Linux, Docker & AWS Deployment Guide

Table of Contents

Linux Fundamentals

Essential Commands

File and Directory Operations

bash

```bash
# Navigation
pwd                    # Print working directory
ls -la                # List files with details
cd /path/to/directory  # Change directory
cd ~                  # Go to home directory
cd ..                 # Go up one directory

# File operations
touch filename.txt    # Create empty file
mkdir dirname         # Create directory
mkdir -p path/to/dir  # Create nested directories
cp source dest        # Copy files
mv source dest        # Move/rename files
rm filename           # Remove file
rm -rf dirname        # Remove directory recursively
ln -s target link     # Create symbolic link

# File viewing and editing
cat filename          # Display file content
less filename         # View file with pagination
head -10 filename     # Show first 10 lines
tail -10 filename     # Show last 10 lines
tail -f filename      # Follow file changes (logs)
nano filename         # Simple text editor
vim filename          # Advanced text editor
```

File Permissions

```bash
# Understanding permissions (rwx for user, group, others)
ls -l filename
# -rw-r--r-- 1 user group 1024 Jan 1 12:00 filename
#  ||| ||| |||
#  ||| ||| ||+-- Others permissions
#  ||| |||+---- Group permissions
#  |||+-------- User permissions
#  ++---------- File type

# Changing permissions
chmod 755 filename    # rwxr-xr-x
chmod +x script.sh    # Add execute permission
chmod u+w filename    # Add write permission for user
chown user:group file # Change ownership

# Common permission patterns
chmod 644 file.txt    # Read/write for owner, read for others
chmod 755 script.sh   # Execute permissions for scripts
chmod 600 private.key # Read/write for owner only
```

Process Management

bash

```
# Process information
ps aux               # Show all running processes
ps aux | grep python # Find Python processes
top                  # Real-time process monitor
htop                 # Enhanced process monitor
pgrep -f flask       # Find processes by name

# Process control
kill PID             # Terminate process by ID
kill -9 PID          # Force kill process
killall python      # Kill all Python processes
jobs                 # Show background jobs
bg                   # Put job in background
fg                   # Bring job to foreground
nohup command &      # Run command that survives logout

# Service management (systemd)
sudo systemctl start service-name
sudo systemctl stop service-name
sudo systemctl restart service-name
sudo systemctl enable service-name    # Auto-start on boot
sudo systemctl status service-name
```

Network and System Information

bash

```
# Network
wget https://example.com/file.zip    # Download files
curl -X GET https://api.example.com  # Make HTTP requests
netstat -tulpn                       # Show network connections
ss -tulpn                           # Modern alternative to netstat
ping google.com                     # Test connectivity
traceroute google.com               # Trace network path

# System information
uname -a            # System information
df -h               # Disk usage
du -sh *            # Directory sizes
free -h             # Memory usage
uptime              # System uptime and load
whoami              # Current user
id                  # User and group IDs
```

Package Management (Ubuntu/Debian)

```bash
# APT package manager
sudo apt update                    # Update package list
sudo apt upgrade                   # Upgrade installed packages
sudo apt install package-name     # Install package
sudo apt remove package-name      # Remove package
sudo apt autoremove               # Remove unused packages
apt search keyword                 # Search for packages
apt show package-name             # Show package information

# Adding repositories
sudo add-apt-repository ppa:repo-name
sudo apt-key add key-file

# Snap packages
sudo snap install package-name
sudo snap list
sudo snap remove package-name
```

Environment Variables and Shell

```bash
# Environment variables
echo $PATH                    # Show PATH variable
export VAR_NAME="value"      # Set environment variable
export PATH=$PATH:/new/path  # Add to PATH
env                          # Show all environment variables
unset VAR_NAME              # Remove environment variable

# Shell configuration
nano ~/.bashrc              # Edit bash configuration
source ~/.bashrc            # Reload configuration
alias ll='ls -la'           # Create command alias
history                     # Show command history
which python3               # Find command location
type python3                # Show command type and location
```

Docker Essentials

Docker Concepts

Core Components

- **Image**: Read-only template for creating containers
- **Container**: Running instance of an image
- **Dockerfile**: Text file with instructions to build an image
- **Registry**: Storage for Docker images (Docker Hub, AWS ECR)
- **Volume**: Persistent data storage for containers

Basic Docker Commands

Image Management

```bash
# Pull images from registry
docker pull ubuntu:20.04
docker pull python:3.9-slim
docker pull nginx:alpine

# List images
docker images
docker image ls

# Remove images
docker rmi image-name:tag
docker rmi image-id
docker image prune    # Remove unused images

# Build image from Dockerfile
docker build -t my-app:latest .
docker build -t my-app:v1.0 -f Dockerfile.prod .

# Tag images
docker tag my-app:latest my-registry.com/my-app:latest

# Push to registry
docker push my-registry.com/my-app:latest
```

