Docker Compose (`docker-compose.yml`)

Definition Docker Compose is a tool for defining and running multi-container Docker applications via a single YAML file that configures services, networks, and volumes ([Docker Documentation][1]).

Key Interview Points

- The top-level `version` specifies the Compose file format (v2, v3, or the Compose Specification) ([Docker Documentation][2]).
- Under `services`, each key is a service name; its value defines either an `image` or a `build` context, along with `ports`, `environment`, and `depends_on` ([Docker Documentation][3]).
- `networks` allow services to communicate on custom bridges; `volumes` define persistent storage shared across services ([Docker Documentation][4]).

Mnemonic

> ***“Very Silly Penguins Wiggle”***
>
> - ***V****ersion*
> - ***S****ervices*
> - ***P****orts/Parameters*
> - ***N****etworks*
> - ***V****olumes*

Story Imagine a penguin colony throwing an ice-party: they pick the **version** of their music playlist, set up **services** (hot cocoa stand, ice slide), map **ports** (ice tunnels) and **parameters** (guest list), string **networks** of glowing lights, and stash leftovers in giant **volumes** of ice.

Interview Tip

> *“In `docker-compose.yml`, I always begin with `version`, define each service under `services`, map its `ports` and environment variables, then configure any custom `networks`, and finally declare `volumes` for data persistence.”*

Example (`docker-compose.yml`)

```yaml
version: '3.8'

services:
  web:
    image: nginx:stable
    ports:
      - "80:80"
    environment:
      - NGINX_HOST=example.com
      - NGINX_PORT=80
  api:
    build:
      context: ./api
      dockerfile: Dockerfile
    ports:
      - "4000:4000"
    depends_on:
      - db

  db:
    image: postgres:13
    restart: always
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: pass
    volumes:
      - db-data:/var/lib/postgresql/data

networks:
  default:
    driver: bridge

volumes:
  db-data:
```

GitHub Actions Workflow (`.github/workflows/ci.yml`)

Definition GitHub Actions workflows automate CI/CD by running jobs in response to repository events, defined in YAML under `.github/workflows/` ([GitHub Docs][5]).

Key Interview Points

- `name` : a human-readable identifier for the workflow.
- `on` : specifies triggers such as `push` , `pull_request` , or a cron `schedule` .
- `jobs` : each job runs in isolation; uses `runs-on` to pick a runner (e.g., `ubuntu-latest`), and contains ordered `steps` (actions or shell commands) ([GitHub Docs][6]).

Mnemonic

> ***“Naughty Otters Jump Right Swiftly”***
>
> - ***N**ame*
> - ***O**n*
> - ***J**obs*
> - ***R**uns-on*
> - ***S**teps*

Story Picture a troupe of mischievous otters putting on a river show: they give the performance a **name**, decide **on** which days to perform, assign **jobs** like juggling and diving, choose the rafts they’ll **run-on**, and rehearse the **steps** of their synchronized splash finale.

Interview Tip

> *“In my GitHub Actions YAML, I define `name`, set the `on` triggers, declare `jobs` with specific `runs-on` environments, and list each `steps` block—often starting with checkout, setup, build, test, and deploy actions.”*

Example (`ci.yml`)

```
name: CI Pipeline

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

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

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '18'

      - name: Install dependencies
        run: npm ci

      - name: Run tests
        run: npm test

      - name: Build Docker image
        run: docker build -t myapp:${{ github.sha }} .

      - name: Push to AWS ECR
        env:
          AWS_REGION: us-east-1
        run: |
          aws ecr get-login-password --region $AWS_REGION \
            | docker login --username AWS --password-stdin
123456789012.dkr.ecr.$AWS_REGION.amazonaws.com
          docker tag myapp:${{ github.sha }}
123456789012.dkr.ecr.$AWS_REGION.amazonaws.com/myapp:latest
          docker push 123456789012.dkr.ecr.$AWS_REGION.amazonaws.com/myapp:latest
```

Jenkins Pipeline (`Jenkinsfile`)

Definition Pipeline-as-code for Jenkins, defining your build/test/deploy workflow in a Groovy DSL.

Key Interview Points

- Top-level `pipeline {}` block encloses everything.
- `agent` declares where the pipeline runs (e.g. `any`, `label 'docker'`).
- `stages` groups named `stage('Build') { steps { … } }`, etc.
- Within each `steps`, you run shell commands or use plugins.
- You can parallelize stages and manage credentials via `environment` and `credentials`.

Mnemonic

> ***“Purple Ants Stroll Softly”***
>
> - ***P**ipeline → **A**gent → **S**tages → **S**tage → **S**teps*

Story Purple ants form a construction crew: they lay the pipeline trunk, each ant (agent) grabs materials, map out stages of the build, work through each stage’s steps, then celebrate when the bridge is complete.

