYSQL_HOST: mysql
      MYSQL_USER: root
      MYSQL_PASSWORD: secret
      MYSQL_DB: todos

  mysql:
    image: mysql:8.0
    volumes:
      - todo-mysql-data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: secret
      MYSQL_DATABASE: todos

volumes:
  todo-mysql-data:
```

- Start the full application with `docker compose up` and stop it with `docker compose down`

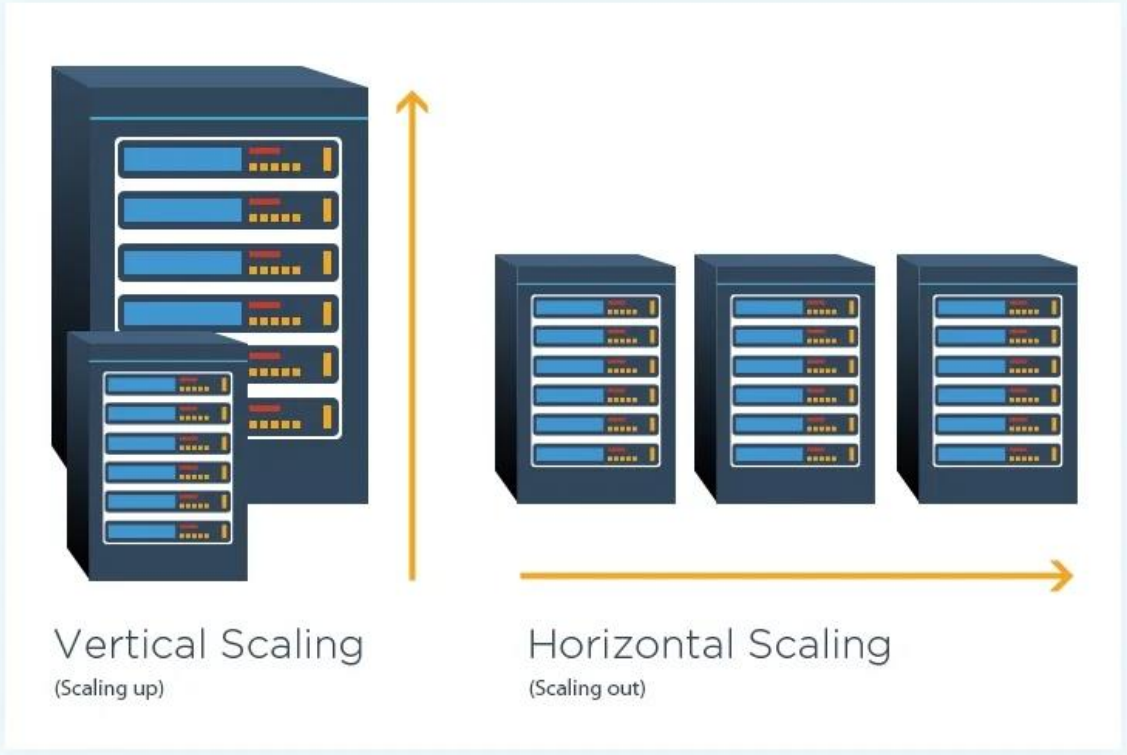
https://docs.docker.com/get-started/08_using_compose/

Managing Containers

Scaling

- How would you scale an application?

Scaling



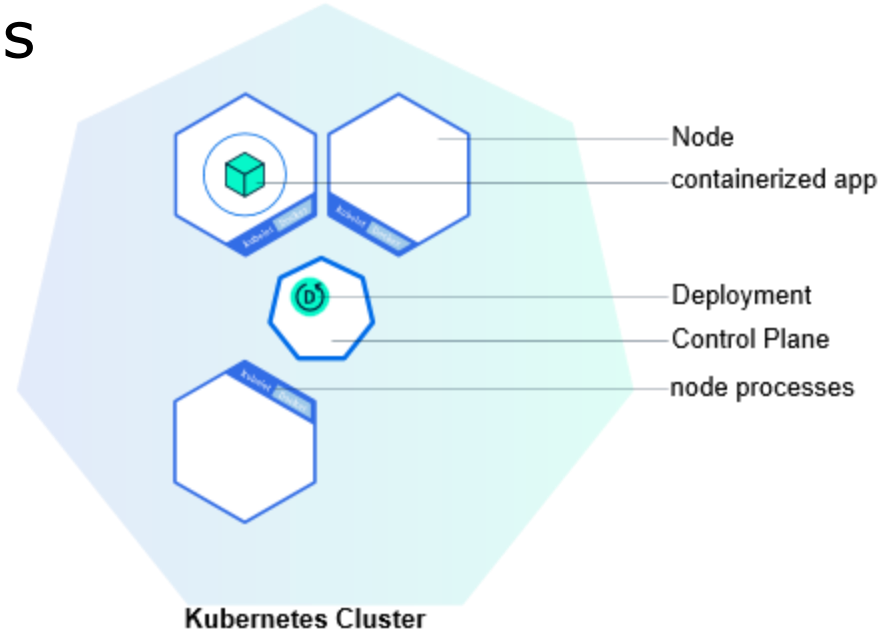
https://www.cloudzero.com/blog/horizontal-vs-vertical-scaling/

Kubernetes

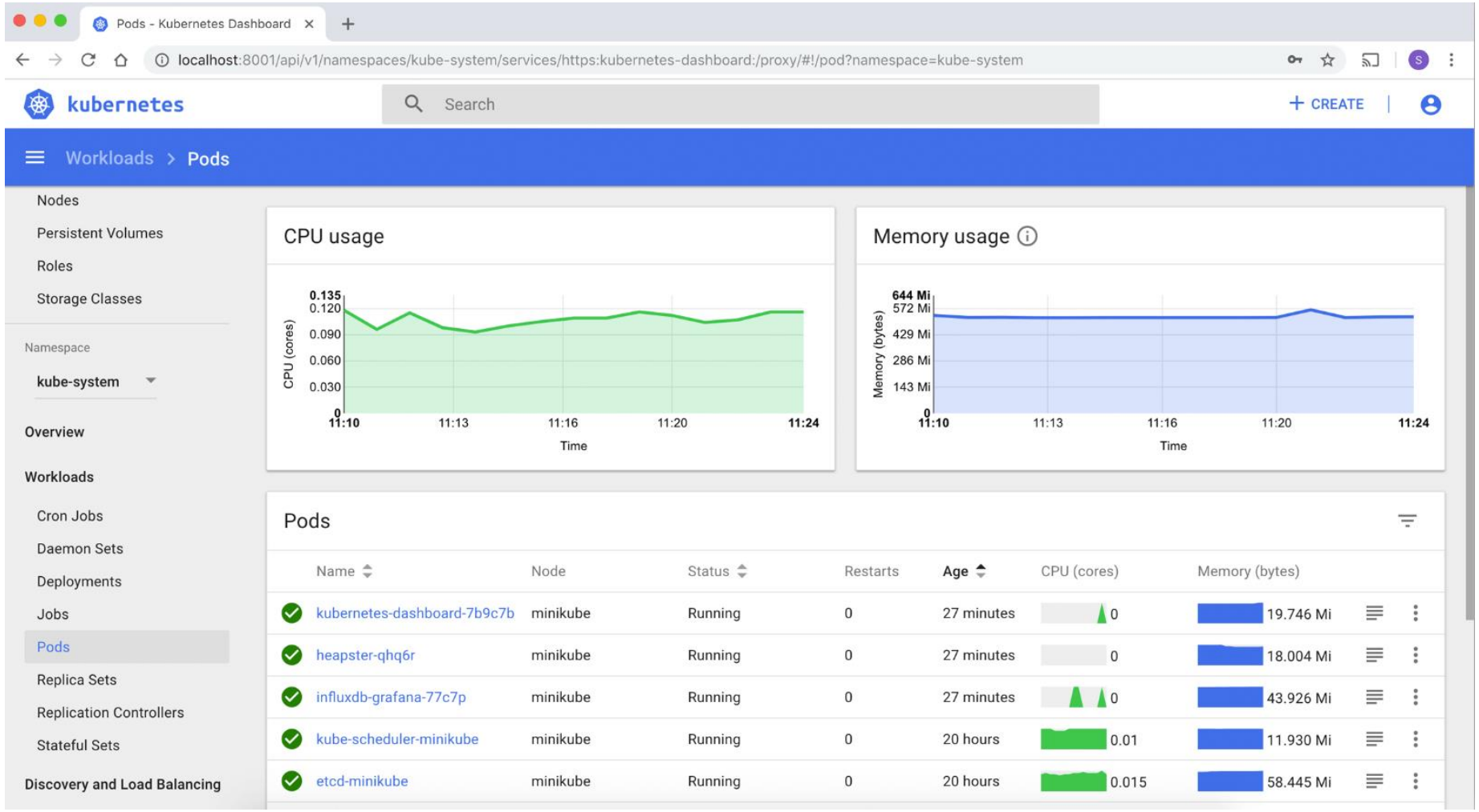
- Also known as K8s (written) or Kate (pronounced)
- Automated deployment, scaling, and management of containerized applications (*orchestration*)
