# **How to Build a Production-Ready DevOps Pipeline with Free Tools**

**Link for the document :**

<https://www.freecodecamp.org/news/how-to-build-a-production-ready-devops-pipeline-with-free-tools/>

**Prerequisites**

* **Basic Git knowledge**: Cloning repos, creating branches, committing code, and creating PRs
* **Familiarity with command line**: For Docker, Terraform, and Kubernetes
* **Basic understanding of CI/CD**: Continuous integration/delivery concepts and pipelines

### **Accounts needed**

* GitHub account
* At least one cloud provider: AWS Free Tier (recommended), Oracle Cloud Free Tier, or Google Cloud/Azure with free credits
* Terraform Cloud (free tier) for infrastructure state management
* Grafana Cloud (free tier) for monitoring
* UptimeRobot (free tier) for external availability checks

**Step 1: Create a Well-Structured Repository**

* Separate folders for frontend, backend, and infrastructure
* A .github folder to hold workflow configurations
* Clear naming conventions and a well-written README.md

**Step 2: Set Up Branch Protection Without Paid Features**

* Require pull requests before merging
* Enable status checks to prevent broken code from landing in main

**Step 3: Implement PR Templates and Automated Checks**

* Add a PULL\_REQUEST\_TEMPLATE.md to guide contributors
* Use GitHub Actions (which we'll set up in the next part) for linting, tests, and formatting checks.

**Step 4: Configure GitHub Issue Templates and Project Boards**

* Add issue templates for bugs and features
* Use GitHub Projects to manage work with a Kanban board, all free and native to GitHub

**Step 5: Advanced Technique: Set Up Custom Validation Scripts as Pre-Commit Hooks**

Before code ever hits GitHub, you can catch issues locally with Git hooks. Using a tool like [Husky](https://typicode.github.io/husky/) or [pre-commit](https://pre-commit.com/), you can:

* Lint code before it's committed
* Run tests or formatters automatically
* Prevent secrets from being accidentally committed.

**Step 6: Create the application**

* Start Creating the application with like front-end codede in the front-end folder and backend code in the back-end folder .
* Before pushing code to GitHub check locally it's working fine or not and after that, run the lint command for improving code quality, making code more consistent, and avoiding bugs

## **How to Optimize Docker Builds for CI**

Before creating the Docker image, we need to install Docker for the desktop .

**1. Create a Baseline Dockerfile**: Create a Dockerfile in the root directory and add the following code to create the Docker image.

# Simple Dockerfile for a Node.js application

FROM node:18-alpine # Use Alpine for a smaller base image

WORKDIR /app # Set working directory

COPY . . # Copy all files to container

RUN npm ci # Install dependencies (clean install)

CMD ["npm", "start"] # Start the application

Using an Alpine-based Node.js image helps keep your image small from the start.

### **2. Multi-Stage Docker Builds**

Multi-stage builds let you compile or build your app in one stage and only copy over the final product to a clean, smaller image.

# Stage 1: Build the application

FROM node:18-alpine AS builder

WORKDIR /app

COPY package\*.json ./ # Copy package files first for better caching

RUN npm ci # Install all dependencies

COPY . . # Then copy source code

RUN npm run build # Build the application

# Stage 2: Production image with minimal footprint

FROM node:18-alpine

WORKDIR /app

# Only copy built assets and production dependencies

COPY --from=builder /app/dist ./dist

COPY --from=builder /app/package\*.json ./

RUN npm ci --production # Install only production dependencies

CMD ["node", "dist/server.js"] # Run the built application

### **3. Optimizing Layer Caching**

This way, Docker reuses the cached npm install step if your dependencies haven't changed

### **4. Enable BuildKit for Faster Builds**

Turning on BuildKit can significantly speed up complex Docker builds and is highly recommended for all CI pipelines.

- name: Build Docker image

run: |

# Enable BuildKit for parallel and more efficient builds

DOCKER\_BUILDKIT=1 docker build -t myapp:latest .

How to Build Your CI Pipeline with GitHub Actions

**Step1: Set Up Your First GitHub Actions Workflow**

First, let’s create a basic workflow that automatically builds, tests, and lints your app every time you push code or open a pull request. This ensures your app stays healthy and any issues are caught early.

* Create a file at .github/workflows/ci.yml and add the following:

name: CI Pipeline

on:

push:

branches: [main] # Trigger on pushes to main branch

pull\_request:

branches: [main] # Trigger on PRs to main branch

jobs:

build:

runs-on: ubuntu-latest # Use GitHub's free Linux runner

steps:

- uses: actions/checkout@v3 # Check out the repository code

- name: Set up Node.js # Install Node.js environment

uses: actions/setup-node@v3

with:

node-version: '18' # Use Node.js 18 for consistency

- name: Cache dependencies # Cache node\_modules to speed up builds

uses: actions/cache@v3

with:

path: ~/.npm # Cache npm’s global cache

key: ${{ runner.os }}-node-${{ hashFiles('\*\*/package-lock.json') }} # Key based on OS and package-lock.json

- run: npm ci # Install dependencies reliably using package-lock.json

- run: npm test # Run tests defined in package.json

- run: npm run lint # Run ESLint to ensure code quality

* This workflow automatically runs on every push and pull request to the main branch. It installs dependencies, runs tests, and performs code linting, with dependency caching to make builds faster over time.