Container Management

bash

```bash
# Run containers
docker run hello-world                    # Simple run
docker run -d nginx                      # Run in background (detached)
docker run -p 8080:80 nginx             # Port mapping
docker run -v /host/path:/container/path ubuntu  # Volume mounting
docker run -e ENV_VAR=value ubuntu      # Environment variables
docker run --name my-container nginx    # Named container
docker run -it ubuntu bash              # Interactive terminal

# List containers
docker ps          # Running containers
docker ps -a       # All containers (including stopped)

# Container operations
docker start container-name    # Start stopped container
docker stop container-name     # Stop running container
docker restart container-name  # Restart container
docker pause container-name    # Pause container
docker unpause container-name  # Unpause container

# Execute commands in running container
docker exec -it container-name bash
docker exec container-name ls /app

# View logs
docker logs container-name
docker logs -f container-name    # Follow logs

# Remove containers
docker rm container-name
docker rm -f container-name     # Force remove running container
docker container prune          # Remove all stopped containers
```

Docker Compose

yaml

```
# docker-compose.yml
version: '3.8'

services:
  web:
    build: .
    ports:
      - "5000:5000"
    environment:
      - FLASK_ENV=development
    volumes:
      - .:/app
    depends_on:
      - db

  db:
    image: postgres:13
    environment:
      POSTGRES_DB: myapp
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    volumes:
      - postgres_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"

volumes:
  postgres_data:
```

bash

```
# Docker Compose commands
docker-compose up                # Start services
docker-compose up -d             # Start in background
docker-compose down              # Stop and remove services
docker-compose build             # Build services
docker-compose logs web          # View service logs
docker-compose exec web bash     # Execute command in service
```

Creating Dockerfiles

Basic Dockerfile Structure

```dockerfile
# Use official Python runtime as base image
FROM python:3.9-slim

# Set working directory in container
WORKDIR /app

# Copy requirements first (for better caching)
COPY requirements.txt .

# Install dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY . .

# Expose port
EXPOSE 5000

# Set environment variables
ENV FLASK_APP=app.py
ENV FLASK_ENV=production

# Create non-root user for security
RUN adduser --disabled-password --gecos '' appuser
USER appuser

# Command to run application
CMD ["python", "app.py"]
```

Multi-stage Dockerfile

```dockerfile
# Build stage
FROM python:3.9 as builder

WORKDIR /app
COPY requirements.txt .
RUN pip install --user -r requirements.txt

# Production stage
FROM python:3.9-slim

WORKDIR /app

# Copy installed packages from builder stage
COPY --from=builder /root/.local /root/.local

# Copy application
COPY . .

# Make sure scripts in .local are usable
ENV PATH=/root/.local/bin:$PATH

EXPOSE 5000
CMD ["python", "app.py"]
```

AWS Basics

Core AWS Services

Compute Services

- **EC2**: Virtual servers in the cloud
- **Lambda**: Serverless compute functions
- **ECS**: Container orchestration service
- **Fargate**: Serverless containers

Storage Services

- **S3**: Object storage service
- **EBS**: Block storage for EC2
- **EFS**: Network file system

Database Services

- **RDS**: Managed relational databases
- **DynamoDB**: NoSQL database
- **ElastiCache**: In-memory caching

Networking

- **VPC**: Virtual Private Cloud
- **Route 53**: DNS service
- **CloudFront**: Content Delivery Network
- **Load Balancer**: Distribute traffic

AWS CLI Setup

Installation and Configuration

```bash
# Install AWS CLI
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
unzip awscliv2.zip
sudo ./aws/install

# Configure AWS CLI
aws configure
# AWS Access Key ID: YOUR_ACCESS_KEY
# AWS Secret Access Key: YOUR_SECRET_KEY
# Default region name: us-east-1
# Default output format: json

# Verify configuration
aws sts get-caller-identity
aws s3 ls
```

Common AWS CLI Commands

bash

```bash
# EC2 operations
aws ec2 describe-instances
aws ec2 start-instances --instance-ids i-1234567890abcdef0
aws ec2 stop-instances --instance-ids i-1234567890abcdef0
aws ec2 create-security-group --group-name my-sg --description "My security group"

# S3 operations
aws s3 ls
aws s3 cp file.txt s3://my-bucket/
aws s3 sync ./local-folder s3://my-bucket/remote-folder/
aws s3 mb s3://my-new-bucket

# IAM operations
aws iam list-users
aws iam create-user --user-name myuser
aws iam attach-user-policy --user-name myuser --policy-arn arn:aws:iam::aws:policy/ReadOnlyAcce
```