- Runs container groups (*pods*) on worker machines (*nodes*, VM or physical)
- Features
  - Replacing applications on failed nodes
  - Manage storage and networking for pods
  - Load balancing, scaling
  - …
- Playground

Managing a Kubernetes Cluster

- Manage cluster from the *control plane*
  - Command line or dashboard interfaces
- *Deployment* manages application instances
  - If a node goes down (failure or maintenance), the deployment starts an instance on another node



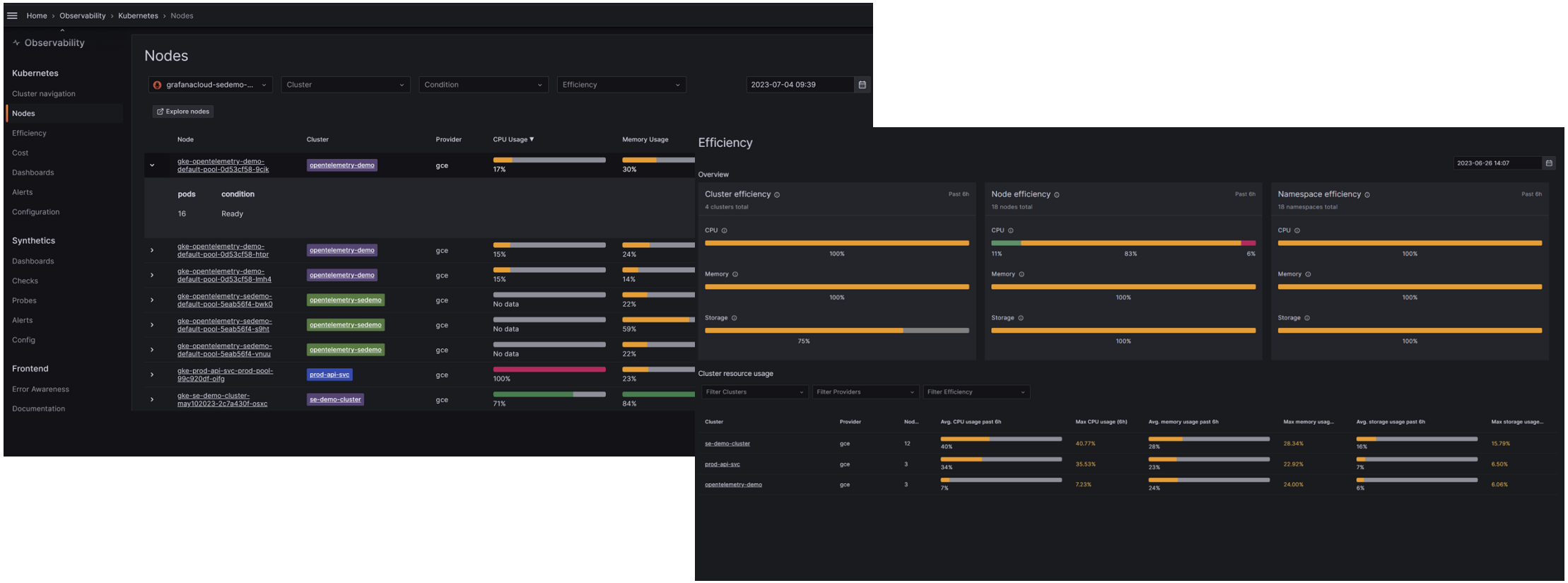
Dashboard



https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/

Interoperable with Tooling

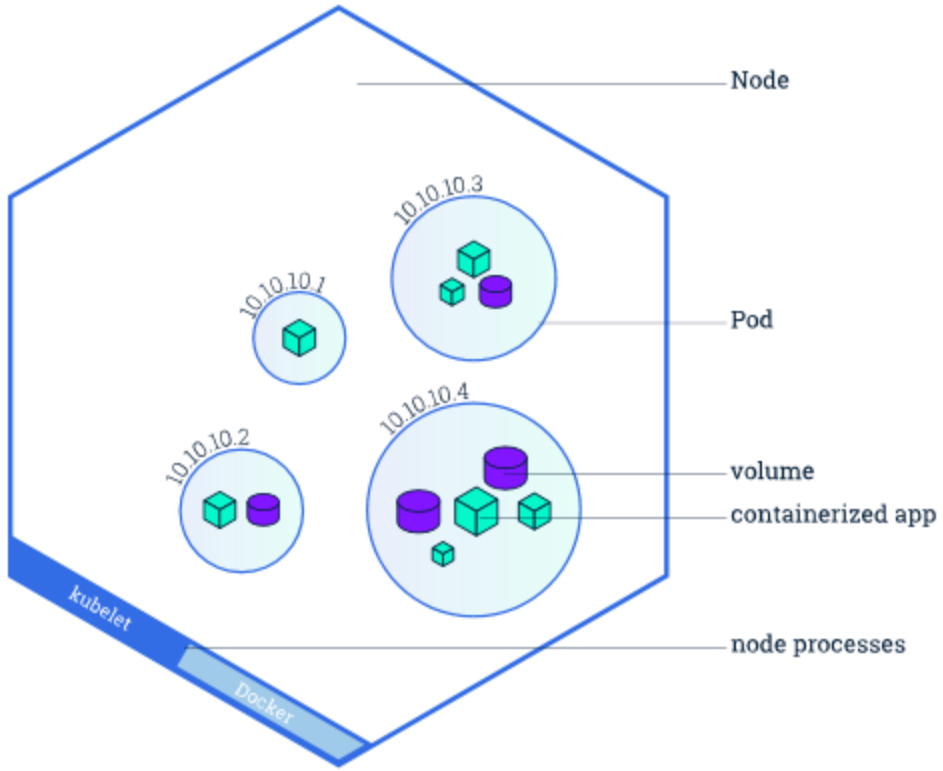
Visualize in Grafana (dashboard and visualization tool)



https://grafana.com/solutions/kubernetes

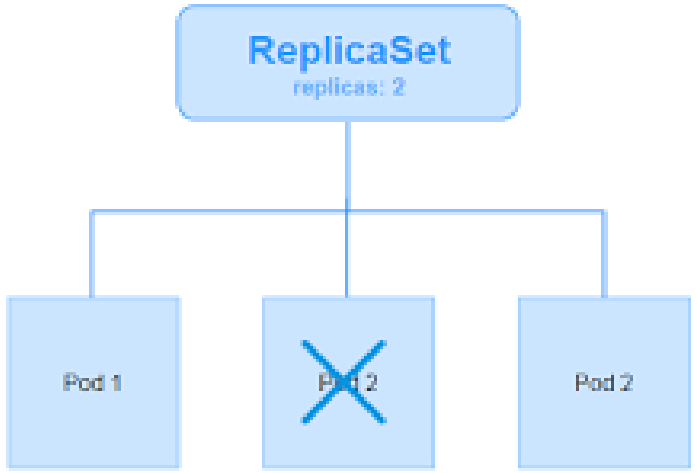
Pods

