Mega Bitbucket & Pipeline Documentation

Technical Documentation for Automated MERN Stack Deployment

Intended audience: New Developers, DevOps Engineers, Release Engineers, and Deployment Managers.

Before you start: You need access to **Bitbucket** (repository and workspace settings), **AWS** (EC2 security groups for the deployment server), and **SSH** to the server.

Overview

This document provides comprehensive guidance on the automated deployment process for our MERN stack applications using Bitbucket Pipelines and Docker containerization.

Previously, deployments required manual intervention including building applications, server access, and service restarts via PM2. Our automated pipeline infrastructure eliminates these manual steps, delivering deployments that are **faster, more reliable, and consistent** across environments.

Table of Contents

---

1. AWS Server & Security Group

This section is required before pipelines or Docker can deploy. The EC2 instance must allow inbound traffic on the ports used by SSH, NGINX, and your applications.

1.1 Purpose

- **SSH (port 22)** — Bitbucket Pipelines and engineers need to connect to the server to run deployment commands (`rsync`, `docker-compose`, etc.).
- **HTTP/HTTPS (ports 80, 443)** — NGINX listens here; without these, the public cannot reach your sites.
- **Application ports (3000, 3001, 5000)** — NGINX proxies requests to these container ports. For Mega, we typically use NGINX as the single entry point; you can restrict 3000, 3001, and 5000 to localhost or the VPC and only open 22, 80, and 443 to the internet.

1.2 Port reference (Mega deployment)

Port	Service	Why it is needed
22	SSH	Pipeline and manual SSH access for deployment and maintenance.
80	HTTP	NGINX; required for public websites.
443	HTTPS	Use when serving sites over SSL (recommended for production).
3000	Admin panel	Default port for the admin dashboard container; NGINX proxies `admin.*` here.
3001	Public frontend	Default port for the public frontend container; NGINX proxies main site here.
5000	Backend API	Default port for the backend API container; NGINX proxies `api.*` here.

1.3 Configuration steps

1. Log in to **AWS Console** and go to **EC2**.
2. In the left sidebar, click **Instances** and select the **target instance** (your deployment server).
3. Open the **Security** tab (below the instance details).
4. Click the **Security group** link (e.g. `sg-xxxxxxxx`).
5. In the security group, open the **Inbound rules** tab and click **Edit inbound rules**.
6. Add or confirm the following rules (use **Type: Custom TCP**; **Source: 0.0.0.0/0** for public access, or restrict **Source** to your IP or Bitbucket IPs for better security) per the table above.
7. Click **Save rules**.

Note (Mega): If frontend, admin, and backend are reached only via NGINX on the same server, you may only need **22**, **80**, and **443** open to the internet; 3000, 3001, and 5000 can be restricted to localhost or the private network.

---

2. Bitbucket Configuration

2.1 Workspace variables setup

Purpose

Workspace-level variables enable secure, reusable configuration of server credentials and environment values across multiple repositories and deployment pipelines.

Configuration steps

1. Access Bitbucket using provided administrative credentials.
2. Navigate to **Settings** (gear icon) → **Workspace settings**.
3. Select **Workspace variables** from the navigation menu.
4. Review existing variable configurations.
5. Add new variables as required:

- **SERVER_USER** — SSH username for server access
- **SERVER_IP** — Public IP address of target deployment server

6. Common existing variables include:
    - **SSH_MEGASERVER_USER** — Pipeline SSH authentication user (or use **DEPLOY_SERVER_USER** in pipeline YAML)
    - **SSH_MEGASERVER_IP** — Deployment server IP address (or use **DEPLOY_SERVER_IP** / **BACKEND_SERVER_IP** in pipeline YAML)

Note: When deploying to the same server instance, existing workspace variables can be reused across multiple projects. Use the same variable names in your `bitbucket-pipelines.yml` (e.g. `$DEPLOY_SERVER_USER`, `$DEPLOY_SERVER_IP`).

2.2 Pipeline activation

Once enabled, Bitbucket Pipelines automatically execute builds and deployments upon code commits to configured branches, provided a valid `bitbucket-pipelines.yml` configuration file exists in the repository.

Activation procedure

1. Open the target repository in Bitbucket.
2. Access **Repository settings** (gear icon or left sidebar).
3. Navigate to **Pipelines** → **Settings**.
4. Click **Enable Pipelines**.
5. Open **SSH keys** section under Pipeline settings.
6. In **Known hosts**, add the deployment server's public IP address.
7. Click **Fetch** to retrieve and validate the server host key.
8. Copy the generated **public SSH key** for server configuration (next section).

