Documentation (DevOps-URL-Shortener)

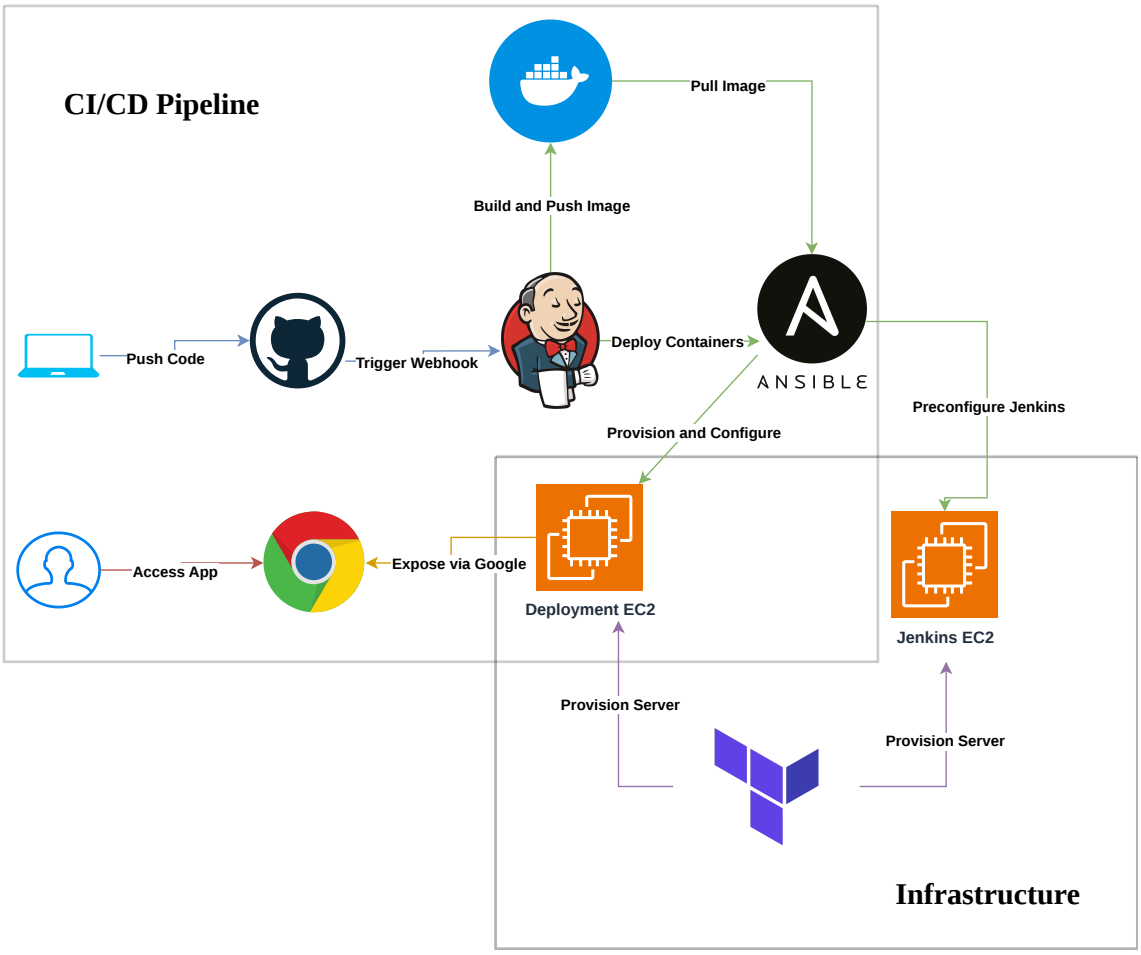


Table of Contents

1. URL Shortener DevOps Project

The **URL Shortener DevOps Project** demonstrates how to build and deploy a modern web application using industry-standard DevOps practices. The goal of this project is to provide a complete, end-to-end pipeline that automates infrastructure provisioning, configuration management, containerization, and continuous integration/continuous delivery (CI/CD).

At the core of this project is a **URL Shortener application**, which is deployed onto **AWS EC2 instances**. The deployment pipeline leverages:

- **Terraform** for provisioning infrastructure as code (IaC)
- **Ansible** for automating configuration and deployment
- **Docker** for packaging the application into lightweight containers
- **Jenkins** for orchestrating the CI/CD pipeline

This project highlights how different tools can work together to enable a scalable, automated, and reliable DevOps workflow.

1.1 Objectives

The main objectives of this project are:

- To design and implement a **production-ready DevOps pipeline**.
- To demonstrate the integration of **IaC, configuration management, and containerization**.

- To automate application build, deployment, and updates using **CI/CD pipelines**.
- To apply **best practices in security and scalability** on cloud infrastructure.
- To provide **real-world experience** with tools widely used in the DevOps ecosystem.

2. System Requirements and Technology Stack

2.1 Functional and Operational Requirements

- **Functional Requirements**
  - The system shall allow users to shorten URLs and redirect them correctly.
  - The CI/CD pipeline shall automatically build, push and deploy the application on code push.
  - The infrastructure shall be provisioned automatically using Infrastructure as Code (Terraform).
- **Operational Requirements**
  - The system shall be deployable on AWS EC2 instances.
  - The system shall support monitoring and alerting (Prometheus + Grafana).
  - The system shall ensure secure access via security groups and role-based permissions.
  - The system shall be scalable to handle increased traffic.