- Groups of containers that are tightly coupled
- Shared resources
  - Storage (volumes/disk)
  - Networking (will have the same IP)
- If shared resources not necessary, can assign containers to separate pods
  - E.g. frontend, backend, database



ReplicaSet

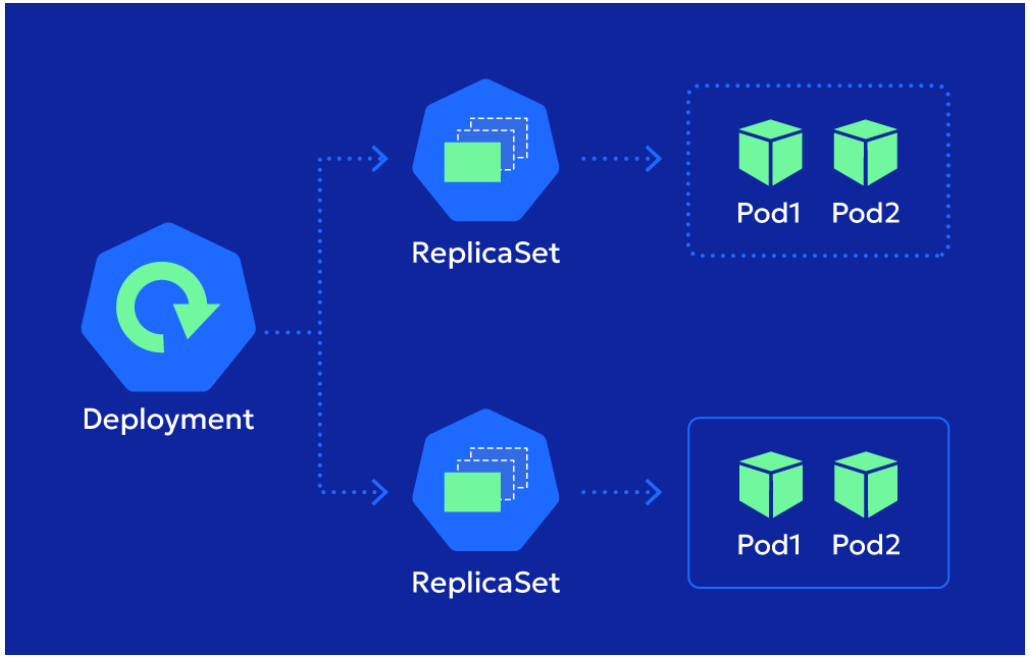
- Desired numbers of pods
- Uses label to manage them
- Does not support rolling updates and rollbacks



https://www.fosstechnix.com/kubernetes-replication-controllers-and-replica-sets/

Deployment

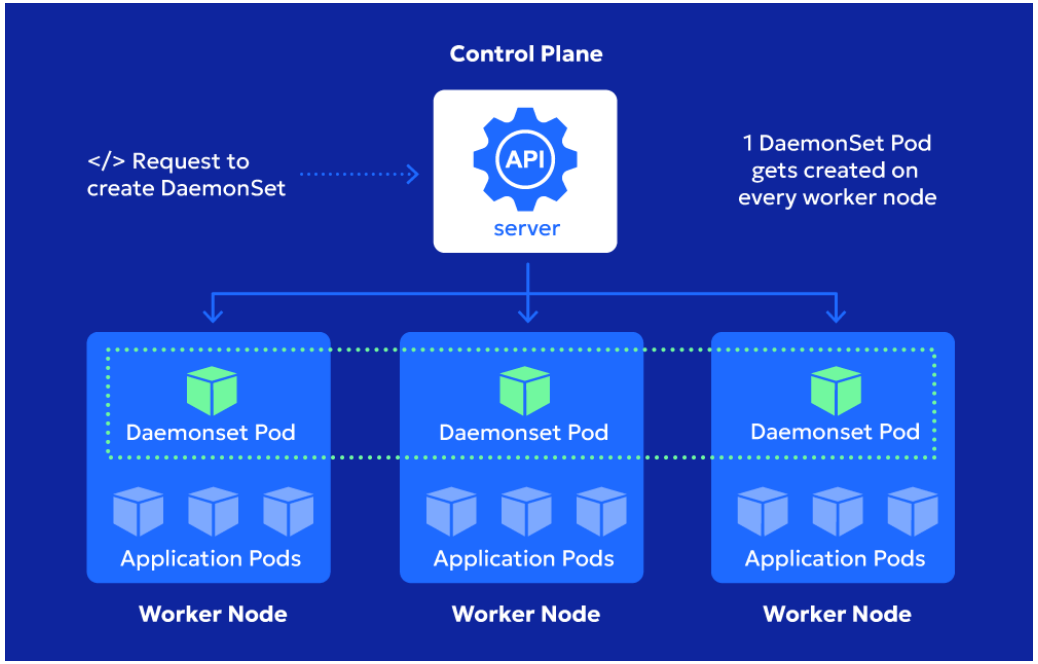
- high-level, declarative object used to manage **stateless** applications
- Scaling
- Rollbacks
- Rolling updates



https://zesty.co/finops-glossary/what-is-a-kubernetes-deployment/

DeamonSet

- Runs on **all nodes**
- Node-level functionality
- Networking
- Log collection (e.g. for Grafana)