Example key format:

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQA.....................
```

2.3 Server SSH key configuration

To let Bitbucket Pipelines deploy to our server, we need to add the pipeline's public SSH key to the server’s authorized keys. This allows the pipeline to securely access the server during deployments.

Steps:

1. First, copy the public SSH key generated by the Bitbucket Pipeline (found in the Pipelines SSH settings).
2. Log in to the server through your usual remote access method (such as PuTTY or RDC). Once you are connected and have a terminal open, paste and run the following command (replace with the actual SSH key you copied):

   ```
   echo "<COPIED-SSH-KEY>" >> ~/.ssh/authorized_keys
   ```

   Example:

   ```
   echo "ssh-rsa AAAAB3NzaC1yc2EAAAADAQ
   A....................." >> ~/.ssh/authorized_keys
   ```

3. Run this command to restart the SSH service and make sure the new key is recognized:

   ```
   sudo service ssh restart
   ```

4. *(Optional)* If NGINX changes don't appear as expected, you can reload its configuration with:

   ```
   sudo systemctl reload nginx
   ```

These steps connect our Bitbucket Pipeline to the server for automated and secure deployments.

3. Pipeline Configuration

3.1 Pipeline architecture overview

The `bitbucket-pipelines.yml` file defines the complete deployment workflow, including:

- Build process execution
- Deployment automation steps
- SSH authentication and access
- Docker container orchestration

Separate pipeline configurations are maintained for frontend and backend deployments, each executing the following workflow:

1. Synchronize code to server via SSH
2. Build updated Docker images
3. Stop existing containers
4. Deploy new containers
5. Clean unused Docker resources

3.2 Frontend pipeline configuration

Automatically deploys the frontend application when code is pushed to the **master** branch.

Frontend pipeline YAML

```
image: node:20

pipelines:
  branches:
    master:
      - step:
          script:
            - apt-get update
            - apt-get install -y rsync
```

```
            - echo "Fetching the public IP of the Bitbucket P
ipeline runner..."
            - curl ifconfig.me
            - ssh-keyscan -H $DEPLOY_SERVER_IP >> ~/.ssh/know
n_hosts
            - cd $BITBUCKET_CLONE_DIR
            - rsync -r -v -e ssh . $DEPLOY_SERVER_USER@$DEPLO
Y_SERVER_IP:/var/www/html/frontend--delete-before --exclude
'.git'
            - ssh $DEPLOY_SERVER_USER@$DEPLOY_SERVER_IP 'cd /
var/www/html/frontend && docker-compose build'
            - ssh $DEPLOY_SERVER_USER@$DEPLOY_SERVER_IP 'cd /
var/www/html/frontend && docker-compose config -q || { echo
"Invalid Docker Compose configuration"; exit 1; }'
            - ssh $DEPLOY_SERVER_USER@$DEPLOY_SERVER_IP 'cd /
var/www/html/frontend && docker-compose down
            - ssh $DEPLOY_SERVER_USER@$DEPLOY_SERVER_IP 'cd /
var/www/html/frontend && docker-compose up -d'
            - ssh $DEPLOY_SERVER_USER@$DEPLOY_SERVER_IP 'dock
er system prune -af'
```

Pipeline step explanation (frontend)

Pipeline step	Purpose and function
`image: node:20`	Specifies Node.js 20 Docker image as the pipeline execution environment.
`branches → master`	Configures pipeline to trigger exclusively on commits to master branch.
`apt-get update`	Updates package repository lists within pipeline container.
`apt-get install rsync`	Installs rsync utility for efficient file synchronization.
`curl ifconfig.me`	Logs pipeline runner's public IP for SSH troubleshooting.
`ssh-keyscan`	Adds server to known hosts, preventing SSH confirmation prompts.
`BITBUCKET_CLONE_DIR`	System variable containing repository clone directory path.

Pipeline step	Purpose and function
`rsync`	Synchronizes frontend code to server, removing obsolete files.
`docker-compose build`	Builds updated Docker images on deployment server.
`docker-compose config -q`	Validates Docker Compose configuration syntax before deployment.
`docker-compose down`	Stops and removes currently running containers.
`docker-compose up -d`	Starts updated containers in detached (background) mode.
`docker system prune -af`	Removes unused Docker images, containers, and build cache.

3.3 Backend pipeline configuration

Deploys backend API services when code is pushed to the designated backend deployment branch (**backend-deploy**).

Backend pipeline YAML