2.2 Technology Stack

- **Cloud Provider:** AWS (EC2, VPC, Security Groups, Elastic IP)
- **Infrastructure as Code (IaC):** Terraform
- **Configuration Management:** Ansible
- **Containerization:** Docker, Docker Compose
- **CI/CD Orchestration:** Jenkins (Blue Ocean UI)
- **Version Control:** GitHub

3. Setup Instructions

3.1 Cloud Service Credentials

Before running any commands to set up infrastructure, you should provide your AWS credentials and store them as environment variables.

This is the most secure method for authenticating Terraform, Ansible, and other tools with AWS.

For Windows (PowerShell)

```
setx AWS_ACCESS_KEY_ID "your-access-key-id"
setx AWS_SECRET_ACCESS_KEY "your-secret-access-key"
setx AWS_DEFAULT_REGION "your-region"
```

⚠️ After running `setx`, restart your terminal so the variables are available.

For Linux / macOS (Bash)

```
export AWS_ACCESS_KEY_ID="your-access-key-id"
export AWS_SECRET_ACCESS_KEY="your-secret-access-key"
export AWS_DEFAULT_REGION="your-region"
```

💡 You can add these lines to your `~/.bashrc` or `~/.zshrc` file to make them persistent across sessions.

3.2 Clone the Repository

Once your AWS credentials are configured, the next step is to clone the project repository to your local machine.

This ensures you have all the necessary Terraform, Ansible, and application files available for setup.

```
git clone https://github.com/4ykh4nCyb3r/url-shortener-devops.git
cd url-shortener-devops
```

💡 Tip: Make sure you have **Git** installed on your system.
You can verify by running `git --version`.

3.3 Spin Up AWS EC2 instances

You should create `SSH` key pairs and allocate `Elastic IP` in `AWS`.

🔑 **Creating SSH Key Pairs in AWS Console**

1. Go to the **AWS Management Console** → **EC2** service.
2. In the left menu, click **Key Pairs**.
3. Click **Create key pair**.
4. Enter a **name** (e.g., `aws-key`).
5. Choose **Key pair type**:
   - `RSA` (most common) or `ED25519` (newer, smaller).
6. Choose **Private key file format**:
   - `.pem` (for Linux/macOS, Ansible, Terraform)
   - `.ppk` (for Windows PuTTY)

⚠️ For the current project chose `.pem`.

1. Click **Create key pair**.

The private key file will be **downloaded automatically** — keep it safe, you won't be able to download it again.

🌐 **How to Allocate Elastic IP in AWS**

1. Go to **AWS Console → EC2 → Elastic IPs**.
2. Click **Allocate Elastic IP address**.
3. Choose **Amazon's pool of IPv4 addresses**.
4. Click **Allocate**.

>> Resources Created

- **Jenkins Server (EC2 instance)**
- **Deployment Server (EC2 instance)**
- **Security Groups** for controlled access
- **Elastic IP** associated with the Deployment Server

⚠️ **Important:**

1. After Terraform finishes, make note of the **Deployment Server Public IP Address**. You will need this IP later to access the application in your browser.
2. Put Deployment Server Private IP address in ansible-deplyment/inventory.ini file.

This project uses `Terraform` to automatically create and configure the necessary AWS resources.

```
cd infra

# initialize Terraform
terraform init

# review the execution plan
terraform plan

# apply the configuration to spin up the infrastructure
terraform apply
```

3.4 Configure the Jenkins Server using Ansible

After provisioning the infrastructure, the next step is to install and configure **Jenkins** on the Jenkins EC2 instance.

This project uses **Ansible** to automate the setup process, ensuring Jenkins is ready with the required plugins and settings.

🐧 Installing Ubuntu WSL on Windows (for Ansible users on Windows)

Since Ansible is a Linux-native tool, Windows users should run it inside **Windows Subsystem for Linux (WSL)**.

Here's how to install and run Ubuntu WSL:

1. **Enable WSL and Install Ubuntu**
   - Open **PowerShell as Administrator** and run:

     ```
     wsl --install -d Ubuntu
     ```

   - Restart your computer when prompted.
   - On first launch, you'll be asked to create a **Linux username and password**.

   > 💡 You can check available distributions with:
   >
   > ```
   > wsl --list --online
   > ```

2. **Launch Ubuntu**
   - Open the **Start Menu** → search for **Ubuntu** → launch it.
   - You'll now have a Linux terminal running inside Windows.
3. **Update Packages**

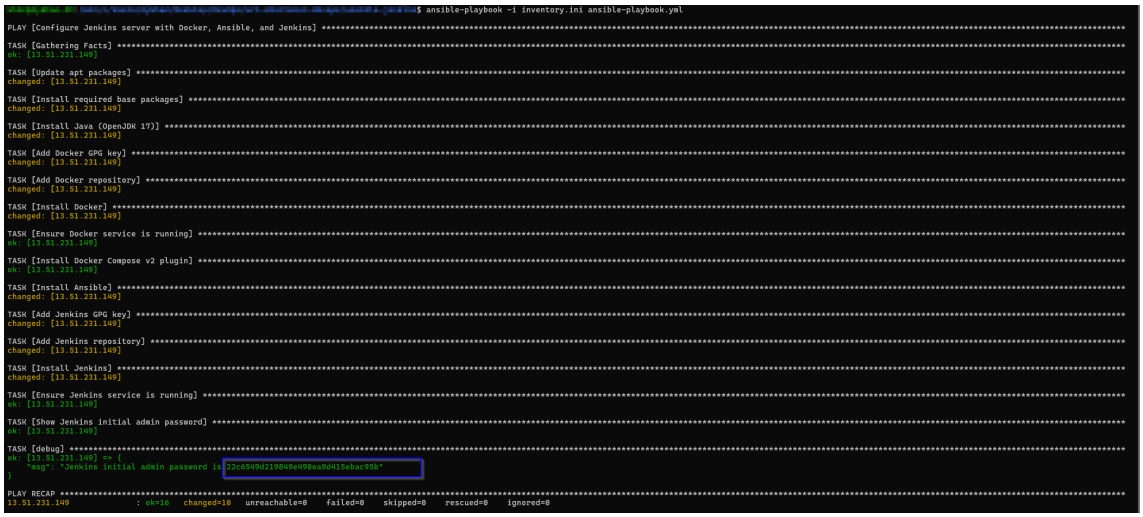
   ```
   sudo apt update && sudo apt upgrade -y
   ```