Interview Tip

> *“My `Jenkinsfile` begins with `pipeline { agent any }`, defines `stages` for Build, Test, and Deploy, and uses shared library steps for environment setup and artifact archiving.”*

Example (`Jenkinsfile`)

```
pipeline {
  agent any
  environment {
    REGISTRY = '123456789012.dkr.ecr.us-east-1.amazonaws.com'
    IMAGE    = "${REGISTRY}/myapp"
  }
  stages {
    stage('Checkout') {
      steps { checkout scm }
    }
    stage('Build & Test') {
      steps {
        sh 'npm ci'
        sh 'npm test'
      }
    }
    stage('Docker Build & Push') {
      steps {
        sh """
          $(aws ecr get-login --no-include-email --region us-east-1)
          docker build -t $IMAGE:${env.BUILD_NUMBER} .
          docker push $IMAGE:${env.BUILD_NUMBER}
        """
      }
    }
    stage('Deploy') {
      steps {
        sh 'kubectl apply -f k8s/deployment.yaml'
        sh 'kubectl rollout status deployment/myapp'
      }
    }
  }
}
```

Terraform Module (`main.tf`)

Definition Terraform is a declarative IaC tool that provisions cloud infrastructure via `.tf` files and a state backend.

Key Interview Points

- `terraform init` bootstraps providers.
- `terraform plan` previews changes.
- `terraform apply` executes.
- `terraform destroy` tears down resources.
- Use remote state (S3 + DynamoDB) for team collaboration.

Mnemonic

> ***“Pandas Really Value Oranges”***
>
> - ***P**rovider → **R**esource → **V**ariable → **O**utput*

Story Pandas run a jungle café: they choose their bamboo crate **provider**, list each fruit **resource**, set pricing **variables**, and update the earnings **outputs** on their chalkboard.

Interview Tip

> *“In Terraform I start with a `provider` block, declare `resource` blocks for each AWS component, parameterize settings with `variable`, and expose useful IDs via `output`.”*

Example (`main.tf`)

```
terraform {
  required_providers {
    aws = { source = "hashicorp/aws", version = "~> 4.0" }
  }
  backend "s3" {
    bucket = "my-terraform-state"
    key    = "eks/terraform.tfstate"
    region = "us-east-1"
  }
}

provider "aws" {
  region = "us-east-1"
}

variable "app_name" {
  type    = string
  default = "myapp"
}

resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
  tags       = { Name = "${var.app_name}-vpc" }
}

output "vpc_id" {
  value = aws_vpc.main.id
}
```

Ansible Playbook (`site.yml`)

Definition YAML-based automation for configuration management and app deployment, agentless over SSH.

Key Interview Points

- Top-level list of plays with `hosts`.
- `become: yes` for privilege escalation.
- `vars` for reusable parameters.
- `tasks` list actions (modules).
- `handlers` run on notification (e.g., service restarts).

Mnemonic

> ***“Hungry Bears Value Tasty Honey”***
>
> - ***H****osts* → ***B****ecome* → ***V****ars* → ***T****asks* → ***H****andlers*

Story Hungry bears pick their berry bush **hosts**, choose to **become** stronger, measure berry **vars**, perform picking **tasks**, and call **handlers** (the bees) if the honey jars break.

Interview Tip

> *“My playbooks start with `hosts: all`, optionally `become: yes`, define `vars`, list `tasks` like package installs and file templates, and include `handlers` for service reloads.”*

Example (`site.yml`)

```yaml
- hosts: webservers
  become: yes
  vars:
    app_repo: https://github.com/example/myapp.git
    app_dest: /var/www/myapp

  tasks:
    - name: Install NGINX
      apt:
        name: nginx
        state: latest
        update_cache: yes
    - name: Checkout application code
      git:
        repo: "{{ app_repo }}"
        dest: "{{ app_dest }}"
        version: main

    - name: Configure NGINX
      template:
        src: nginx.conf.j2
        dest: /etc/nginx/sites-available/myapp
      notify:
        - reload nginx

  handlers:
    - name: reload nginx
      service:
        name: nginx
        state: reloaded
```

Kubernetes Deployment (`deployment.yaml`)

Definition A Deployment resource manages stateless pods with rolling updates and self-healing.

Key Interview Points

- `apiVersion: apps/v1` & `kind: Deployment`.
- `metadata.name` and labels.
- `spec.replicas`, `selector.matchLabels`.
- `template` block contains pod spec (containers, ports, env).
- Enables rolling updates and rollbacks out of the box.

Mnemonic

> ***“Ancient Kings Make Simple Temples”***
>
> - *__A__PI Version → __K__ind → __M__etadata → __S__pec → __T__emplate*

Story Ancient kings declare the **apiVersion**, choose the **kind** of structure, engrave **metadata**, outline the **spec**, and lay the **template** foundation blocks to ensure the temple always remains intact.