```
image: node:20

pipelines:
  branches:
    backend-deploy:
      - step:
          caches:
            - node
          script:
            - apt-get update
            - apt-get install -y rsync
            - echo "Fetching the public IP of the Bitbucket P
ipeline runner..."
            - curl ifconfig.me
            - ssh-keyscan -H $BACKEND_SERVER_IP >> ~/.ssh/kno
wn_hosts
            - cd $BITBUCKET_CLONE_DIR
            - rsync -r -v -e ssh . $DEPLOY_SERVER_USER@$BACKE
ND_SERVER_IP:/var/www/html/backend --delete-before --exclude
'.git'
```

```
            - ssh $DEPLOY_SERVER_USER@$BACKEND_SERVER_IP 'rm
/var/www/html/backend/.env && mv /var/www/html/backend/.env.s
tage /var/www/html/backend/.env'
            - ssh $DEPLOY_SERVER_USER@$BACKEND_SERVER_IP cd /
var/www/html/backend && docker-compose build'
            - ssh $DEPLOY_SERVER_USER@$BACKEND_SERVER_IP 'cd
/var/www/html/backend && docker-compose config -q || { echo
"Invalid Docker Compose configuration"; exit 1; }'
            - ssh $DEPLOY_SERVER_USER@$BACKEND_SERVER_IP 'cd
/var/www/html/backend && docker-compose down'
            - ssh $DEPLOY_SERVER_USER@$BACKEND_SERVER_IP 'cd
/var/www/html/backend && docker-compose up -d'
            - ssh $DEPLOY_SERVER_USER@$BACKEND_SERVER_IP 'doc
ker system prune -af'
```

Pipeline step explanation (backend)

Pipeline step	Purpose and function
`branches → backend-deploy`	Triggers pipeline exclusively on backend deployment branch.
`caches: node`	Caches Node.js modules to accelerate pipeline execution.
`rsync`	Synchronizes backend source code to deployment server.
`.env` replacement	Replaces environment configuration file with environment-specific version (`.env.stage` → `.env`).
`docker-compose build`	Builds backend application Docker images.
`docker-compose config -q`	Validates Docker Compose configuration integrity.
`docker-compose down`	Stops existing backend containers gracefully.
`docker-compose up -d`	Deploys updated backend containers in detached mode.
`docker system prune -af`	Reclaims disk space by removing unused Docker resources.

Required environment variables

These variables must be configured in Bitbucket Workspace Variables:

Variable name	Description
`DEPLOY_SERVER_USER`	Linux user account for SSH server authentication.
`DEPLOY_SERVER_IP`	Public IP address of frontend deployment server.
`BACKEND_SERVER_IP`	Public IP address of backend deployment server.
`BITBUCKET_CLONE_DIR`	Default repository clone directory (system-provided).

Implementation benefits

- Automated deployment triggered by branch commits
- Elimination of manual SSH access, build processes, and service restarts
- Docker ensures environment consistency across deployments
- Complete deployment lifecycle managed by pipeline automation

---

4. Docker Containerization

4.1 Docker overview

Docker containerization provides consistent, isolated application environments, ensuring deployment reliability across development, staging, and production.

Traditional deployment methods require manual installation of Node.js, application builds, PM2 configuration, and service management. Docker eliminates these requirements by packaging complete application environments into portable containers.

Key benefits

- Accelerated deployment cycles
- Environment consistency and reproducibility
- Automated container restart and recovery
- Elimination of manual server configuration

4.2 Core Docker concepts

Concept	Definition
Image	Immutable template containing application code, runtime, and dependencies.
Container	Running instance of a Docker image with isolated filesystem and processes.
Dockerfile	Text file containing instructions for building Docker images.
Docker Compose	Tool for defining and orchestrating multi-container applications.

Note: Docker and Node.js are pre-installed by the infrastructure team. No additional installation is required.

4.3 Backend API Dockerfile

Directory structure

```
Backend/
├── Dockerfile
├── package.json
├── src/
└── build/
```

Dockerfile configuration