Deploying Flask App to AWS

Method 1: Direct Deployment on EC2

Step 1: Launch EC2 Instance

bash

```bash
# Create key pair
aws ec2 create-key-pair --key-name my-flask-key --query 'KeyMaterial' --output text > my-flask-
chmod 400 my-flask-key.pem

# Launch instance
aws ec2 run-instances \
    --image-id ami-0c02fb55956c7d316 \
    --count 1 \
    --instance-type t2.micro \
    --key-name my-flask-key \
    --security-groups flask-app-sg
```

Step 2: Connect to Instance

bash

```
# Get instance public IP
aws ec2 describe-instances --query 'Reservations[*].Instances[*].PublicIpAddress' --output text

# Connect via SSH
ssh -i my-flask-key.pem ubuntu@YOUR_INSTANCE_IP
```

Step 3: Setup Environment on EC2

bash

```
# Update system
sudo apt update && sudo apt upgrade -y

# Install Python and dependencies
sudo apt install -y python3 python3-pip python3-venv nginx

# Create application directory
sudo mkdir -p /var/www/flask-app
sudo chown ubuntu:ubuntu /var/www/flask-app
cd /var/www/flask-app

# Create virtual environment
python3 -m venv venv
source venv/bin/activate

# Clone your application (or upload files)
git clone https://github.com/yourusername/your-flask-app.git .
# OR upload files using scp:
# scp -i my-flask-key.pem -r ./my-flask-app ubuntu@YOUR_INSTANCE_IP:/var/www/flask-app/

# Install dependencies
pip install -r requirements.txt
pip install gunicorn
```

Step 4: Configure Gunicorn

bash

```
# Create Gunicorn configuration
cat > gunicorn.conf.py << EOF
bind = "127.0.0.1:5000"
workers = 2
worker_class = "sync"
worker_connections = 1000
max_requests = 1000
max_requests_jitter = 100
timeout = 30
keepalive = 5
preload_app = True
EOF

# Create systemd service file
sudo tee /etc/systemd/system/flask-app.service << EOF
[Unit]
Description=Gunicorn instance to serve Flask App
After=network.target

[Service]
User=ubuntu
Group=www-data
WorkingDirectory=/var/www/flask-app
Environment="PATH=/var/www/flask-app/venv/bin"
ExecStart=/var/www/flask-app/venv/bin/gunicorn --config gunicorn.conf.py app:app
ExecReload=/bin/kill -s HUP \$MAINPID
Restart=always

[Install]
WantedBy=multi-user.target
EOF

# Start and enable service
sudo systemctl daemon-reload
sudo systemctl start flask-app
sudo systemctl enable flask-app
sudo systemctl status flask-app
```

Step 5: Configure Nginx

bash

```bash
# Create Nginx configuration
sudo tee /etc/nginx/sites-available/flask-app << EOF
server {
    listen 80;
    server_name YOUR_DOMAIN_OR_IP;

    location / {
        proxy_pass http://127.0.0.1:5000;
        proxy_set_header Host \$host;
        proxy_set_header X-Real-IP \$remote_addr;
        proxy_set_header X-Forwarded-For \$proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto \$scheme;
    }

    # Optional: Serve static files directly
    location /static {
        alias /var/www/flask-app/static;
        expires 1y;
        add_header Cache-Control "public, immutable";
    }
}
EOF

# Enable site and restart Nginx
sudo ln -s /etc/nginx/sites-available/flask-app /etc/nginx/sites-enabled/
sudo nginx -t
sudo systemctl restart nginx
```

Step 6: Configure Security Group

bash

```bash
# Create security group
aws ec2 create-security-group \
    --group-name flask-app-sg \
    --description "Security group for Flask application"

# Allow HTTP traffic
aws ec2 authorize-security-group-ingress \
    --group-name flask-app-sg \
    --protocol tcp \
    --port 80 \
    --cidr 0.0.0.0/0

# Allow HTTPS traffic
aws ec2 authorize-security-group-ingress \
    --group-name flask-app-sg \
    --protocol tcp \
    --port 443 \
    --cidr 0.0.0.0/0

# Allow SSH access
aws ec2 authorize-security-group-ingress \
    --group-name flask-app-sg \
    --protocol tcp \
    --port 22 \
    --cidr 0.0.0.0/0
```

Method 2: Using AWS Elastic Beanstalk

Step 1: Prepare Application

```bash
# Create application.py (Elastic Beanstalk expects this name)
cp app.py application.py

# Create requirements.txt
pip freeze > requirements.txt

# Create .ebextensions/python.config
mkdir .ebextensions
cat > .ebextensions/python.config << EOF
option_settings:
  aws:elasticbeanstalk:container:python:
    WSGIPath: application:app
  aws:elasticbeanstalk:environment:proxy:staticfiles:
    /static: static
EOF
```

