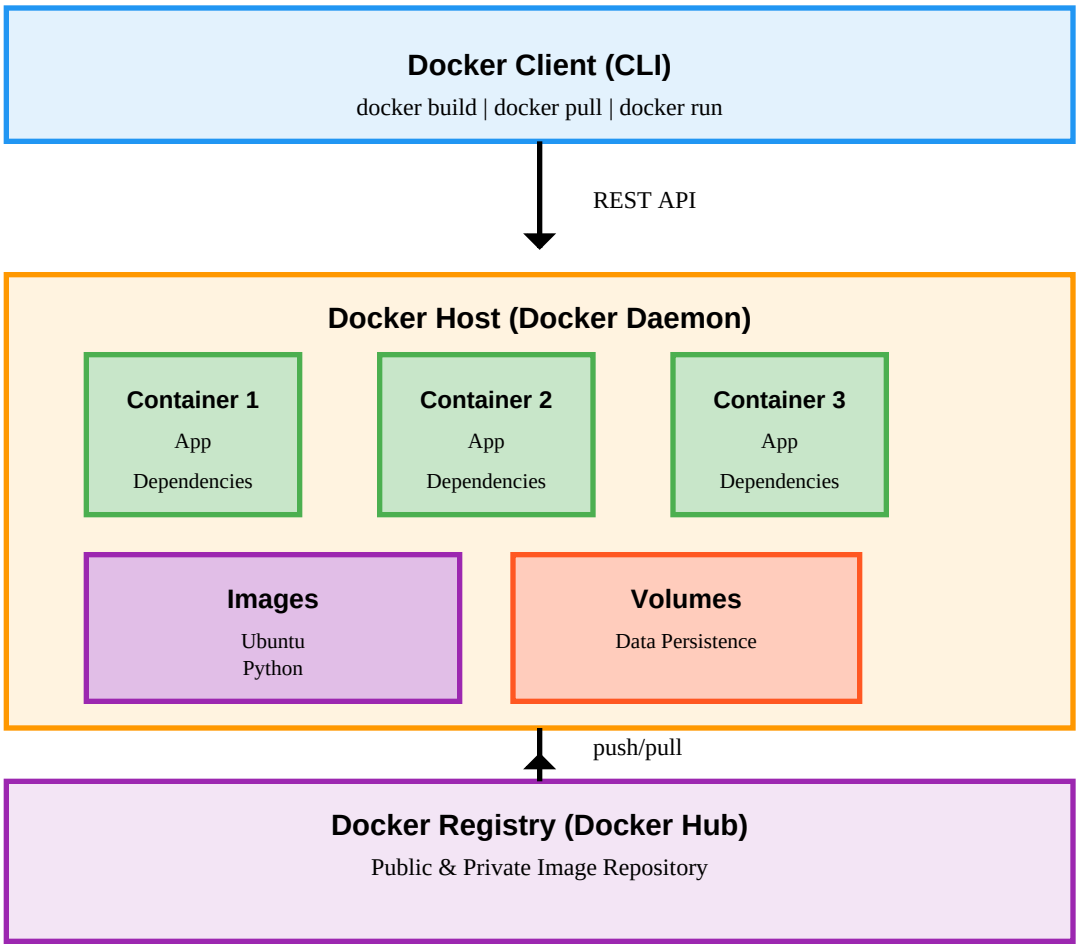
Docker Complete Guide for Data Engineers

Architecture, Deployment, and Best Practices

Docker Architecture Overview

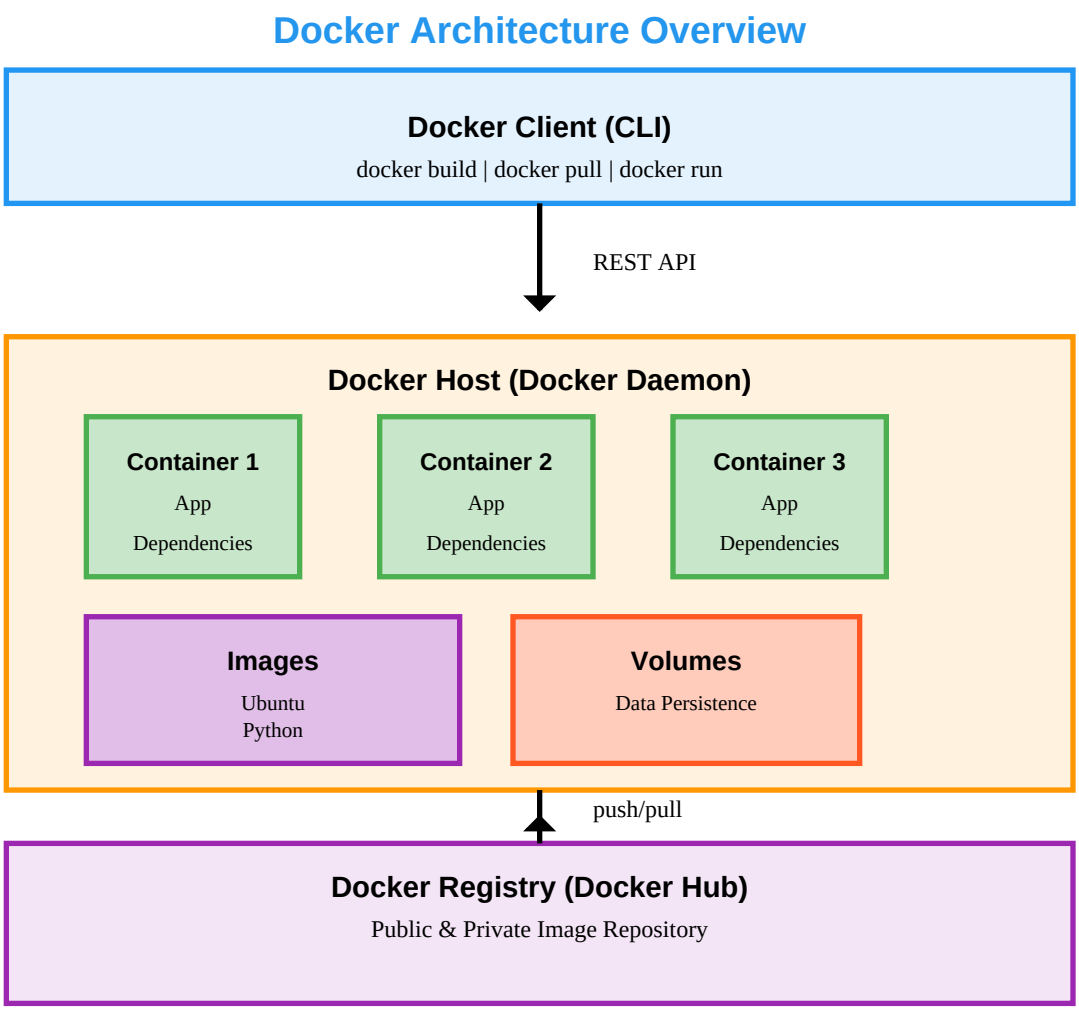


A comprehensive guide covering containerization, orchestration, cloud deployment, and CI/CD pipelines

Table of Contents

1. Docker Architecture Overview

Docker is a platform for developing, shipping, and running applications in containers. Understanding its architecture is crucial for data engineers working with distributed systems.