https://zesty.co/finops-glossary/daemonsets/

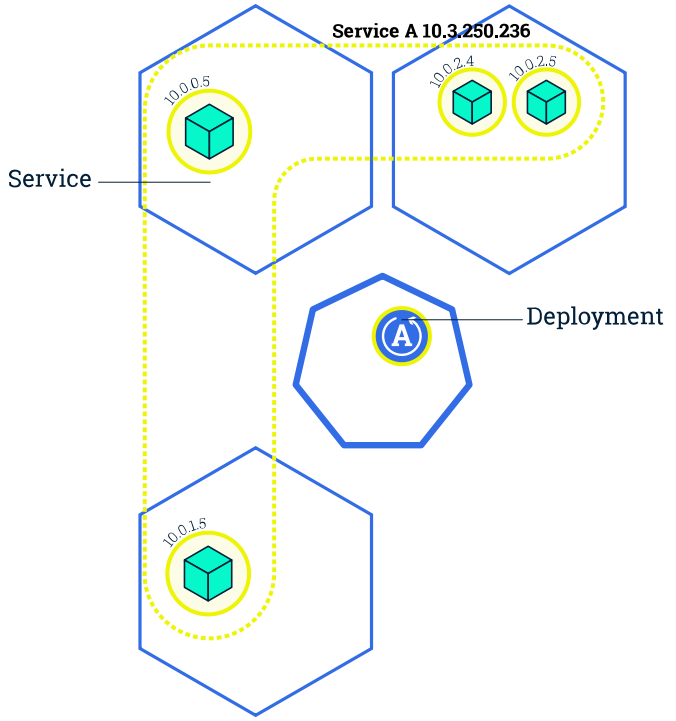
StatefulSet

- ensures each Pod has a **consistent name, hostname, and volume across restarts**
- Removes flexibility, adds consistency
- E.g. for databases (or other things that store data consistently)

- Forget these again, you will notice when you need one ;)
- Question: How to store documents in an application?

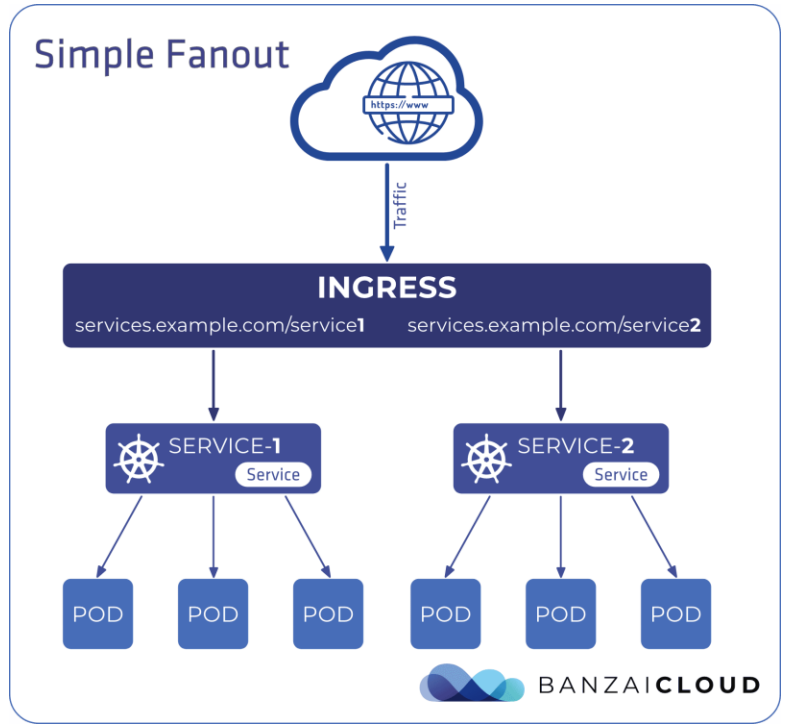
Service

- A *service* exposes pods into the same network
  - Can be on different nodes
  - Enables communication
  - Can add automatic load balancing (requests are distributed to different pods)



Ingress

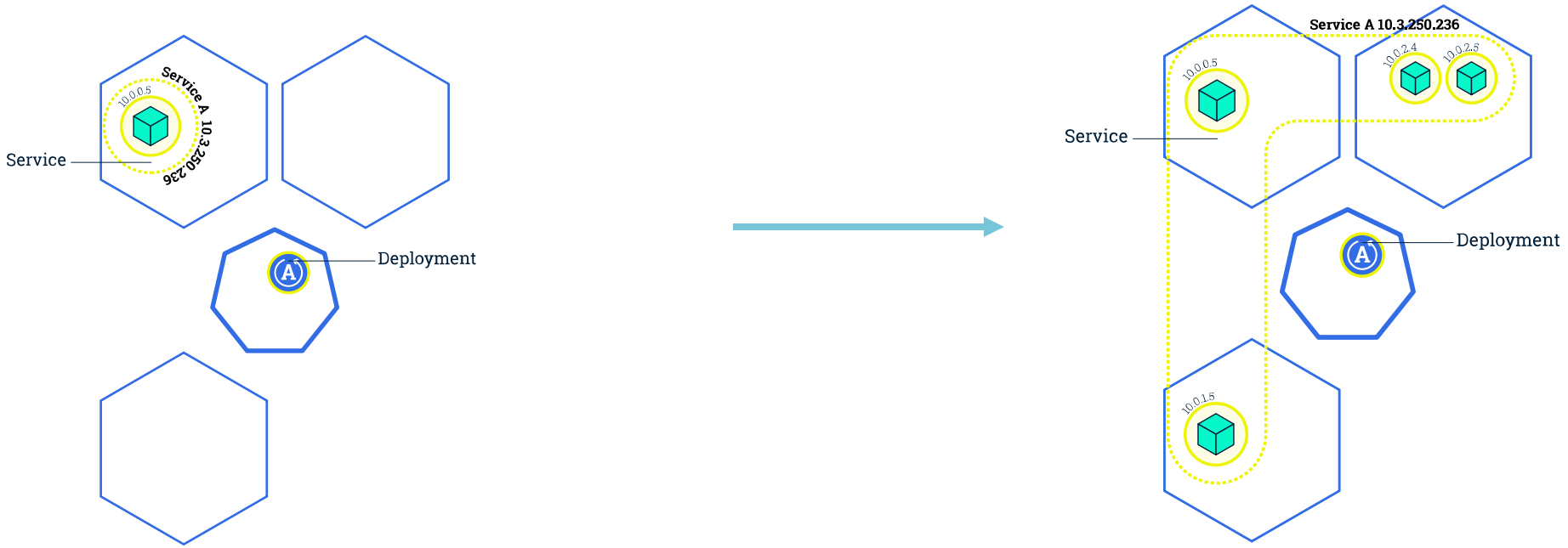
- An ingress is the exposed network interface for applications
- Binds to a service



https://outshift.cisco.com/blog/k8s-ingress

Scaling