4. **Install Ansible**

   ```
   sudo apt install ansible -y
   ```

If you are Linux user then navigate to ansible-jenkins directory and run the Ansible:

```
cd ansible-jenkins

ansible-playbook -i inventory.ini ansible-playbook.yml
```



Once the playbook completes, Jenkins will be installed and accessible on **port 8080** of your Jenkins server's public IP. You can verify by opening your browser and navigating to:

```
http://<jenkins-server-elastic-public-ip>:8080
```

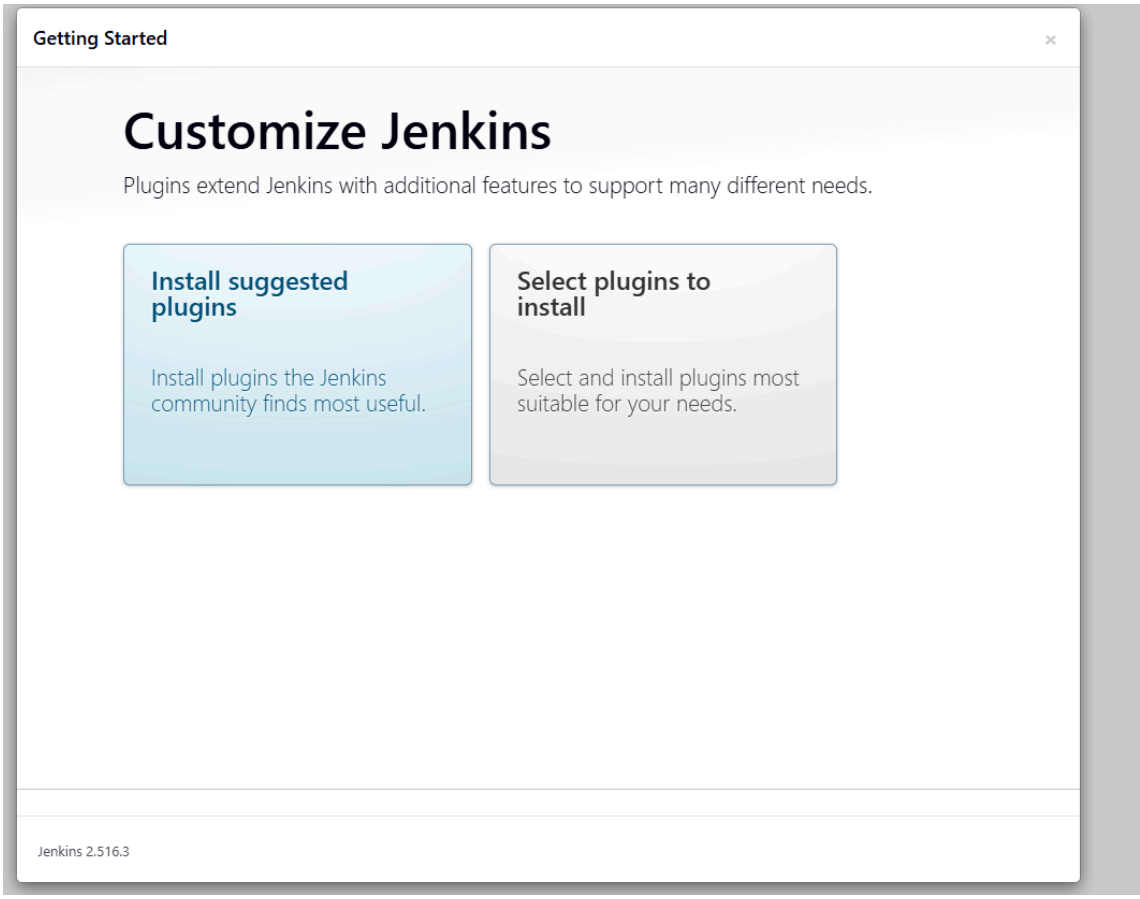
In the login page paste the Initial Admin Password provided after running above command:

⚠️ Keep your `inventory.ini` file updated with the correct elastic public IPs of your servers to avoid connection issues.