Key Components:

• **Docker Client:** Command-line interface where you interact with Docker. Commands like 'docker build', 'docker run', and 'docker pull' are executed here.

• **Docker Daemon (dockerd):** Background service that manages Docker objects like images, containers, networks, and volumes. It listens for Docker API requests.

• **Docker Images:** Read-only templates containing application code, runtime, libraries, and dependencies. Images are built from Dockerfiles.

• **Docker Containers:** Runnable instances of images. Containers are isolated from each other and the host system, making them portable across environments.

• **Docker Registry:** Storage and distribution system for Docker images. Docker Hub is the default public registry, but private registries (ECR, GCR, ACR) are common.

• **Docker Volumes:** Persistent data storage mechanism that survives container restarts. Essential for databases and stateful applications.

How Docker Works:

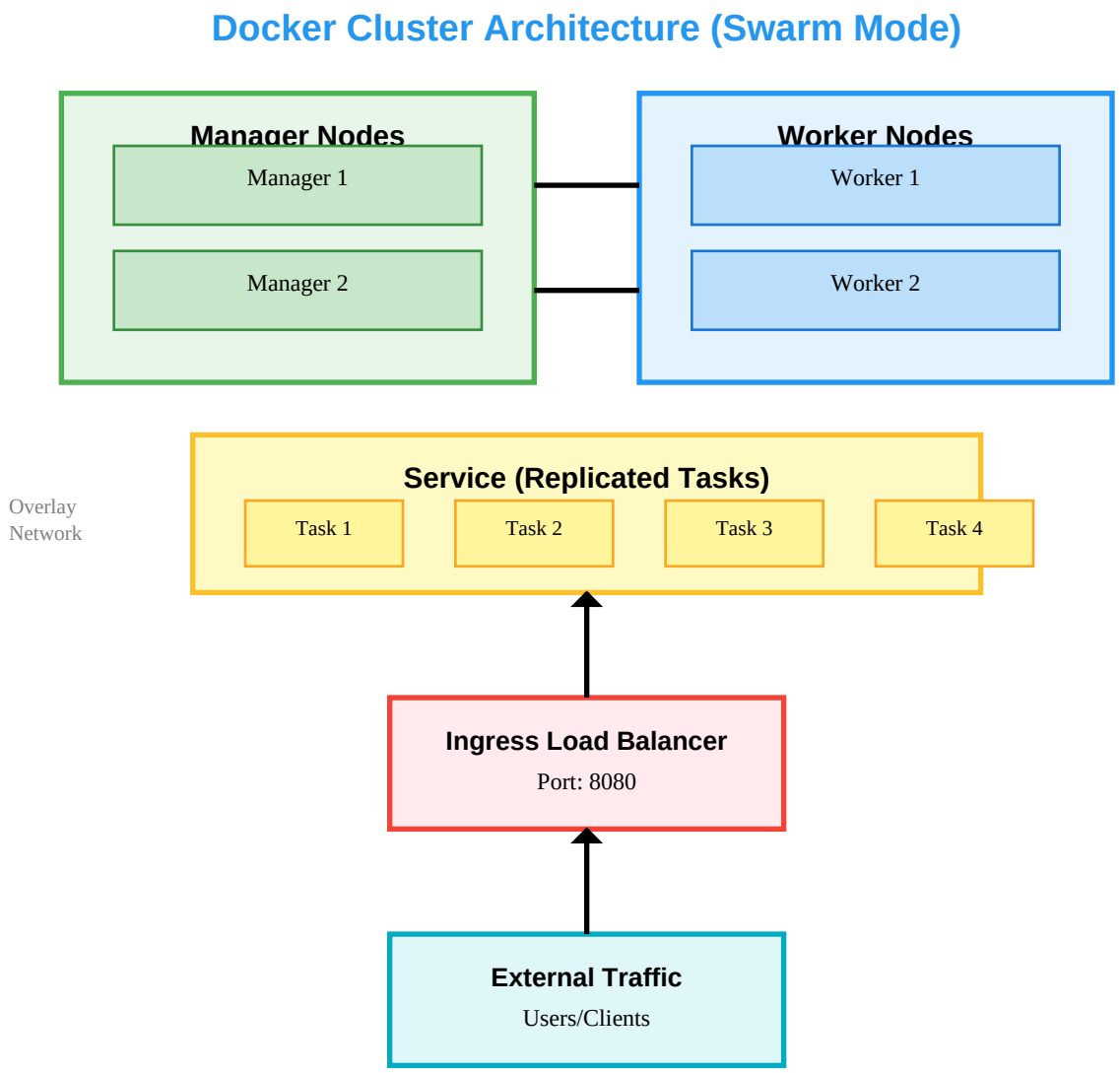
1. Build Phase: Write a Dockerfile with instructions → Execute 'docker build' → Docker daemon reads Dockerfile → Creates image layers → Final image stored locally.

2. Ship Phase: Tag image with repository name → Push to registry using 'docker push' → Image becomes available for deployment anywhere.

3. Run Phase: Pull image from registry → Create container instance → Allocate resources (CPU, memory) → Start application inside isolated environment.

2. Docker Cluster Architecture

For production data engineering workloads, single-node Docker deployments are insufficient. Cluster orchestration provides high availability, load balancing, and automatic scaling across multiple nodes.



2.1 Docker Swarm Mode

Docker Swarm is Docker's native clustering solution. It turns multiple Docker hosts into a single virtual Docker host.