Step 2: Deploy with EB CLI

```bash
# Install EB CLI
pip install awsebcli

# Initialize Elastic Beanstalk application
eb init flask-app --region us-east-1 --platform python-3.9

# Create environment and deploy
eb create flask-app-env

# Deploy updates
eb deploy

# Open application in browser
eb open

# View logs
eb logs

# Terminate environment (cleanup)
eb terminate flask-app-env
```

Dockerizing Flask Applications

Complete Flask Application Structure

```
flask-docker-app/
├── app.py
├── requirements.txt
├── Dockerfile
├── docker-compose.yml
├── .dockerignore
├── nginx/
│   └── nginx.conf
└── static/
    └── styles.css
```

Sample Flask Application

python

```python
# app.py
from flask import Flask, jsonify, request
import os
import redis
from datetime import datetime

app = Flask(__name__)

# Redis connection (optional)
try:
    redis_client = redis.Redis(
        host=os.environ.get('REDIS_HOST', 'localhost'),
        port=int(os.environ.get('REDIS_PORT', 6379)),
        decode_responses=True
    )
except:
    redis_client = None

@app.route('/')
def home():
    return jsonify({
        "message": "Flask Docker App",
        "timestamp": datetime.now().isoformat(),
        "environment": os.environ.get('FLASK_ENV', 'production')
    })

@app.route('/health')
def health():
    return jsonify({"status": "healthy"}), 200

@app.route('/api/counter', methods=['GET', 'POST'])
def counter():
    if not redis_client:
        return jsonify({"error": "Redis not available"}), 503

    if request.method == 'POST':
        count = redis_client.incr('counter')
        return jsonify({"count": count}), 201
    else:
        count = redis_client.get('counter') or 0
        return jsonify({"count": int(count)})

if __name__ == '__main__':
    app.run(
        host='0.0.0.0',
        port=int(os.environ.get('PORT', 5000)),
        debug=os.environ.get('FLASK_ENV') == 'development'
    )
```

Production Dockerfile

dockerfile

```
# Dockerfile
FROM python:3.9-slim

# Set environment variables
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1
ENV FLASK_APP=app.py
ENV FLASK_ENV=production

# Set work directory
WORKDIR /app

# Install system dependencies
RUN apt-get update \
    && apt-get install -y --no-install-recommends \
        gcc \
        && rm -rf /var/lib/apt/lists/*

# Install Python dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy project
COPY . .

# Create non-root user
RUN adduser --disabled-password --gecos '' appuser \
    && chown -R appuser:appuser /app
USER appuser

# Expose port
EXPOSE 5000

# Health check
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
    CMD curl -f http://localhost:5000/health || exit 1

# Run application
CMD ["gunicorn", "--bind", "0.0.0.0:5000", "--workers", "2", "app:app"]
```

Docker Compose for Development

yaml

```yaml
# docker-compose.yml
version: '3.8'

services:
  web:
    build: .
    ports:
      - "5000:5000"
    environment:
      - FLASK_ENV=development
      - REDIS_HOST=redis
      - REDIS_PORT=6379
    volumes:
      - .:/app
    depends_on:
      - redis
    restart: unless-stopped

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data
    restart: unless-stopped

  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - web
    restart: unless-stopped

volumes:
  redis_data:
```

Production Docker Compose

```
yaml
```

```yaml
# docker-compose.prod.yml
version: '3.8'

services:
  web:
    build:
      context: .
      dockerfile: Dockerfile
    environment:
      - FLASK_ENV=production
      - REDIS_HOST=redis
      - DATABASE_URL=postgresql://user:pass@db:5432/myapp
    depends_on:
      - redis
      - db
    restart: unless-stopped
    networks:
      - app-network

  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
      - ./nginx/ssl:/etc/nginx/ssl:ro
    depends_on:
      - web
    restart: unless-stopped
    networks:
      - app-network

  redis:
    image: redis:7-alpine
    volumes:
      - redis_data:/data
    restart: unless-stopped
    networks:
      - app-network

  db:
    image: postgres:13
    environment:
      POSTGRES_DB: myapp
      POSTGRES_USER: user
    POSTGRES_PASSWORD: password
  volumes:
    - postgres_data:/var/lib/postgresql/data
  restart: unless-stopped
  networks:
    - app-network

volumes:
  redis_data:
  postgres_data:

networks:
  app-network:
    driver: bridge
```

Nginx Configuration

nginx

```
# nginx/nginx.conf
events {
    worker_connections 1024;
}

http {
    upstream app {
        server web:5000;
    }

    server {
        listen 80;
        server_name localhost;

        client_max_body_size 10M;

        location / {
            proxy_pass http://app;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
            proxy_redirect off;
        }

        location /static/ {
            alias /app/static/;
            expires 1y;
            add_header Cache-Control "public, immutable";
        }
    }
}
```