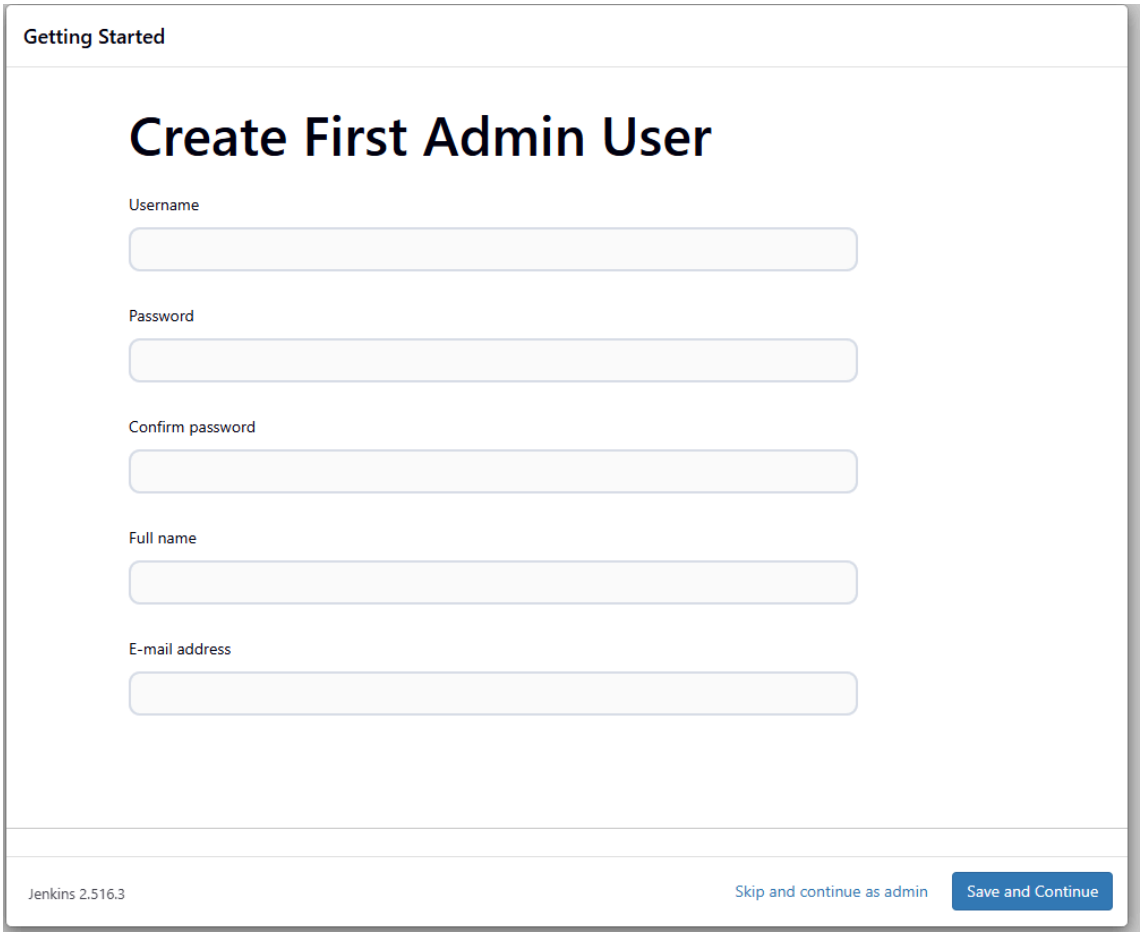
4. Manually Configure Jenkins

Once you have provided the Jenkins initial admin password you need to:

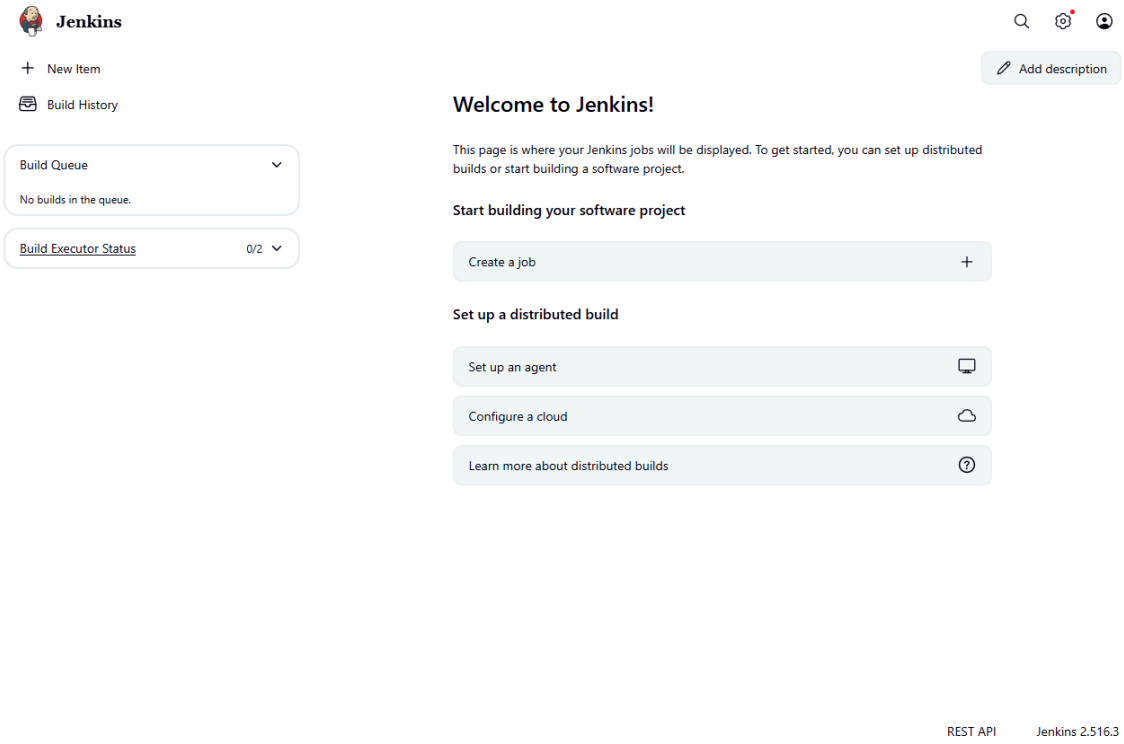
1. **Select and install suggested plugins:**



2. **After installing plugins create a user:**



If you have configured everything correctly you should see Jenkins main UI:



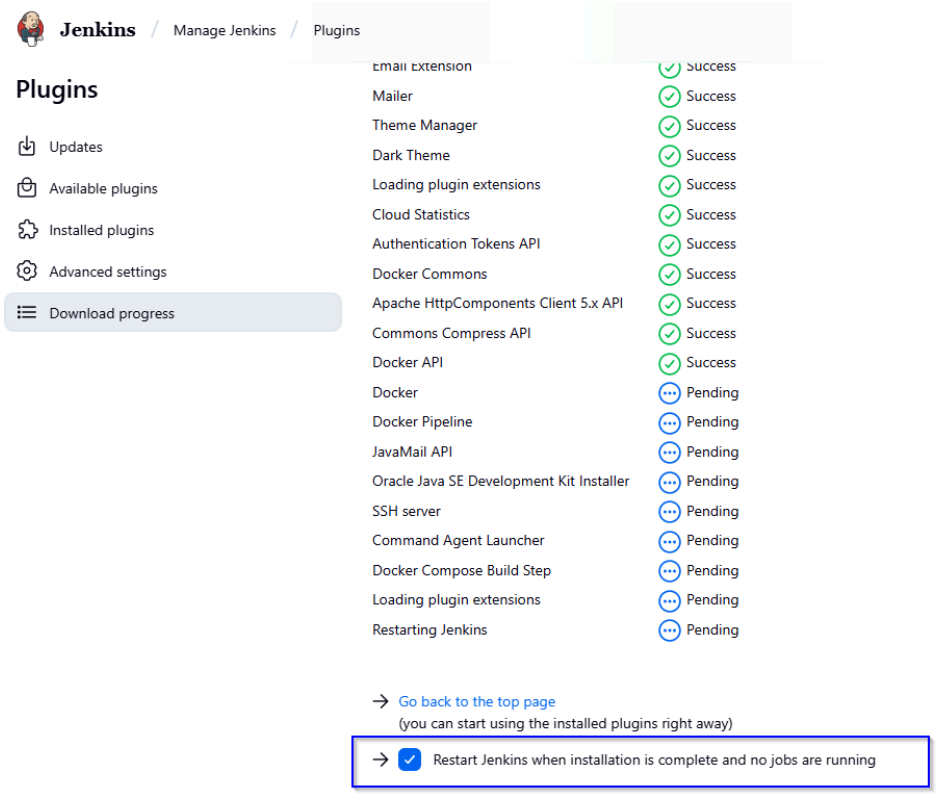
3. **Install other required plugins**

Go to:
Manage Jenkins → Plugins → Available Plugins

Search and install:

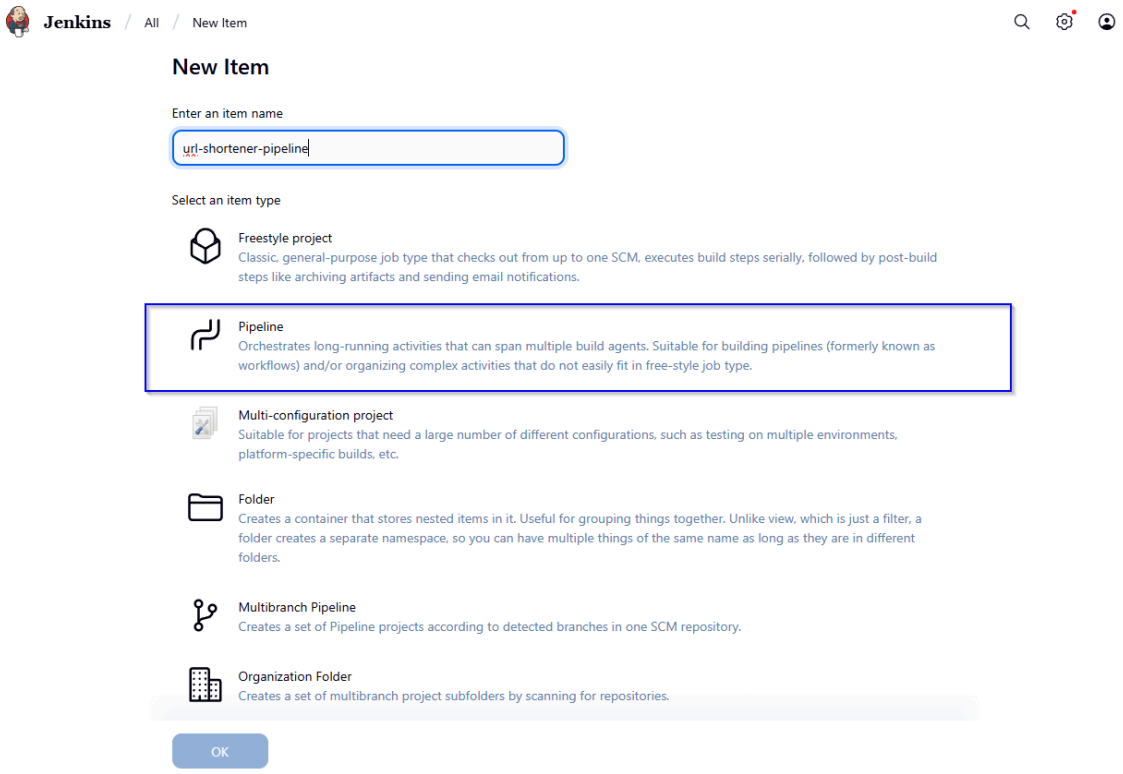
- ✅ Docker
- ✅ Docker Pipeline
- ✅ Docker Compose Build Step

When installing plugins check the box to restart the Jenkins server once all plugins are successfully installed:

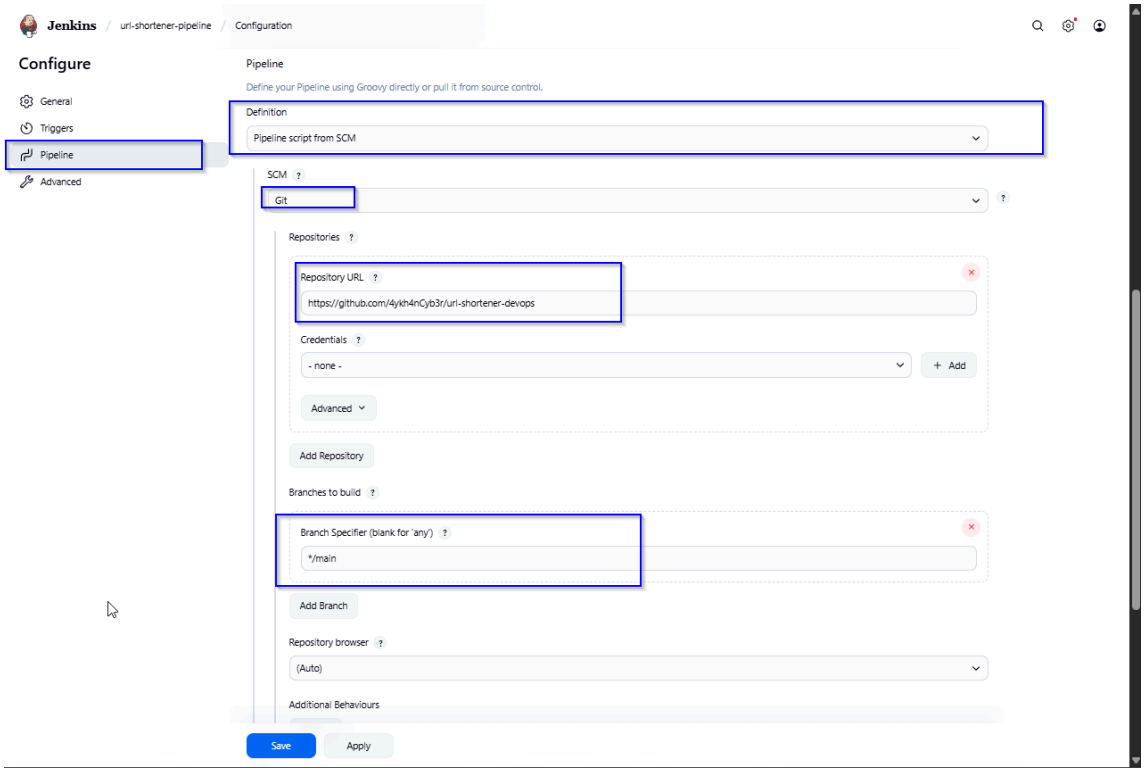


4. **Create a New Pipeline Job**

Go to **New Item** → name it (e.g., `url-shortener-pipeline`) → select **Pipeline** → OK.



Go to `Pipeline` and select the definition as `Pipeline script from SCM` as you already have `Jenkinsfile` in the repo, chose `Git` as Source Code Management (SCM) platform and select a branch.



5. **Configure a GitHub webhook**

To automatically trigger the pipeline when code is pushed to your repository, you need to configure a GitHub webhook. On the same page go to `Triggers` and select `GitHub hook trigger for GITScm polling`.

6. **Add Credentials in Jenkins**

1. **DockerHub Credentials (** `dockerhub-creds` **)**

These will be used by your pipeline to authenticate with DockerHub for pushing and pulling images.

1. In Jenkins UI, go to **Manage Jenkins → Credentials → System → Global credentials (unrestricted)**.
2. Click **Add Credentials**.
3. Select **Kind: Username with password**.
4. Enter:
   - **Username**: your DockerHub username
   - **Password**: your DockerHub password or access token
   - **ID**: `dockerhub-creds` (important — this is how the Jenkinsfile will reference it)
   - **Description**: e.g., *DockerHub account for CI/CD*
5. Click **OK**.

2. **Ansible SSH Key (** `ansible_ssh_key` **)**

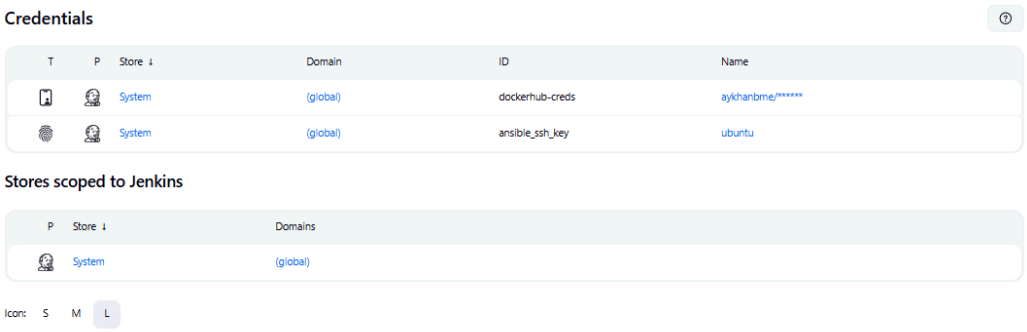
This will allow Jenkins to SSH into the deployment server and run Ansible playbooks.