Setting Up Docker Swarm Cluster:

Step 1: Initialize Swarm on Manager Node

```
# On manager node (e.g., 192.168.1.100) docker swarm init --advertise-addr
192.168.1.100 # Output provides join token for workers # Swarm initialized: current
node is now a manager
```

Step 2: Join Worker Nodes

```
# On worker nodes (192.168.1.101, 192.168.1.102, etc.) docker swarm join --token
SWMTKN-1-xxx \ 192.168.1.100:2377
```

Step 3: Verify Cluster

```
# On manager node docker node ls # Output shows all nodes in cluster # ID HOSTNAME
STATUS AVAILABILITY MANAGER STATUS # abc123 manager1 Ready Active Leader # def456
worker1 Ready Active # ghi789 worker2 Ready Active
```

Step 4: Deploy Service Across Cluster

```
# Deploy replicated service docker service create --name spark-app \ --replicas 5 \
--publish 8080:8080 \ myregistry/spark-app:v1 # Docker automatically distributes
containers across nodes docker service ps spark-app # Check task distribution
```

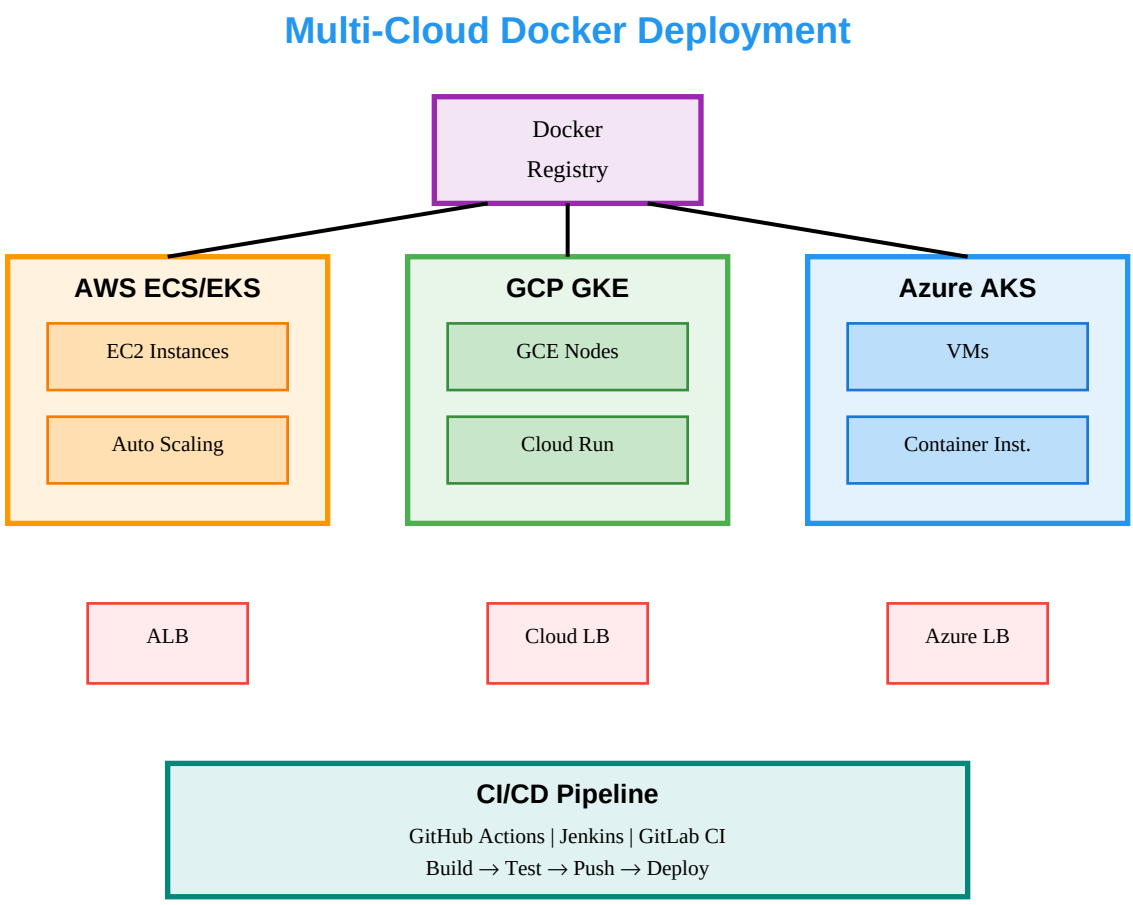
2.2 Kubernetes Alternative

While Docker Swarm is simple, Kubernetes has become the industry standard for production orchestration. Key advantages include richer ecosystem, better scaling, and superior service mesh integration.

Feature	Docker Swarm	Kubernetes
Setup Complexity	Simple	Complex
Load Balancing	Built-in	Requires setup
Auto-scaling	Basic	Advanced (HPA/VPA)
Community	Smaller	Large ecosystem
Best For	Small to medium deployments	Enterprise production

3. Cloud Deployment Guide

Modern data engineering requires deploying Docker containers across multiple cloud providers. Each cloud offers managed container services that simplify orchestration and scaling.



3.1 Google Cloud Platform (GCP)

GCP offers Google Kubernetes Engine (GKE) for managed Kubernetes and Cloud Run for serverless containers.

Deploying to GKE - Step by Step:

```
# Step 1: Install gcloud CLI and authenticate gcloud auth login gcloud config set project
YOUR_PROJECT_ID # Step 2: Create GKE cluster gcloud container clusters create data-cluster \
--zone us-central1-a \ --num-nodes 3 \ --machine-type n1-standard-4 \ --disk-size 100GB # Step
3: Get cluster credentials gcloud container clusters get-credentials data-cluster \ --zone
us-central1-a # Step 4: Push image to Google Container Registry (GCR) docker tag myapp:v1
gcr.io/YOUR_PROJECT_ID/myapp:v1 docker push gcr.io/YOUR_PROJECT_ID/myapp:v1
```

Step 5: Deploy Application

```
# Create Kubernetes deployment kubectl create deployment myapp \
--image=gcr.io/YOUR_PROJECT_ID/myapp:v1 \ --replicas=3 # Expose as load-balanced service
kubectl expose deployment myapp \ --type=LoadBalancer \ --port=80 \ --target-port=8080 # Check
status kubectl get services # Get external IP kubectl get pods # Verify pod status
```

3.2 Amazon Web Services (AWS)

AWS provides Elastic Container Service (ECS) and Elastic Kubernetes Service (EKS) for container orchestration.