```
FROM node:18
WORKDIR /app
COPY package.json .
RUN npm install --only=production
COPY . ./
ARG MONGODB_DB_TOOLS_URL=https://fastdl.mongodb.org/tools/db/
mongodb-database-tools-ubuntu1804-x86_64-100.5.0.deb
ADD ${MONGODB_DB_TOOLS_URL} /mongodb-tools.deb
RUN apt install /mongodb-tools.deb
RUN rm /mongodb-tools.deb
EXPOSE 5000
ENV NODE_ENV=production
```

Dockerfile instruction explanation

Instruction	Purpose
`FROM node:18`	Establishes Node.js 18 as base image for production environment.
`WORKDIR /app`	Sets /app as the container working directory.
`COPY package.json .`	Copies dependency manifest for installation.
`RUN npm install --only=production`	Installs production dependencies, excluding development packages.
`COPY . ./`	Copies complete backend application source code.
`ARG MONGODB_DB_TOOLS_URL`	Defines build argument for MongoDB tools download URL.
`ADD ${MONGODB_DB_TOOLS_URL}`	Downloads MongoDB database tools package.
`RUN apt install`	Installs MongoDB utilities within container.
`RUN rm`	Removes installation package to minimize image size.
`EXPOSE 5000`	Documents container port (must be unique per application).
`ENV NODE_ENV=production`	Sets Node.js environment to production mode.

4.4 Admin panel Dockerfile

Configuration for the internal administrative dashboard interface.

```
FROM node:20-alpine
WORKDIR /app
COPY package.json .
RUN npm install --force --only=production
COPY . ./
EXPOSE 3000
RUN npm run build
```

Instruction	Purpose
`FROM node:20-alpine`	Lightweight Node.js 20 image.
`WORKDIR /app`	Sets /app as working directory.

Instruction	Purpose
`COPY package.json .` / `RUN npm install --force --only=production`	Installs production dependencies; `--force` can resolve peer dependency conflicts.
`COPY . ./`	Copies admin panel source code.
`EXPOSE 3000`	Admin panel port (unique per application).
`RUN npm run build`	Builds the admin frontend for production.

4.5 Frontend application Dockerfile

Configuration for the public-facing web application.

```
FROM node:20-alpine
WORKDIR /web
COPY package.json .
RUN mkdir -p /web/lib
COPY lib /web/lib
RUN npm install
COPY . ./
EXPOSE 3001
RUN npm run build
```

Instruction	Purpose
`FROM node:20-alpine`	Lightweight Node.js 20 image.
`WORKDIR /web`	Sets /web as working directory.
`COPY package.json .` / `RUN mkdir -p /web/lib` / `COPY lib /web/lib`	Dependencies and shared lib (Mega-specific).
`RUN npm install`	Installs dependencies (including dev for build).
`COPY . ./`	Copies frontend source code.
`EXPOSE 3001`	Public frontend port (unique per application).
`RUN npm run build`	Builds the static/served frontend for production.

4.6 Docker Compose configuration

The `docker-compose.yml` file resides in the project root directory and orchestrates multiple service containers. Separate compose configurations are maintained for backend API, admin panel, and frontend applications.

4.7 Build artifact exclusion (`.dockerignore`)

The `.dockerignore` file prevents unnecessary files from being included in Docker images and ensures sensitive files are excluded from containers while reducing image size.

Example:

```
.env.local
node_modules
.git
```

---

5. NGINX Virtual Host Configuration

5.1 Overview

Following Docker and pipeline setup, NGINX reverse proxy configuration enables domain-based application access.

Prerequisites

- NGINX web server installed on deployment server
- Administrative access to NGINX configuration directories
- DNS records configured for application domains

Note: If NGINX is not installed, contact the infrastructure team for installation.

5.2 Configuration procedure

1. Navigate to NGINX configuration directory: `/etc/nginx/sites-available`
2. Create configuration file named after your domain (e.g. `example.mega.com.conf`)

3. Add separate server blocks for backend, frontend, and (optionally) admin services
4. Update port numbers, domain names, and paths per project requirements

Example configuration (backend + frontend + admin)

Replace `example.mega.com` with your domain and ensure ports match your Docker Compose exposed ports (e.g. 5000, 3001, 3000).