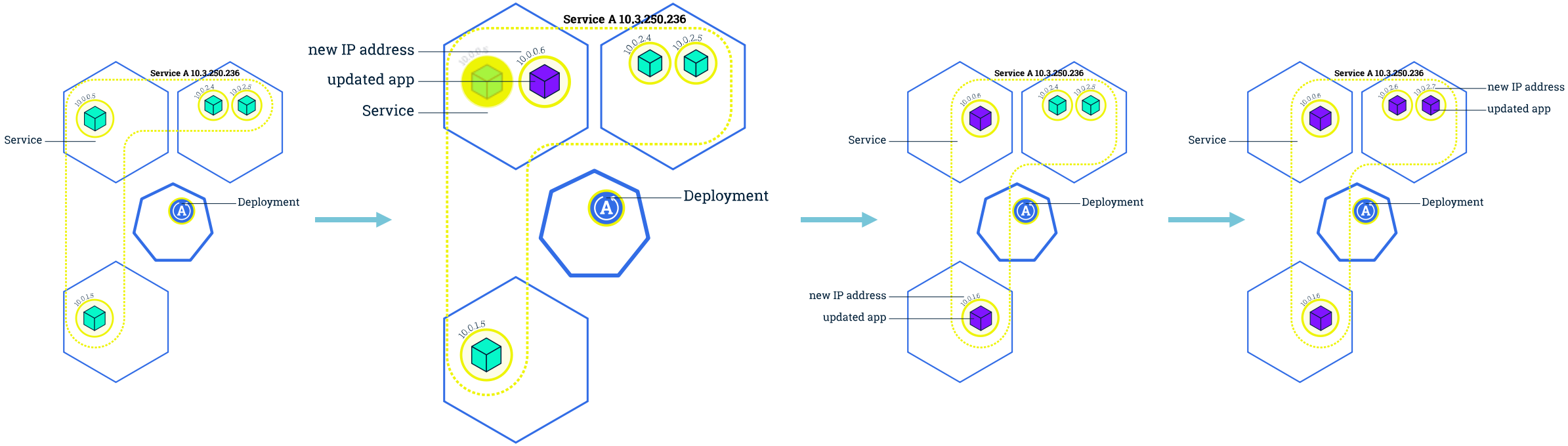
- Define number of replicas (replicaset) when starting a deployment
- Scaling: Change number of replicas (up and down)



https://kubernetes.io/docs/tutorials/kubernetes-basics/scale/scale-intro/

Rolling Updates

- K8s updates (or rolls back) deployment with zero downtime
  - Incrementally updates pod instances with new version
  - Service load balances to available pods during update



https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/

Example

- https://gitlab.switch.ch/hslu/research/abiz/hybrid_creativity/prompt_analysis_text_app/-/tree/main/deployment?ref_type=heads

HELM

- Imagine you have multiple complex applications to combine
- Supports templating

Building Pipelines

Building Pipelines

- Execute all stages of an ML system
  - Preprocessing > feature extraction > training > validation > deployment
- Automatic pipelines/workflows help by
  - running only the necessary stages
  - automatic execution speeds up development
  - standardized processing steps avoid differences between developers/machines
- Workflow typically defined as a directed graph
  - Order of stages
  - If there are no cycles: directed acyclic graph (DAG)
- All tools have dashboards with visualizations

Apache Airflow

- First data science workflow management tool
- Configuration as code
  - Can interact with it from rest of codebase
  - Versioning with Git
- Disadvantages
  - Entire workflow is a single container
  - Designed for batch workloads, scheduled executions
    - Not for constant execution and streaming data

Apache Airflow

- DAG shows the workflow
- Tasks are operators
- Set dependencies between tasks