Deploying to AWS ECS - Step by Step:

```
# Step 1: Install AWS CLI and configure aws configure # Enter: Access Key ID, Secret Access
Key, Region (us-east-1) # Step 2: Create ECR repository aws ecr create-repository
--repository-name data-app # Step 3: Authenticate Docker to ECR aws ecr get-login-password
--region us-east-1 | \ docker login --username AWS --password-stdin \
123456789.dkr.ecr.us-east-1.amazonaws.com # Step 4: Push image to ECR docker tag myapp:v1 \
123456789.dkr.ecr.us-east-1.amazonaws.com/data-app:v1 docker push \
123456789.dkr.ecr.us-east-1.amazonaws.com/data-app:v1
```

Step 5: Create ECS Cluster and Service

```
# Create ECS cluster aws ecs create-cluster --cluster-name data-cluster # Register task
definition (task-def.json required) aws ecs register-task-definition \ --cli-input-json
file://task-definition.json # Create service with load balancer aws ecs create-service \
--cluster data-cluster \ --service-name data-service \ --task-definition data-task:1 \
--desired-count 3 \ --launch-type FARGATE \ --network-configuration "awsvpcConfiguration={
subnets=[subnet-xxx], securityGroups=[sg-xxx], assignPublicIp=ENABLED }" # Check service
status aws ecs describe-services --cluster data-cluster \ --services data-service
```

3.3 Microsoft Azure

Azure offers Azure Kubernetes Service (AKS) and Azure Container Instances for container deployment.

Deploying to Azure AKS - Step by Step:

```
# Step 1: Install Azure CLI and login az login # Step 2: Create resource group az group create
--name DataRG --location eastus # Step 3: Create Azure Container Registry (ACR) az acr create
--resource-group DataRG \ --name mydataregistry --sku Basic # Step 4: Login to ACR az acr
login --name mydataregistry # Step 5: Push image to ACR docker tag myapp:v1
mydataregistry.azurecr.io/myapp:v1 docker push mydataregistry.azurecr.io/myapp:v1
```

Step 6: Create AKS Cluster and Deploy

```
# Create AKS cluster az aks create --resource-group DataRG \ --name data-aks-cluster \
--node-count 3 \ --node-vm-size Standard_DS2_v2 \ --attach-acr mydataregistry \
--generate-ssh-keys # Get cluster credentials az aks get-credentials --resource-group DataRG \
--name data-aks-cluster # Deploy application kubectl create deployment myapp \
--image=mydataregistry.azurecr.io/myapp:v1 \ --replicas=3 # Expose service kubectl expose
deployment myapp \ --type=LoadBalancer --port=80 # Monitor deployment kubectl get services
kubectl get pods
```

4. Application Deployment Examples

4.1 Web Application Deployment

Deploying a Flask/Django web application with PostgreSQL database and Nginx reverse proxy.

Dockerfile for Flask Web App:

```
# Dockerfile FROM python:3.11-slim WORKDIR /app # Install dependencies COPY
requirements.txt . RUN pip install --no-cache-dir -r requirements.txt # Copy
application code COPY . . # Expose port EXPOSE 5000 # Run application CMD
["gunicorn", "-b", "0.0.0.0:5000", "app:app"]
```

docker-compose.yml for Complete Stack:

```
version: '3.8' services: db: image: postgres:15 environment: POSTGRES_DB: webapp_db
POSTGRES_USER: admin POSTGRES_PASSWORD: secure_password volumes: -
postgres_data:/var/lib/postgresql/data networks: - backend web: build: . ports: - "5000:5000"
environment: DATABASE_URL: postgresql://admin:secure_password@db:5432/webapp_db depends_on: -
db networks: - backend - frontend nginx: image: nginx:alpine ports: - "80:80" volumes: -
./nginx.conf:/etc/nginx/nginx.conf:ro depends_on: - web networks: - frontend volumes:
postgres_data: networks: backend: frontend:
```

Local Deployment:

```
# Build and start all services docker-compose up -d # View logs docker-compose logs
-f web # Scale web service docker-compose up -d --scale web=3 # Stop services
docker-compose down
```

Cloud Deployment (GCP):

```
# Build and push to GCR docker build -t gcr.io/PROJECT_ID/webapp:v1 . docker push
gcr.io/PROJECT_ID/webapp:v1 # Deploy to Cloud Run (serverless) gcloud run deploy webapp \
--image gcr.io/PROJECT_ID/webapp:v1 \ --platform managed \ --region us-central1 \
--allow-unauthenticated \ --set-env-vars DATABASE_URL=postgresql://... # Or deploy to GKE
kubectl apply -f k8s-deployment.yaml
```

4.2 AI/ML Model Pipeline Deployment

Deploying a machine learning inference API with model serving, monitoring, and scaling.

Dockerfile for ML Model API:

```
# Dockerfile for ML API FROM python:3.11-slim WORKDIR /app # Install ML dependencies COPY
requirements.txt . RUN pip install --no-cache-dir -r requirements.txt # Copy model and code
COPY models/ ./models/ COPY src/ ./src/ # Expose API port EXPOSE 8000 # Health check
HEALTHCHECK --interval=30s --timeout=10s --retries=3 \ CMD curl -f
http://localhost:8000/health || exit 1 # Run FastAPI server CMD ["uvicorn", "src.main:app",
"--host", "0.0.0.0", "--port", "8000"] # requirements.txt includes: # fastapi #
uvicorn[standard] # tensorflow or pytorch # scikit-learn # pandas # numpy
```

docker-compose.yml with Model Serving:

```
version: '3.8' services: ml-api: build: . ports: - "8000:8000" environment: MODEL_PATH:
/app/models/model.pkl INFERENCE_BATCH_SIZE: 32 volumes: - ./models:/app/models:ro deploy:
resources: reservations: devices: - driver: nvidia count: 1 capabilities: [gpu] prometheus:
image: prom/prometheus ports: - "9090:9090" volumes: -
./prometheus.yml:/etc/prometheus/prometheus.yml grafana: image: grafana/grafana ports: -
"3000:3000" depends_on: - prometheus
```

Kubernetes Deployment for Horizontal Pod Autoscaling:

```
# ml-deployment.yaml apiVersion: apps/v1 kind: Deployment metadata: name: ml-api spec:
replicas: 3 selector: matchLabels: app: ml-api template: metadata: labels: app: ml-api spec:
containers: - name: ml-api image: gcr.io/PROJECT_ID/ml-api:v1 ports: - containerPort: 8000
resources: requests: memory: "2Gi" cpu: "1000m" limits: memory: "4Gi" cpu: "2000m" ---
apiVersion: autoscaling/v2 kind: HorizontalPodAutoscaler metadata: name: ml-api-hpa spec:
scaleTargetRef: apiVersion: apps/v1 kind: Deployment name: ml-api minReplicas: 3 maxReplicas:
10 metrics: - type: Resource resource: name: cpu target: type: Utilization averageUtilization:
70
```

4.3 Big Data & PySpark CI/CD Pipeline

Containerizing Apache Spark applications with CI/CD pipeline for automated testing and deployment.