```
# Backend API
server {
    listen 80;
    server_name api.example.mega.com;
    location / {
        proxy_pass <http://localhost:5000>;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}

# Public frontend
server {
    listen 80;
    server_name example.mega.com www.example.mega.com;
    location / {
        proxy_pass <http://localhost:3001>;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

```
# Admin panel (optional)
server {
    listen 80;
    server_name admin.example.mega.com;
    location / {
        proxy_pass <http://localhost:3000>;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

5.3 Configuration activation

Step 1 — Validate configuration syntax:

```
sudo nginx -t
```

Step 2 — Enable site configuration:

```
sudo ln -s /etc/nginx/sites-available/example.mega.com.conf /
etc/nginx/sites-enabled/
sudo nginx -t
```

Step 3 — Reload NGINX service:

```
sudo systemctl reload nginx
```

Step 4 — Restart NGINX (if required):

```
sudo systemctl restart nginx
```

5.4 Deployment complete

Upon successful configuration:

- Code commits to configured branches trigger automatic deployment
- No manual build or restart procedures required
- Configuration is a one-time setup per application

Your application deployment pipeline is now fully automated.

---

6. Troubleshooting Common Issues

6.1 SSH authentication failures

Symptoms	`Permission denied (publickey)` error message; `Host key verification failed` error; pipeline execution stops at SSH or rsync commands.
Possible causes	SSH key not properly added to deployment server; incorrect `DEPLOY_SERVER_USER` or IP address configuration; server IP not added to Bitbucket Known Hosts.
Resolution steps	1. Verify Workspace Variables: `DEPLOY_SERVER_USER`, `DEPLOY_SERVER_IP` / `BACKEND_SERVER_IP`. 2. In Bitbucket: Repository Settings → Pipelines → SSH Keys; add server IP under Known Hosts. 3. On the server, verify SSH key exists: `cat ~/.ssh/authorized_keys`. 4. Re-add the pipeline SSH key if missing.

6.2 Pipeline success but application not updated

Symptoms	Pipeline shows successful completion; website or API continues displaying previous version.
Possible causes	Docker container not rebuilt properly; previous container still running; code synchronization incomplete.
Resolution steps	Execute the following manually on the server to verify: `cd /var/www/html/<project-folder>`, then `docker-compose down`, `docker-compose build --no-cache`, `docker-compose up -d`. Then refresh the application in your browser.

6.3 Docker build failures

Symptoms	Pipeline fails at `docker-compose build` step; errors related to `npm install`, missing files, or permissions.
Possible causes	Missing or incorrect `package.json`; wrong Node version in Dockerfile; corrupted images or cache.
Resolution steps	1. SSH into the deployment server. 2. Navigate to project directory and attempt manual build: `cd /var/www/html/<project-folder>`, `docker-compose build`. 3. Review the exact error message. 4. Common fixes: verify `package.json` exists in project root; confirm correct Node.js version in Dockerfile; remove corrupted images: `docker system prune -af`.

6.4 NGINX service issues

Symptoms	Domain not accessible; 502 Bad Gateway error; connection refused errors.
Possible causes	NGINX configuration error; containers not running; port mismatch between docker-compose and NGINX.
Resolution steps	1. Test NGINX configuration syntax: `sudo nginx -t`. 2. Verify containers are running: `docker ps`. 3. Verify port consistency between `docker-compose.yml` port mappings and NGINX `proxy_pass` configuration. 4. Reload NGINX service: `sudo systemctl reload nginx`.

6.5 Disk space exhaustion

Symptoms	Docker build fails unexpectedly; errors containing "no space left on device".
Possible causes	Unused images, stopped containers, and build cache consuming disk.
Resolution steps	Execute on the server to clean Docker resources: `docker system prune -af`. This removes unused Docker images, stopped containers, build cache, and orphaned volumes.

6.6 Environment variables not applied (backend)

Symptoms	Backend API crashes on startup; database connection failures; application functions locally but fails in production.
Possible causes	`.env` missing or not replaced by pipeline (e.g. `.env.stage` not moved to `.env`).
Resolution steps	1. Verify environment file handling: `cd /var/www/html/backend`, `ls -la`. 2. Ensure `.env.stage` file exists. 3. Verify pipeline successfully replaces environment file: `rm .env && mv .env.stage .env`. 4. Restart containers to apply changes: `docker-compose down`, `docker-compose up -d`.

7. Additional Resources

- Bitbucket Pipelines Documentation
- Docker Compose Documentation
- NGINX Documentation
- AWS EC2 User Guide

Final Notes

- **Test thoroughly**: Always validate your changes in a staging or non-production environment prior to any production deployment to catch potential issues early.
- **Maintain dependencies**: Regularly update all pipeline, server, and container dependencies. This ensures you benefit from the latest security patches, features, and stability improvements.
- **Manage Docker resources**: Frequently remove unused Docker images, containers, volumes, and caches (e.g., via `docker system prune -af`) to maintain plenty of free disk space and avoid unexpected build failures.
- **Review pipeline logs**: Thoroughly inspect Bitbucket Pipeline logs after each deployment. Promptly address any warnings, errors, or failed steps.