.dockerignore

```
# .dockerignore
.git
.gitignore
README.md
Dockerfile
docker-compose*.yml
.env
.venv
venv/
__pycache__/
*.pyc
*.pyo
*.pyd
.Python
.pytest_cache
.coverage
htmlcov/
.tox/
.cache
nosetests.xml
coverage.xml
```

Deploying Docker Containers on AWS EC2

Method 1: Docker on EC2

Step 1: Launch EC2 Instance with Docker

bash

```
# Create user data script for Docker installation
cat > user-data.sh << 'EOF'
#!/bin/bash
yum update -y
amazon-linux-extras install docker
service docker start
usermod -a -G docker ec2-user
systemctl enable docker

# Install Docker Compose
curl -L "https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname -s)-
chmod +x /usr/local/bin/docker-compose
ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose
EOF

# Launch instance with user data
aws ec2 run-instances \
    --image-id ami-0c02fb55956c7d316 \
    --count 1 \
    --instance-type t3.small \
    --key-name my-flask-key \
    --security-group-ids sg-xxxxxxxxx \
    --user-data file://user-data.sh \
    --tag-specifications 'ResourceType=instance,Tags=[{Key=Name,Value=Flask-Docker-App}]'
```

Step 2: Deploy Application

bash

```
# Connect to instance
ssh -i my-flask-key.pem ec2-user@YOUR_INSTANCE_IP

# Clone repository
git clone https://github.com/yourusername/flask-docker-app.git
cd flask-docker-app

# Build and run with Docker Compose
docker-compose -f docker-compose.prod.yml up -d

# Check status
docker-compose -f docker-compose.prod.yml ps
docker-compose -f docker-compose.prod.yml logs
```

Method 2: Using Amazon ECR (Elastic Container Registry)

Step 1: Create ECR Repository

bash

```bash
# Create repository
aws ecr create-repository --repository-name flask-app

# Get login token
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 12

# Build and tag image
docker build -t flask-app .
docker tag flask-app:latest 123456789012.dkr.ecr.us-east-1.amazonaws.com/flask-app:latest

# Push image
docker push 123456789012.dkr.ecr.us-east-1.amazonaws.com/flask-app:latest
```

Step 2: Deploy from ECR

bash

```bash
# On EC2 instance, pull and run image
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 12

docker pull 123456789012.dkr.ecr.us-east-1.amazonaws.com/flask-app:latest
docker run -d -p 80:5000 --name flask-app 123456789012.dkr.ecr.us-east-1.amazonaws.com/flask-ap
```

Method 3: Using Amazon ECS (Elastic Container Service)

Step 1: Create Task Definition

json

```json
{
  "family": "flask-app-task",
  "networkMode": "awsvpc",
  "requiresCompatibilities": ["FARGATE"],
  "cpu": "256",
  "memory": "512",
  "executionRoleArn": "arn:aws:iam::123456789012:role/ecsTaskExecutionRole",
  "containerDefinitions": [
    {
      "name": "flask-app",
      "image": "123456789012.dkr.ecr.us-east-1.amazonaws.com/flask-app:latest",
      "portMappings": [
        {
          "containerPort": 5000,
          "protocol": "tcp"
        }
      ],
      "essential": true,
      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
          "awslogs-group": "/ecs/flask-app",
          "awslogs-region": "us-east-1",
          "awslogs-stream-prefix": "ecs"
        }
      },
      "environment": [
        {
          "name": "FLASK_ENV",
          "value": "production"
        }
      ]
    }
  ]
}
```

Step 2: Create ECS Cluster and Service

```bash
# Create cluster
aws ecs create-cluster --cluster-name flask-app-cluster

# Register task definition
aws ecs register-task-definition --cli-input-json file://task-definition.json

# Create service
aws ecs create-service \
    --cluster flask-app-cluster \
    --service-name flask-app-service \
    --task-definition flask-app-task:1 \
    --desired-count 2 \
    --launch-type FARGATE \
    --network-configuration "awsvpcConfiguration={subnets=[subnet-12345678],securityGroups=[sg-
```

Advanced Deployment Strategies

Auto Scaling with Application Load Balancer

Step 1: Create Load Balancer

bash

```bash
# Create Application Load Balancer
aws elbv2 create-load-balancer \
    --name flask-app-alb \
    --subnets subnet-12345678 subnet-87654321 \
    --security-groups sg-12345678

# Create target group
aws elbv2 create-target-group \
    --name flask-app-targets \
    --protocol HTTP \
    --port 80 \
    --vpc-id vpc-12345678 \
    --health-check-path /health

# Create listener
aws elbv2 create-listener \
    --load-balancer-arn arn:aws:elasticloadbalancing:us-east-1:123456789012:loadbalancer/app/fl
    --protocol HTTP \
    --port 80 \
    --default-actions Type=forward,TargetGroupArn=arn:aws:elasticloadbalancing:us-east-1:123456
```