Dockerfile for PySpark Application:

```
# Dockerfile for Spark App FROM apache/spark:3.5.0-python3 USER root # Install additional
dependencies RUN pip install --no-cache-dir \ pyspark==3.5.0 \ delta-spark \ pandas \ pyarrow
\ great-expectations # Copy application code COPY jobs/ /opt/spark-apps/ COPY configs/
/opt/spark-configs/ # Set working directory WORKDIR /opt/spark-apps USER spark # Default
command runs spark-submit ENTRYPOINT ["/opt/spark/bin/spark-submit"] CMD ["--master",
"spark://spark-master:7077", "jobs/etl_pipeline.py"]
```

docker-compose.yml for Spark Cluster:

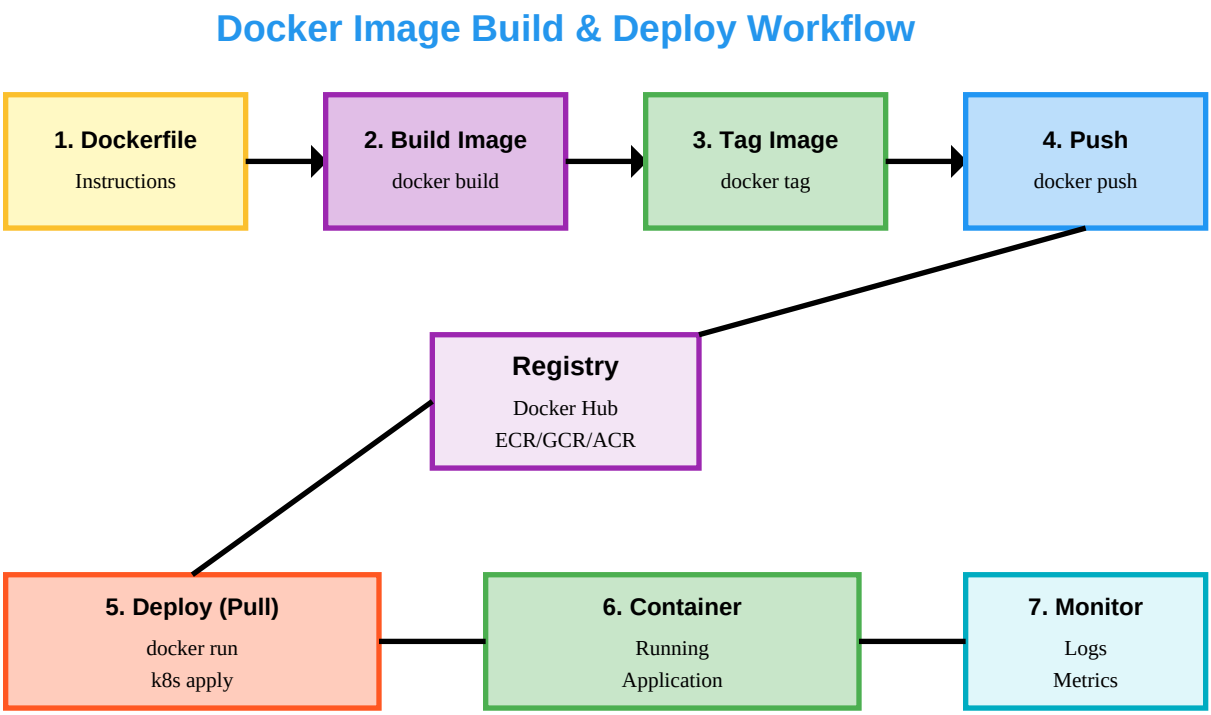
```
version: '3.8' services: spark-master: image: apache/spark:3.5.0 command: bin/spark-class
org.apache.spark.deploy.master.Master ports: - "8080:8080" - "7077:7077" environment: -
SPARK_MASTER_HOST=spark-master - SPARK_MASTER_PORT=7077 - SPARK_MASTER_WEBUI_PORT=8080
spark-worker-1: image: apache/spark:3.5.0 command: bin/spark-class
org.apache.spark.deploy.worker.Worker spark://spark-master:7077 depends_on: - spark-master
environment: - SPARK_WORKER_CORES=2 - SPARK_WORKER_MEMORY=4g -
SPARK_MASTER=spark://spark-master:7077 spark-worker-2: image: apache/spark:3.5.0 command:
bin/spark-class org.apache.spark.deploy.worker.Worker spark://spark-master:7077 depends_on: -
spark-master environment: - SPARK_WORKER_CORES=2 - SPARK_WORKER_MEMORY=4g -
SPARK_MASTER=spark://spark-master:7077 spark-app: build: . depends_on: - spark-master volumes:
- ./data:/data - ./output:/output
```

CI/CD Pipeline with GitHub Actions:

```
# .github/workflows/spark-deploy.yml name: Spark App CI/CD on: push: branches: [main] jobs:
test-and-deploy: runs-on: ubuntu-latest steps: - uses: actions/checkout@v3 - name: Run Unit
Tests run: | docker build -t spark-app:test . docker run spark-app:test pytest tests/ - name:
Build and Push to Registry env: REGISTRY: gcr.io PROJECT_ID: ${{ secrets.GCP_PROJECT_ID }}
run: | echo ${{ secrets.GCP_SA_KEY }} | docker login -u _json_key --password-stdin
https://gcr.io docker build -t $REGISTRY/$PROJECT_ID/spark-app:${{ github.sha }} . docker push
$REGISTRY/$PROJECT_ID/spark-app:${{ github.sha }} - name: Deploy to GKE run: | gcloud
container clusters get-credentials spark-cluster kubectl set image deployment/spark-app \
spark-app=$REGISTRY/$PROJECT_ID/spark-app:${{ github.sha }} kubectl rollout status
deployment/spark-app
```

5. Creating Docker Images

Understanding how to create efficient, secure Docker images is essential for data engineering workflows.