1. In Jenkins UI, go to **Manage Jenkins → Credentials → System → Global credentials (unrestricted)**.
2. Click **Add Credentials**.
3. Select **Kind: SSH Username with private key**.
4. Enter:
   - **Username**: the SSH user for your deployment server (e.g., `ubuntu` or `ec2-user`)

- **Private Key**: paste the contents of your `.pem` key file (the one you downloaded when creating the AWS key pair)
- **ID**: `ansible_ssh_key`
- **Description**: e.g., *SSH key for Ansible to connect to deployment host*

5. Click **OK**.

Once credentials are created you shell see them under `Manage Jenkins > Credentials`:



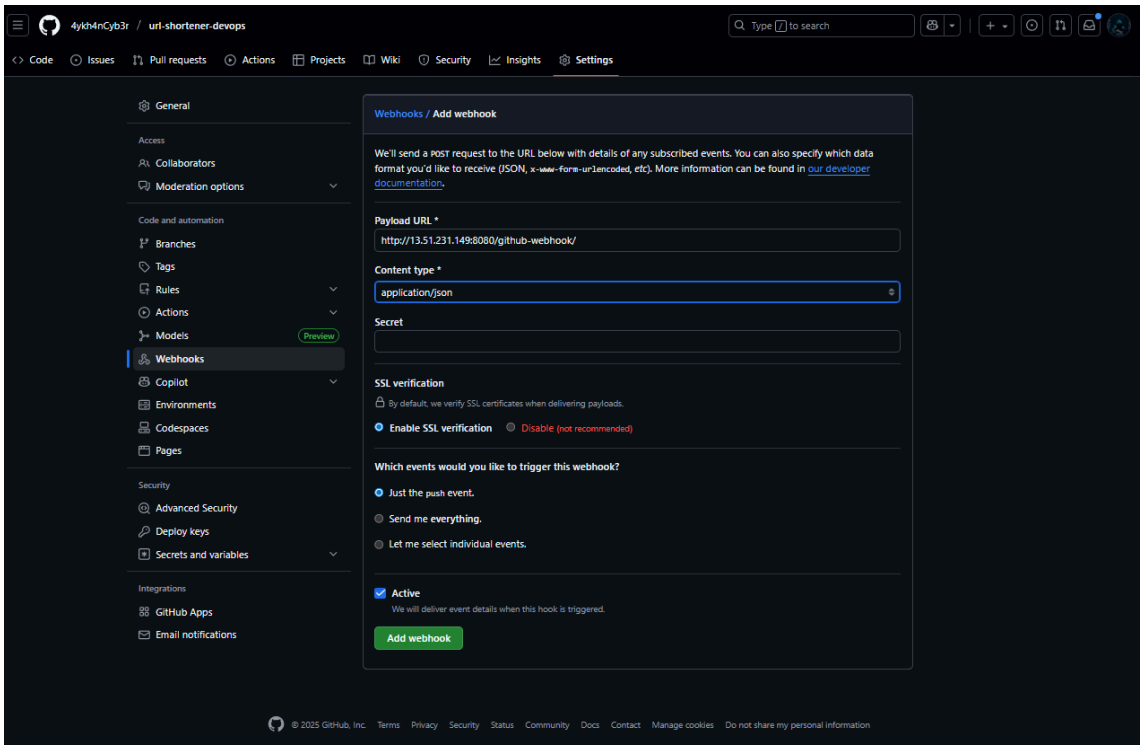
Once these steps are complete, Jenkins will be ready to build and deploy your application.

5. Add GitHub WebHook

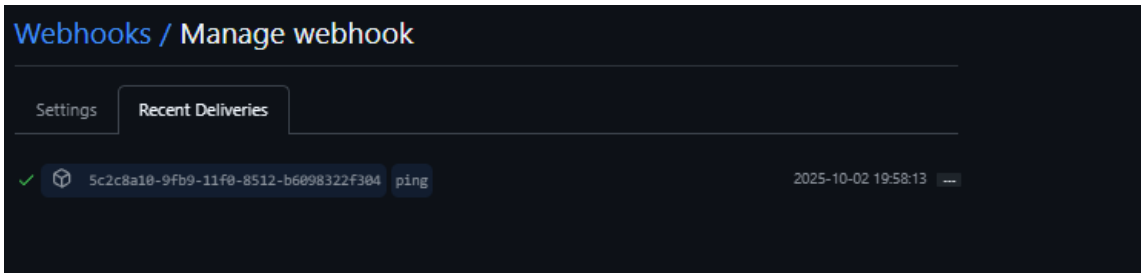
1. Open your repository → **Settings → Webhooks**.
2. Click **Add webhook**.
3. Fill in:
   - **Payload URL**:

     ```
     http://<your-elastic-ip>:8080/github-webhook/
     ```

     (replace `<your-elastic-ip>` with your Jenkins server's Elastic IP)
   - **Content type**: `application/json`
   - **Events**: Select **Just the push event** (or "Send me everything" if you want broader triggers)
4. Click **Add webhook**.



It first will check if it can reach the webhook endpoint, if yes, it will show green when you click on the webhook you created and go to `Recent Deliveries`:



6. Triggering the Pipeline

To kick off the Jenkins pipeline, commit and push changes to the branch your pipeline monitors (for example, `main` or `develop`).

When you do this:

- GitHub will send a notification to Jenkins via the webhook you set up.
- Jenkins will automatically launch the pipeline's build and deployment process.
- The stages defined in your Jenkinsfile (such as build, push, and deploy) will be executed in order.

After the pipeline finishes successfully, the latest version of your application will be deployed.