Step 2: Auto Scaling Group

bash

```
# Create launch template
aws ec2 create-launch-template \
    --launch-template-name flask-app-template \
    --launch-template-data '{
        "ImageId": "ami-0c02fb55956c7d316",
        "InstanceType": "t3.micro",
        "KeyName": "my-flask-key",
        "SecurityGroupIds": ["sg-12345678"],
        "UserData": "'$(base64 -w 0 user-data.sh)'"
    }'

# Create Auto Scaling Group
aws autoscaling create-auto-scaling-group \
    --auto-scaling-group-name flask-app-asg \
    --launch-template LaunchTemplateName=flask-app-template,Version=1 \
    --min-size 1 \
    --max-size 5 \
    --desired-capacity 2 \
    --target-group-arns arn:aws:elasticloadbalancing:us-east-1:123456789012:targetgroup/flask-a
    --vpc-zone-identifier "subnet-12345678,subnet-87654321"
```

CI/CD Pipeline with GitHub Actions

yaml

```yaml
# .github/workflows/deploy.yml
name: Deploy Flask App

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

env:
  AWS_REGION: us-east-1
  ECR_REPOSITORY: flask-app
  ECS_SERVICE: flask-app-service
  ECS_CLUSTER: flask-app-cluster
  ECS_TASK_DEFINITION: task-definition.json

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v3
      with:
        python-version: '3.9'

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt
        pip install pytest pytest-cov

    - name: Run tests
      run: |
        pytest tests/ --cov=app --cov-report=xml

    - name: Upload coverage to Codecov
      uses: codecov/codecov-action@v3

  build-and-deploy:
    needs: test
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'

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

      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v2
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: ${{ env.AWS_REGION }}

      - name: Login to Amazon ECR
        id: login-ecr
        uses: aws-actions/amazon-ecr-login@v1

      - name: Build, tag, and push image to Amazon ECR
        id: build-image
        env:
          ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
          IMAGE_TAG: ${{ github.sha }}
        run: |
          docker build -t $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG .
          docker push $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG
          echo "image=$ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG" >> $GITHUB_OUTPUT

      - name: Fill in the new image ID in the Amazon ECS task definition
        id: task-def
        uses: aws-actions/amazon-ecs-render-task-definition@v1
        with:
          task-definition: ${{ env.ECS_TASK_DEFINITION }}
          container-name: flask-app
          image: ${{ steps.build-image.outputs.image }}

      - name: Deploy Amazon ECS task definition
        uses: aws-actions/amazon-ecs-deploy-task-definition@v1
        with:
          task-definition: ${{ steps.task-def.outputs.task-definition }}
          service: ${{ env.ECS_SERVICE }}
          cluster: ${{ env.ECS_CLUSTER }}
          wait-for-service-stability: true
```

Blue-Green Deployment Strategy

Step 1: Setup Blue-Green Environment

bash

```bash
# Create two identical environments
# Blue environment (current production)
aws ecs create-service \
    --cluster flask-app-cluster \
    --service-name flask-app-blue \
    --task-definition flask-app-task:1 \
    --desired-count 2 \
    --launch-type FARGATE

# Green environment (new version)
aws ecs create-service \
    --cluster flask-app-cluster \
    --service-name flask-app-green \
    --task-definition flask-app-task:2 \
    --desired-count 2 \
    --launch-type FARGATE
```

Step 2: Traffic Switching Script

bash

```
#!/bin/bash
# blue-green-deploy.sh

GREEN_TARGET_GROUP_ARN="arn:aws:elasticloadbalancing:us-east-1:123456789012:targetgroup/flask-a
BLUE_TARGET_GROUP_ARN="arn:aws:elasticloadbalancing:us-east-1:123456789012:targetgroup/flask-ap
LISTENER_ARN="arn:aws:elasticloadbalancing:us-east-1:123456789012:listener/app/flask-app-alb/12

echo "Starting Blue-Green Deployment..."

# Health check green environment
echo "Checking green environment health..."
GREEN_HEALTH=$(aws elbv2 describe-target-health --target-group-arn $GREEN_TARGET_GROUP_ARN --qu

if [ "$GREEN_HEALTH" = "healthy" ]; then
    echo "Green environment is healthy. Switching traffic..."

    # Switch traffic to green
    aws elbv2 modify-listener \
        --listener-arn $LISTENER_ARN \
        --default-actions Type=forward,TargetGroupArn=$GREEN_TARGET_GROUP_ARN

    echo "Traffic switched to green environment"

    # Wait and verify
    sleep 30

    # Scale down blue environment
    aws ecs update-service \
        --cluster flask-app-cluster \
        --service flask-app-blue \
        --desired-count 0

    echo "Blue-Green deployment completed successfully"
else
    echo "Green environment is not healthy. Deployment aborted."
    exit 1
fi
```