Dockerfile Best Practices:

• **Use Official Base Images:** Start with official images (python:3.11-slim, openjdk:11) for security and reliability.

• **Multi-Stage Builds:** Separate build and runtime stages to reduce final image size.

• **Layer Caching:** Order instructions from least to most frequently changing (dependencies before code).

• **Minimize Layers:** Combine RUN commands with && to reduce layer count.

• **Use .dockerignore:** Exclude unnecessary files (.git, tests, docs) from build context.

• **Security Scanning:** Regularly scan images with tools like Trivy or Snyk.

Example: Multi-Stage Dockerfile for Python Data App

```
# Stage 1: Builder FROM python:3.11 AS builder WORKDIR /build # Install build dependencies RUN apt-get update && apt-get install -y gcc g++ \ && rm -rf /var/lib/apt/lists/* # Install Python packages COPY requirements.txt . RUN pip install --user --no-cache-dir -r requirements.txt # Stage 2: Runtime FROM python:3.11-slim WORKDIR /app # Copy only installed packages from builder COPY --from=builder /root/.local /root/.local # Copy application code COPY src/ ./src/ COPY data/ ./data/ # Update PATH ENV PATH=/root/.local/bin:$PATH # Non-root user for security RUN useradd -m -u 1000 appuser && chown -R appuser:appuser /app USER appuser EXPOSE 8000 CMD ["python", "src/main.py"] # Result: Image size reduced from 1.2GB to 400MB
```

Build and Tag Commands:

```
# Build image docker build -t myapp:v1.0 . # Build with build arguments docker build --build-arg PYTHON_VERSION=3.11 -t myapp:v1.0 . # Tag for registry docker tag myapp:v1.0 myregistry.com/myapp:v1.0 docker tag myapp:v1.0 myregistry.com/myapp:latest # Push to registry docker push myregistry.com/myapp:v1.0 docker push myregistry.com/myapp:latest # Build for multiple platforms (ARM & x86) docker buildx build --platform linux/amd64,linux/arm64 \ -t myregistry.com/myapp:v1.0 --push .
```

6. Best Practices for Data Engineers

1. **Stateless Applications:** Design containers to be stateless. Store data in external volumes, databases, or object storage (S3, GCS). This enables easy scaling and recovery.

2. **Environment Variables:** Use environment variables for configuration instead of hardcoding values. Use tools like dotenv or secrets management (Vault, AWS Secrets Manager).

3. **Health Checks:** Implement /health endpoints and configure Docker HEALTHCHECK instructions. Kubernetes probes (liveness, readiness) depend on these.

4. **Resource Limits:** Always set CPU and memory limits to prevent resource exhaustion. Use docker run --cpus=2 --memory=4g or Kubernetes resource specifications.

5. **Logging Strategy:** Log to stdout/stderr for container-native logging. Use log aggregation tools (ELK stack, Loki, CloudWatch) for centralized monitoring.

6. **Security Scanning:** Integrate vulnerability scanning into CI/CD pipelines. Tools: Trivy, Snyk, Anchore. Never deploy images with critical vulnerabilities.

7. **Image Tagging Strategy:** Use semantic versioning (v1.2.3) and immutable tags. Avoid using 'latest' in production. Tag format: registry/app:version-commit_sha

8. **Network Isolation:** Use custom networks for multi-container applications. Separate frontend, backend, and database networks for security.

9. **Volume Management:** Use named volumes for persistence. Backup volumes regularly. For big data workloads, consider mounting cloud storage (S3, GCS) via FUSE.

10. **Monitoring & Alerting:** Deploy Prometheus + Grafana for metrics. Set up alerts for high CPU/memory usage, container restarts, and failed deployments.

7. Troubleshooting Guide

• **Container Fails to Start:** Check logs with 'docker logs container_id'. Verify environment variables and volume mounts. Ensure port is not already in use.

• **Out of Memory:** Container killed with exit code 137. Increase memory limit or optimize application memory usage. Check 'docker stats' for resource usage.

• **Image Pull Errors:** Authentication failure or network issues. Run 'docker login' or check registry credentials. Verify image name and tag exist.

• **Networking Issues:** Containers can't communicate. Check if they're on same network. Use 'docker network inspect' to verify. Ensure correct port mapping.

• **Volume Permission Denied:** User ID mismatch between host and container. Use USER directive in Dockerfile or chown volumes appropriately.

• **Build Cache Issues:** Use --no-cache flag to force fresh build. Clear old images with 'docker system prune -a' to free space.

Useful Troubleshooting Commands:

```
# View container logs docker logs -f container_id # Execute shell in running
container docker exec -it container_id /bin/bash # Inspect container details docker
inspect container_id # View resource usage docker stats # Check network connections
docker network ls docker network inspect network_name # Remove unused resources
docker system prune -a --volumes # View detailed build output docker build
--progress=plain -t myapp:v1 . # Export container filesystem docker export
container_id > container.tar
```

Conclusion: Docker has revolutionized how data engineers deploy and manage applications. By mastering containerization, orchestration, and cloud deployment, you can build scalable, portable, and maintainable data platforms. Continue learning by exploring Kubernetes advanced features, service mesh (Istio), and GitOps workflows (ArgoCD, Flux).

Additional Resources:

• Docker Documentation: docs.docker.com
• Kubernetes Documentation: kubernetes.io/docs
• Docker Hub: hub.docker.com
• Cloud Provider Guides: GCP, AWS, Azure container documentation

Docker Command Reference

Complete Guide with Detailed Explanations

Every command explained with purpose, options, and best practices

Container Management Commands

`docker run`

Creates and starts a new container from an image. This is the most fundamental Docker command combining 'docker create' and 'docker start' into one operation.

Basic Usage:

```
docker run [OPTIONS] IMAGE [COMMAND]
```

Common Examples:

`docker run nginx`

→ Runs nginx in foreground. Your terminal will be attached to container output. Press Ctrl+C to stop.

`docker run -d nginx`

→ Runs nginx in detached mode (background). Container runs independently of your terminal. Get container ID to manage it later.