Interview Tip

> *“In my Deployment YAML I use `apiVersion` and `kind`, set `metadata` labels, define the desired `replicas` and `selector`, and under `template` configure containers, ports, and liveness probes.”*

Example (`deployment.yaml`)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
  labels:
    app: myapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: myapp
          image: 123456789012.dkr.ecr.us-east-1.amazonaws.com/myapp:latest
          ports:
            - containerPort: 80
          env:
            - name: NODE_ENV
              value: production
```

Helm Chart (`Chart.yaml` & `values.yaml`)

Definition Helm is the Kubernetes package manager; charts bundle multiple manifests and parameterize them with values.

Key Interview Points

- `Chart.yaml` contains chart metadata: `apiVersion` , `name` , `version` , `dependencies` .
- `values.yaml` holds default configuration that templates consume.
- Commands: `helm repo add` , `helm install` , `helm upgrade` , `helm rollback` .

Mnemonic

> ***"All Noble Vampires Drink Vanilla"***
>
> - ***A**PI Version → **N**ame → **V**ersion → **D**ependencies → **V**alues*

Story Noble vampires host a midnight ball: they note the **apiVersion** of their society, announce their **name**, update fang **version**, invite **dependencies** (bat entourage), and sip **vanilla** elixir as they dance.

Interview Tip

> *"In `Chart.yaml` I set `apiVersion` , `name` , `version` , and any `dependencies` . I use `values.yaml` to override container image tags, replica counts, and environment-specific configs."*

Example (`Chart.yaml`)

```
apiVersion: v2
name: myapp
description: A Helm chart for myapp
type: application
version: 0.1.0
```

```
appVersion: "1.0.0"
dependencies:
  - name: redis
    version: 15.0.0
    repository: https://charts.bitnami.com/bitnami
```

Example (`values.yaml`)

```
replicaCount: 2

image:
  repository: 123456789012.dkr.ecr.us-east-1.amazonaws.com/myapp
  tag: latest

service:
  type: LoadBalancer
  port: 80

resources: {}
nodeSelector: {}
tolerations: []
affinity: {}
```

ArgoCD Application (`application.yaml`)

Definition An ArgoCD Application declares a Git repo path to sync into one or more Kubernetes clusters, enabling GitOps.

Key Interview Points

- `apiVersion: argoproj.io/v1alpha1` & `kind: Application` .
- `metadata.name` , `metadata.namespace` .
- Under `spec` : `project` , `source` (repoURL, targetRevision, path), `destination` (server, namespace).
- `syncPolicy` can be `automated` (with `prune` and `selfHeal`) or manual.

Mnemonic

> ***"Always Keep Managing Syncs"***
>
> - ***A****PI Version →* ***K****ind →* ***M****etadata →* ***S****pec →* ***S****yncPolicy*

Story ArgoCD is like a dream chef: you write your recipe (Git), it notes the **apiVersion**, selects the **kind** of dish, labels it in **metadata**, follows the **spec** for ingredients, and uses **syncPolicy** to auto-reheat whenever the menu changes.

Interview Tip

> *"An ArgoCD Application YAML starts with `apiVersion` and `kind` , includes `metadata` , then under `spec` defines the Git `source` , `destination` cluster, and a `syncPolicy` for automated reconciliation."*

Example (`application.yaml`)

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: myapp
  namespace: argocd
spec:
  project: default
  source:
    repoURL: https://github.com/example/myapp
    targetRevision: main
    path: helm-chart
  destination:
    server: https://kubernetes.default.svc
    namespace: production
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

Prometheus Config (`prometheus.yml`)

Definition Prometheus configuration defines how metrics are scraped, which alerting rules to load, and where to send long-term data.

Key Interview Points

- `global` : defaults such as `scrape_interval` and `evaluation_interval` .
- `scrape_configs` : list of jobs, each with `job_name` and service discovery or static targets.
- `rule_files` : paths to recording and alerting rules.
- `alerting` : configuration for Alertmanager endpoints.
- `remote_write` : send data to long-term storage.

Mnemonic

> ***“Giant Sharks Race Around Reefs”***
>
> - ***G**lobal → **S**crape_configs → **R**ule_files → **A**lerting → **R**emote_write*

Story Giant sharks patrol the reef: they set **global** hunting grounds, follow **scrape_configs** for prey locations, memorize **rule_files** for safe zones, fire **alerting** signals when low on energy, and **remote_write** their catch logs to the deep-sea archives.

Interview Tip

> *“In `prometheus.yml` , I configure the `global` scrape interval, define `scrape_configs` for each target, reference `rule_files` for alerts, set up the `alerting` section to point to Alertmanager, and use `remote_write` for long-term storage in Thanos or Cortex.”*

Example (`prometheus.yml`)

```yaml
global:
  scrape_interval: 15s
  evaluation_interval: 30s

scrape_configs:
  - job_name: 'kubernetes-nodes'
    kubernetes_sd_configs:
      - role: node

  - job_name: 'kubernetes-pods'
    kubernetes_sd_configs:
      - role: pod

rule_files:
  - 'alert.rules.yml'

alerting:
  alertmanagers:
    - static_configs:
        - targets:
            - 'alertmanager:9093'

remote_write:
  - url: 'https://long-term-storage.example.com/api/v1/write'
```