Monitoring and Logging

CloudWatch Logs Setup

bash

```
# Create log group
aws logs create-log-group --log-group-name /aws/ecs/flask-app

# Create log stream
aws logs create-log-stream \
    --log-group-name /aws/ecs/flask-app \
    --log-stream-name flask-app-stream
```

Application Monitoring with CloudWatch

python

```
# Enhanced Flask app with monitoring
import boto3
import time
from flask import Flask, jsonify, request, g
from datetime import datetime

app = Flask(__name__)

# CloudWatch client
cloudwatch = boto3.client('cloudwatch', region_name='us-east-1')

@app.before_request
def before_request():
    g.start_time = time.time()

@app.after_request
def after_request(response):
    # Calculate response time
    response_time = (time.time() - g.start_time) * 1000

    # Send metrics to CloudWatch
    try:
        cloudwatch.put_metric_data(
            Namespace='FlaskApp/Performance',
            MetricData=[
                {
                    'MetricName': 'ResponseTime',
                    'Value': response_time,
                    'Unit': 'Milliseconds',
                    'Dimensions': [
                        {
                            'Name': 'Endpoint',
                            'Value': request.endpoint or 'unknown'
                        }
                    ]
                },
                {
                    'MetricName': 'RequestCount',
                    'Value': 1,
                    'Unit': 'Count',
                    'Dimensions': [
                        {
                            'Name': 'StatusCode',
                            'Value': str(response.status_code)
                        }
                    ]
                }
            ]
        )
    except Exception as e:
        app.logger.error(f"Failed to send metrics: {e}")

    return response

@app.route('/metrics')
def metrics():
    """Custom metrics endpoint"""
    return jsonify({
        "custom_metrics": {
            "uptime": time.time() - app.start_time,
            "timestamp": datetime.now().isoformat()
        }
    })

# Initialize start time
app.start_time = time.time()
```

Health Check Endpoint

python

```
@app.route('/health')
def health_check():
    """Comprehensive health check"""
    health_status = {
        "status": "healthy",
        "timestamp": datetime.now().isoformat(),
        "version": os.environ.get('APP_VERSION', 'unknown'),
        "environment": os.environ.get('FLASK_ENV', 'production')
    }

    # Check database connection
    try:
        # Example database health check
        # db.session.execute('SELECT 1')
        health_status["database"] = "connected"
    except Exception as e:
        health_status["status"] = "unhealthy"
        health_status["database"] = f"error: {str(e)}"

    # Check Redis connection
    try:
        if redis_client:
            redis_client.ping()
            health_status["redis"] = "connected"
        else:
            health_status["redis"] = "not_configured"
    except Exception as e:
        health_status["status"] = "unhealthy"
        health_status["redis"] = f"error: {str(e)}"

    status_code = 200 if health_status["status"] == "healthy" else 503
    return jsonify(health_status), status_code
```

Security Best Practices

Dockerfile Security

dockerfile

```dockerfile
# Security-hardened Dockerfile
FROM python:3.9-slim

# Security updates
RUN apt-get update && apt-get upgrade -y && \
    apt-get install -y --no-install-recommends \
    gcc && \
    rm -rf /var/lib/apt/lists/* && \
    apt-get clean

# Create non-root user early
RUN groupadd -r appuser && useradd -r -g appuser appuser

# Set secure working directory
WORKDIR /app

# Copy and install dependencies as root
COPY requirements.txt .
RUN pip install --no-cache-dir --upgrade pip && \
    pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY . .

# Change ownership to non-root user
RUN chown -R appuser:appuser /app

# Switch to non-root user
USER appuser

# Remove unnecessary packages
RUN pip uninstall -y pip setuptools

# Security labels
LABEL security.policy="restricted" \
      maintainer="your-team@company.com"

# Health check
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
    CMD python -c "import requests; requests.get('http://localhost:5000/health')" || exit 1

EXPOSE 5000

CMD ["gunicorn", "--bind", "0.0.0.0:5000", "--workers", "2", "--timeout", "30", "app:app"]
```

AWS Security Configuration

```bash
# Create IAM role for ECS tasks
aws iam create-role \
    --role-name ecsTaskRole \
    --assume-role-policy-document '{
        "Version": "2012-10-17",
        "Statement": [
            {
                "Effect": "Allow",
                "Principal": {
                    "Service": "ecs-tasks.amazonaws.com"
                },
                "Action": "sts:AssumeRole"
            }
        ]
    }'

# Attach minimal permissions policy
aws iam attach-role-policy \
    --role-name ecsTaskRole \
    --policy-arn arn:aws:iam::aws:policy/CloudWatchLogsFullAccess

# Create security group with minimal permissions
aws ec2 create-security-group \
    --group-name flask-app-secure-sg \
    --description "Secure security group for Flask app"

# Allow only necessary ports
aws ec2 authorize-security-group-ingress \
    --group-name flask-app-secure-sg \
    --protocol tcp \
    --port 80 \
    --source-group sg-alb-security-group-id

# No direct SSH access - use Session Manager instead
```