`docker run -it ubuntu bash`

→ Opens interactive bash shell in Ubuntu container. '-i' keeps STDIN open, '-t' allocates terminal. Perfect for debugging and exploration.

`docker run -p 8080:80 nginx`

→ Maps host port 8080 to container port 80. Access nginx at localhost:8080. Format: HOST_PORT:CONTAINER_PORT

`docker run --name myapp nginx`

→ Assigns name 'myapp' to container. Easier to reference than random names or IDs. Names must be unique.

`docker run -v /host/path:/container/path nginx`

→ Mounts host directory into container. Changes in either location are visible to both. Essential for data persistence.

`docker run -e API_KEY=secret123 myapp`

→ Sets environment variable API_KEY inside container. Apps read config from env vars. Can pass multiple -e flags.

`docker run --rm ubuntu echo hello`

→ Runs command and automatically removes container when done. Useful for one-off tasks to avoid container buildup.

`docker run --restart=always nginx`

→ Container automatically restarts if it crashes or after Docker daemon restarts. Options: no, on-failure, always, unless-stopped

`docker run --network=mynet nginx`

→ Connects container to custom network. Containers on same network can communicate using container names as hostnames.

Option	Purpose	Example
`-d`	Run in background	docker run -d nginx

`-p`	Publish port	docker run -p 80:80 nginx
`-v`	Mount volume	docker run -v data:/data app
`-e`	Set env variable	docker run -e KEY=val app
`--name`	Name container	docker run --name web nginx
`--rm`	Auto-remove when stops	docker run --rm ubuntu ls
`-it`	Interactive terminal	docker run -it ubuntu bash
`--network`	Connect to network	docker run --network=net1 app
`--restart`	Restart policy	docker run --restart=always app
`--memory`	Memory limit	docker run --memory=1g app
`--cpus`	CPU limit	docker run --cpus=2 app

■ **Pro Tip:** Combine options for powerful configurations. Example: `docker run -d -p 8080:80 -v data:/app/data --name webapp --restart=always myapp:v1` runs a production-ready web application with port mapping, data persistence, auto-restart, and easy management.

`docker ps`

Lists containers. Default shows only running containers. Essential for monitoring and getting container IDs/names for other commands.

`docker ps`

→ Shows running containers with ID, image, command, status, ports, names

`docker ps -a`

→ Shows ALL containers including stopped ones. Use for cleanup and debugging

`docker ps -q`

→ Shows only container IDs. Perfect for scripting: docker stop $(docker ps -q)

`docker ps --filter status=running`

→ Filters by status. Other statuses: exited, paused, created

`docker ps --format 'table {{.Names}} {{.Status}}'`

→ Custom output format for better readability

`docker stop / start / restart`

Controls container lifecycle. 'stop' gracefully shuts down (SIGTERM then SIGKILL), 'start' resumes stopped container, 'restart' combines stop+start.

`docker stop container_name`

→ Graceful shutdown. Waits 10s for process to exit cleanly before force-killing

`docker stop -t 30 container_name`

→ Custom grace period of 30 seconds before force kill

`docker start container_name`

→ Starts previously stopped container. Container retains all data

`docker restart container_name`

→ Stops then starts container. Useful for applying config changes

`docker kill container_name`

→ Immediately kills container (SIGKILL). Use only when stop fails

`docker pause container_name`

→ Freezes all processes using cgroups. Container remains in memory

`docker unpause container_name`

→ Resumes paused container processes

Image Management Commands

`docker build`

Builds Docker image from Dockerfile. Each instruction in Dockerfile creates a new layer. Layers are cached for faster subsequent builds.

```
docker build -t myapp:v1.0 .
```

→ Builds image and tags it as 'myapp:v1.0'. Dot (.) is build context (current directory). All files in context are sent to Docker daemon.

```
docker build -t myapp:v1.0 -f Dockerfile.prod .
```

→ Uses custom Dockerfile name. Useful for multiple environments (dev/staging/prod). Default filename is 'Dockerfile'.

```
docker build --no-cache -t myapp:v1.0 .
```

→ Builds without using cached layers. Forces fresh build. Use when dependencies must update or debugging build issues.

```
docker build --build-arg VERSION=1.0 -t myapp .
```

→ Passes build-time variable. Access in Dockerfile with ARG VERSION. Useful for versions, API keys, build configs.

```
docker build --target production -t myapp .
```

→ Builds specific stage from multi-stage Dockerfile. Enables different outputs from single Dockerfile (dev with tools, prod minimal).

```
docker build --platform linux/amd64,linux/arm64 -t myapp .
```

→ Builds for multiple CPU architectures. Creates manifest list. Essential for supporting different hardware (Intel, ARM/Apple Silicon).

`docker push / pull`

Push uploads images to registry, pull downloads them. Layers are transferred incrementally - only changed layers are uploaded/downloaded.

```
docker pull nginx:1.25
```

→ Downloads nginx version 1.25 from Docker Hub. Image stored locally for future use. Always specify version in production.

```
docker pull gcr.io/project/myapp:v1
```

→ Pulls from Google Container Registry. URL format: REGISTRY/PROJECT/IMAGE:TAG. Must authenticate first: docker login gcr.io

```
docker tag myapp:v1 myregistry.com/myapp:v1
```

→ Creates new tag pointing to same image. Required before pushing to custom registry. Doesn't duplicate image data.

```
docker push myregistry.com/myapp:v1
```

→ Uploads image to registry. Must have push permissions. Only uploads layers not already present in registry.

```
docker push myregistry.com/myapp:v1 --all-tags
```

→ Pushes all tags of the image. Useful when you've tagged image as both :v1.0 and :latest

Network Commands

`docker network create mynet`

→ Creates custom bridge network. Containers on this network get automatic DNS resolution and can communicate using container names.

`docker network create --driver overlay --attachable swarm-net`

→ Creates overlay network for Docker Swarm. Enables communication across multiple hosts. --attachable allows non-swarm containers to connect.

`docker network create --subnet 172.20.0.0/16 mynet`

→ Creates network with custom IP range. Prevents conflicts with other networks. Docker assigns IPs from this range to containers.

`docker network connect mynet container_name`

→ Connects running container to network. Containers can be on multiple networks. Connection is immediate - no restart needed.

`docker network disconnect mynet container_name`

→ Removes container from network. Container loses connectivity to other containers on that network.