```python
from datetime import datetime

from airflow import DAG
from airflow.decorators import task
from airflow.operators.bash import BashOperator

# A DAG represents a workflow, a collection of tasks
with DAG(dag_id="demo", start_date=datetime(2022, 1, 1), schedule="0 0 * * *") as dag:

    # Tasks are represented as operators
    hello = BashOperator(task_id="hello", bash_command="echo hello")

    @task()
    def airflow():
        print("airflow")

    # Set dependencies between tasks
    hello >> airflow()
```

```python
dag = DAG(
    'docker_sample',
    default_args={
        'owner': 'airflow',
        'depends_on_past': False,
        'email': ['airflow@example.com'],
        'email_on_failure': False,
        'email_on_retry': False,
        'retries': 1,
        'retry_delay': timedelta(minutes=5),
    },
    schedule_interval=timedelta(minutes=10),
    start_date=days_ago(2),
)

t1 = BashOperator(task_id='print_date', bash_command='date', dag=dag)

t2 = BashOperator(task_id='sleep', bash_command='sleep 5', retries=3, dag=dag)

t3 = DockerOperator(
    api_version='1.19',
    docker_url='tcp://localhost:2375',  # Set your docker URL
    command='/bin/sleep 30',
    image='centos:latest',
    network_mode='bridge',
    task_id='docker_op_tester',
    dag=dag,
)


t4 = BashOperator(task_id='print_hello', bash_command='echo "hello world!!!"', dag=dag)