Backup and Disaster Recovery

Database Backup Strategy

bash

```
# Automated RDS backup
aws rds create-db-snapshot \
    --db-instance-identifier flask-app-db \
    --db-snapshot-identifier flask-app-backup-$(date +%Y%m%d%H%M%S)

# Cross-region backup
aws rds copy-db-snapshot \
    --source-db-snapshot-identifier flask-app-backup-20231201120000 \
    --target-db-snapshot-identifier flask-app-backup-20231201120000-dr \
    --source-region us-east-1 \
    --target-region us-west-2
```

Application Data Backup

bash

```
# S3 backup script
#!/bin/bash
# backup.sh

BACKUP_DATE=$(date +%Y%m%d_%H%M%S)
BACKUP_BUCKET="flask-app-backups"

# Create backup archive
tar -czf app-backup-$BACKUP_DATE.tar.gz \
    /var/www/flask-app \
    --exclude='*.pyc' \
    --exclude='__pycache__' \
    --exclude='.git'

# Upload to S3
aws s3 cp app-backup-$BACKUP_DATE.tar.gz s3://$BACKUP_BUCKET/backups/

# Set lifecycle policy for cost optimization
aws s3api put-bucket-lifecycle-configuration \
    --bucket $BACKUP_BUCKET \
    --lifecycle-configuration '{
        "Rules": [
            {
                "ID": "BackupLifecycle",
                "Status": "Enabled",
                "Filter": {"Prefix": "backups/"},
                "Transitions": [
                    {
                        "Days": 30,
                        "StorageClass": "STANDARD_IA"
                    },
                    {
                        "Days": 90,
                        "StorageClass": "GLACIER"
                    }
                ]
            }
        ]
    }'

# Clean up local backup
rm app-backup-$BACKUP_DATE.tar.gz
```

Cost Optimization

Resource Right-sizing

bash

```
# Monitor instance utilization
aws cloudwatch get-metric-statistics \
    --namespace AWS/EC2 \
    --metric-name CPUUtilization \
    --dimensions Name=InstanceId,Value=i-1234567890abcdef0 \
    --start-time 2023-11-01T00:00:00Z \
    --end-time 2023-12-01T00:00:00Z \
    --period 3600 \
    --statistics Average

# Use Spot instances for non-critical workloads
aws ec2 request-spot-instances \
    --instance-count 2 \
    --type "one-time" \
    --launch-specification '{
        "ImageId": "ami-0c02fb55956c7d316",
        "InstanceType": "t3.medium",
        "KeyName": "my-flask-key",
        "SecurityGroups": ["flask-app-sg"]
    }'
```

Container Optimization

dockerfile

```
# Multi-stage build for smaller images
FROM python:3.9 as builder
WORKDIR /app
COPY requirements.txt .
RUN pip install --user -r requirements.txt

FROM python:3.9-slim
WORKDIR /app
COPY --from=builder /root/.local /root/.local
COPY . .
ENV PATH=/root/.local/bin:$PATH
CMD ["gunicorn", "--bind", "0.0.0.0:5000", "app:app"]
```

Conclusion

This comprehensive guide covered the complete journey from Linux fundamentals to deploying containerized Flask applications on AWS. Key takeaways include:

Best Practices Summary

1. **Security First**: Always use non-root users, minimal permissions, and security groups
2. **Monitoring**: Implement comprehensive logging and monitoring from day one
3. **Automation**: Use Infrastructure as Code and CI/CD pipelines
4. **Scalability**: Design for horizontal scaling with load balancers and auto-scaling
5. **Cost Optimization**: Right-size resources and use appropriate storage classes
6. **Disaster Recovery**: Implement backup strategies and multi-region deployments

Deployment Decision Matrix

Use Case	Recommended Approach
Simple Apps	Direct EC2 + Nginx
Scalable Apps	ECS Fargate + ALB
Quick Prototyping	Elastic Beanstalk
Microservices	ECS + Service Discovery
High Traffic	ECS + Auto Scaling + CloudFront
Development	Docker Compose locally

Next Steps

1. Implement Infrastructure as Code using AWS CDK or Terraform
2. Add comprehensive testing and security scanning to CI/CD
3. Explore container orchestration with Amazon EKS
4. Implement observability with AWS X-Ray and CloudWatch Insights
5. Consider serverless options with AWS Lambda for event-driven workloads