`docker network ls`

→ Lists all networks. Shows network names, IDs, drivers, and scope. Default networks: bridge, host, none.

`docker network inspect mynet`

→ Shows detailed network info: subnet, gateway, connected containers, options. Essential for debugging connectivity issues.

`docker network prune`

→ Removes all unused networks. Network is unused if no containers are connected. Helps cleanup and free resources.

■ **Best Practice:** Always use custom networks for multi-container apps. Default bridge network doesn't provide DNS resolution. With custom networks, containers can reach each other using names: `http://database:5432` instead of IP addresses.

Volume Commands

`docker volume create mydata`

→ Creates named volume managed by Docker. Stored in /var/lib/docker/volumes/ by default. Survives container deletion - essential for databases.

`docker volume create --driver local --opt type=nfs mydata`

→ Creates volume with custom driver. NFS enables shared storage across hosts. Cloud providers offer drivers for EBS, GCE persistent disks.

`docker run -v mydata:/data nginx`

→ Mounts named volume to container. Data written to /data in container is stored in mydata volume. Multiple containers can share same volume.

`docker run -v /host/path:/container/path:ro nginx`

→ Bind mount with read-only flag. Container can read but not modify host files. Security best practice. Use :rw for read-write (default).

`docker run -v $(pwd):/app nginx`

→ Mounts current directory into container. Perfect for development - code changes immediately visible in container. $(pwd) expands to current path.

`docker volume ls`

→ Lists all volumes. Shows volume names and drivers. Identify unused volumes for cleanup.

`docker volume inspect mydata`

→ Shows volume details: mountpoint on host, driver, options. Find where Docker stores volume data for backup purposes.

`docker volume rm mydata`

→ Permanently deletes volume and ALL its data. Volume must not be in use. Irreversible - backup first!

`docker volume prune`

→ Removes all unused volumes. Unused = not mounted by any container (running or stopped). Frees disk space but be careful with important data.

Backup Volume Example:
`docker run --rm -v mydata:/source -v $(pwd):/backup ubuntu tar czf /backup/mydata.tar.gz /source`
→ Creates temporary Ubuntu container, mounts volume and current directory, creates compressed backup. Container automatically removed after backup completes.

Troubleshooting & Debugging Commands

`docker logs container_name`

→ Shows all logs from container. Application logs written to stdout/stderr are captured. Essential for debugging failures.

`docker logs -f --tail 100 container_name`

→ -f follows logs in real-time like 'tail -f'. --tail 100 shows last 100 lines. Press Ctrl+C to stop following.

`docker logs --since 1h container_name`

→ Shows logs from last hour. Also accepts: --since 2h, --since 2023-01-01, --until 2023-01-02. Useful for pinpointing issues.

`docker exec -it container_name /bin/bash`

→ Opens interactive shell in running container. Your 'ssh into container' command. Debug live issues, check files, inspect processes.

`docker exec container_name ls -la /app`

→ Runs command in container without entering shell. Get quick output. Works with any command: ps aux, cat file, env

`docker inspect container_name`

→ Shows complete container config: environment variables, volumes, networks, resources, health status. JSON output - pipe to jq for parsing.

`docker inspect --format '{{.State.Status}}' container_name`

→ Extracts specific field from inspect output. Useful for scripts and automation. Check docs for available fields.

`docker stats`

→ Real-time resource usage: CPU, memory, network I/O, disk I/O for all containers. Press Ctrl+C to stop. Add container name for specific container.

`docker top container_name`

→ Shows processes running inside container. Like 'ps' but for container namespace. See what's consuming resources.

`docker events`

→ Streams Docker daemon events in real-time. See containers starting/stopping, images being pulled, networks created. Great for monitoring.

`docker system df`

→ Shows disk usage by images, containers, volumes, build cache. Identify what's consuming space before cleanup.

`docker system prune -a --volumes --force`

→ Nuclear option cleanup. Removes: stopped containers, unused images, unused networks, unused volumes, build cache. --force skips confirmation prompt.

Docker Compose Commands

Docker Compose manages multi-container applications. Define services, networks, volumes in docker-compose.yml, then control entire stack with simple commands.

`docker-compose up`

→ Builds, creates, and starts all services defined in docker-compose.yml. Runs in foreground showing combined logs from all services. Press Ctrl+C to stop.

`docker-compose up -d`

→ Starts services in background (detached mode). Services run independently. Use 'docker-compose logs' to view output.

`docker-compose up --build`

→ Forces rebuild of images before starting services. Use when Dockerfile or dependencies changed. Without --build, existing images are used.

`docker-compose up --scale web=3`

→ Starts 3 instances of 'web' service. Load balancer distributes traffic. Useful for testing horizontal scaling.

`docker-compose down`

→ Stops and removes all containers, networks created by 'up'. Volumes are preserved. Clean shutdown of entire application stack.

`docker-compose down -v`

→ Also removes volumes. Deletes all data. Use carefully - only for complete cleanup or testing fresh start.

`docker-compose ps`

→ Lists all containers in current Compose project. Shows status, ports, commands. Quick health check of application stack.

`docker-compose logs -f web`

→ Follows logs from 'web' service in real-time. Omit service name for all services. Add --tail 100 for last 100 lines only.

`docker-compose exec web bash`

→ Opens shell in running 'web' service container. Same as docker exec but uses service name instead of container ID.

`docker-compose restart web`

→ Restarts 'web' service. Useful after config changes. Service briefly goes down then comes back up.

`docker-compose pull`

→ Updates all images to latest versions from registry. Run before 'up' to ensure you're using newest images.

`docker-compose build`

→ Builds or rebuilds all services. Use when Dockerfiles changed. Add service name to build specific service.

Typical Workflow:
1. `docker-compose up -d` - Start application
2. `docker-compose ps` - Check all services are running
3. `docker-compose logs -f` - Monitor logs
4. `docker-compose exec web bash` - Debug if needed
5. `docker-compose down` - Stop everything

Summary: This guide covered essential Docker commands with detailed explanations of why and when to use each command. Remember:

- Use specific image versions (:v1.0) instead of :latest in production
- Always create custom networks for multi-container apps
- Use volumes for data persistence
- Monitor resources with docker stats
- Clean up regularly with prune commands

For complete deployment guides and cloud platform details, refer to the main "Docker Complete Guide for Data Engineers" document.