t1 >> t2
t1 >> t3
t3 >> t4
```


Airflow website and CS329s lecture slides

Kubeflow

- Containerized, parameterized, dynamic DAG
- Built for Kubernetes
- Same code in development and production

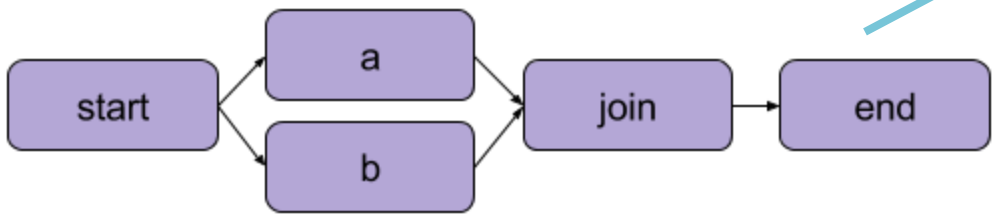
```yaml
name: xgboost4j - Train classifier
description: Trains a boosted tree ensemble classifier using xgboost4j

inputs:
- {name: Training data}
- {name: Rounds, type: Integer, default: '30', description: 'Number of training rounds'}

outputs:
- {name: Trained model, type: XGBoost model, description: 'Trained XGBoost model'}

implementation:
  container:
    image: gcr.io/ml-pipeline/xgboost-classifier-train@sha256:b3a64d57
    command: [
      /ml/train.py,
      --train-set, {inputPath: Training data},
      --rounds,    {inputValue: Rounds},
      --out-model, {outputPath: Trained model},
    ]
```

```python
@func_to_container_op
def produce_two_small_outputs() -> NamedTuple('Outputs', [('text', str), ('number', int)]):
    return ("data 1", 42)

@func_to_container_op
def consume_two_arguments(text: str, number: int):
    print('Text={}'.format(text))
    print('Number={}'.format(str(number)))

def producers_to_consumers_pipeline(text: str = "Hello world"):
    '''Pipeline that passes data from producer to consumer'''
    produce1_task = produce_one_small_output()
    produce2_task = produce_two_small_outputs()

    consume_task1 = consume_two_arguments(produce1_task.output, 42)
    consume_task2 = consume_two_arguments(text, produce2_task.outputs['number'])
    consume_task3 = consume_two_arguments(produce2_task.outputs['text'], produce2_task.outputs['number'])


kfp.Client(host=kfp_endpoint).create_run_from_pipeline_func(producers_to_consumers_pipeline, arguments={})
```


Spec file and workflow (different examples)

Metaflow

- Built by Netflix, open-source
- Same capabilities as Kubeflow, but less boilerplate




```python
from metaflow import FlowSpec, step

class BranchFlow(FlowSpec):

    @step
    def start(self):
        self.next(self.a, self.b)

    @step
    def a(self):
        self.x = 1
        self.next(self.join)

    @step
    def b(self):
        self.x = 2
        self.next(self.join)

    @step
    def join(self, inputs):
        print('a is %s' % inputs.a.x)
        print('b is %s' % inputs.b.x)
        print('total is %d' % sum(input.x for input in inputs))
        self.next(self.end)

    @step
    def end(self):
        pass

if __name__ == '__main__':
    BranchFlow()
```

TFX (TensorFlow Extended)

- ML platform based on TensorFlow
- Includes building pipelines

Pipeline definition

```
def _create_pipeline(pipeline_name: str, pipeline_root: str, data_root: str,
                     module_file: str, serving_model_dir: str,
                     metadata_path: str) -> tfx.dsl.Pipeline:
  """Creates a three component penguin pipeline with TFX."""
  # Brings data into the pipeline.
  example_gen = tfx.components.CsvExampleGen(input_base=data_root)

  # Uses user-provided Python function that trains a model.
  trainer = tfx.components.Trainer(
      module_file=module_file,
      examples=example_gen.outputs['examples'],
      train_args=tfx.proto.TrainArgs(num_steps=100),
      eval_args=tfx.proto.EvalArgs(num_steps=5))

  # Pushes the model to a filesystem destination.
  pusher = tfx.components.Pusher(
      model=trainer.outputs['model'],
      push_destination=tfx.proto.PushDestination(
          filesystem=tfx.proto.PushDestination.Filesystem(
              base_directory=serving_model_dir)))

  # Following three components will be included in the pipeline.
  components = [
      example_gen,
      trainer,
      pusher,
  ]

  return tfx.dsl.Pipeline(
      pipeline_name=pipeline_name,
      pipeline_root=pipeline_root,
      metadata_connection_config=tfx.orchestration.metadata
      .sqlite_metadata_connection_config(metadata_path),
      components=components)
```

Run pipeline

```
tfx.orchestration.LocalDagRunner().run(
  _create_pipeline(
      pipeline_name=PIPELINE_NAME,
      pipeline_root=PIPELINE_ROOT,
      data_root=DATA_ROOT,
      module_file=_trainer_module_file,
      serving_model_dir=SERVING_MODEL_DIR,
      metadata_path=METADATA_PATH))
```


https://www.tensorflow.org/tfx/tutorials/tfx/penguin_simple