4. CI/CD Workflow

1. Developer pushes code → GitHub
2. Jenkins triggers build
3. Docker image built & pushed to DockerHub
4. Ansible deploys the container to AWS EC2
5. Application is live ✅

```
flowchart TD
    Dev[Developer] -->|Push Code| GitHub
    GitHub -->|Webhook| Jenkins
    Jenkins -->|Build & Push| DockerHub
    Jenkins -->|Deploy| Ansible
    Ansible --> EC2[Deployment Server]
    EC2 -->|Serve App| User[Browser]
```

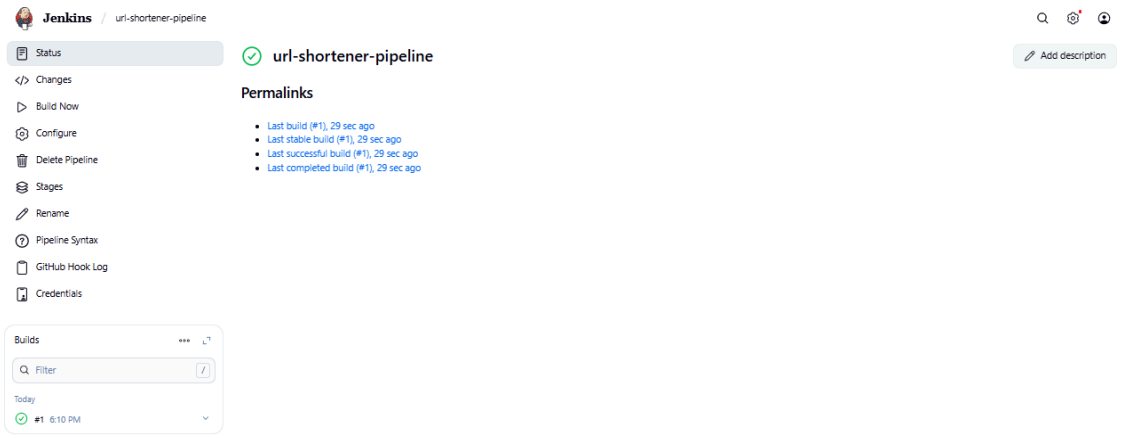
5. Demonstration: Successful Build & Deployment

Modify the application's JavaScript source code and push the changes to the remote repository:

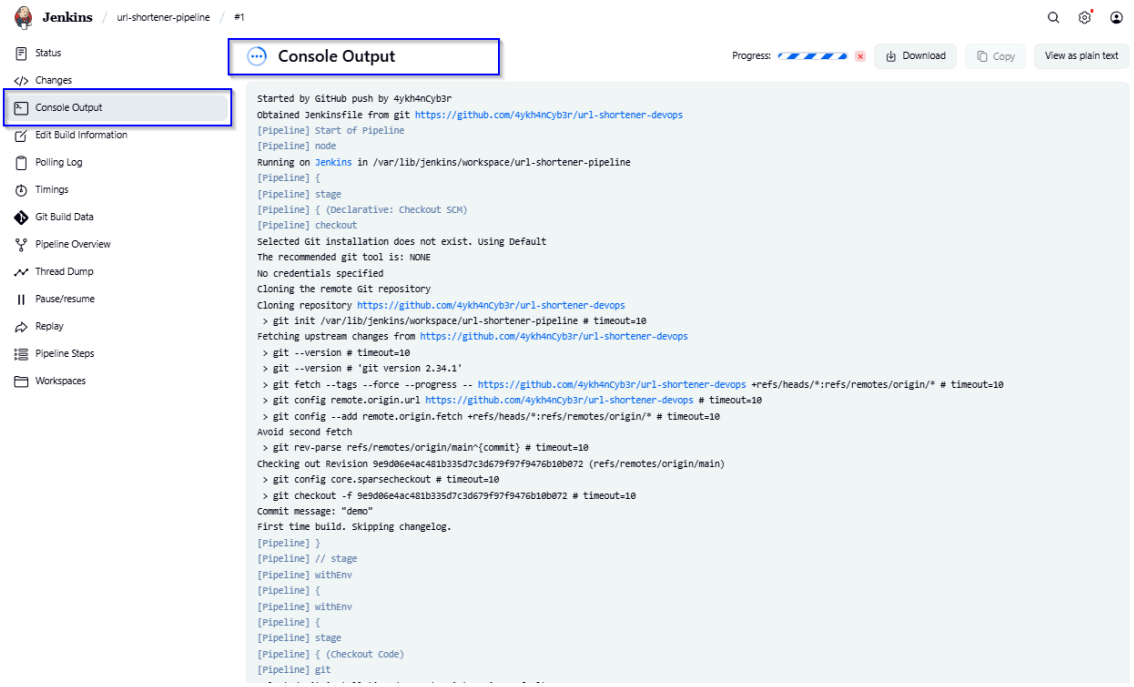
```
const express = require("express");
const mongoose = require("mongoose"); //interact with MongoDB database
const shortid = require("shortid");
const path = require("path");
const Url = require("./models/Url");

const app = express();
app.use(express.json());
app.use(express.static(path.join(__dirname, "public"))); // serve frontend

// Connect to MongoDB
mongoose.connect("mongodb://mongo:27017/urlshortener", {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});
// change the MongoDB connection string to use the service name 'mongo' as hostname
// POST /shorten - create short URL
app.post("/shorten", async (req, res) => {
  const { originalUrl } = req.body;
  if (!originalUrl) return res.status(400).json({ error: "URL is required" });

  const shortUrl = shortid.generate();

  const newUrl = new Url({ originalUrl, shortUrl });
  await newUrl.save();

  res.json({ shortUrl: `/${shortUrl}` }); // frontend will handle base URL
});

// GET /:shortUrl - redirect to original URL
app.get("/:shortUrl", async (req, res) => {
  const { shortUrl } = req.params;
  const url = await Url.findOne({ shortUrl });

  if (!url) return res.status(404).send("URL not found");

  res.redirect(url.originalUrl);
});

app.listen(3000, () => console.log("🚀 Server running on port 3000"));

```

```
# Stage all modified and new files for commit
git add .

# Commit the staged changes with a descriptive message
git commit -m "demo"

# Push the committed changes to the 'main' branch on the remote repository
git push origin main
```

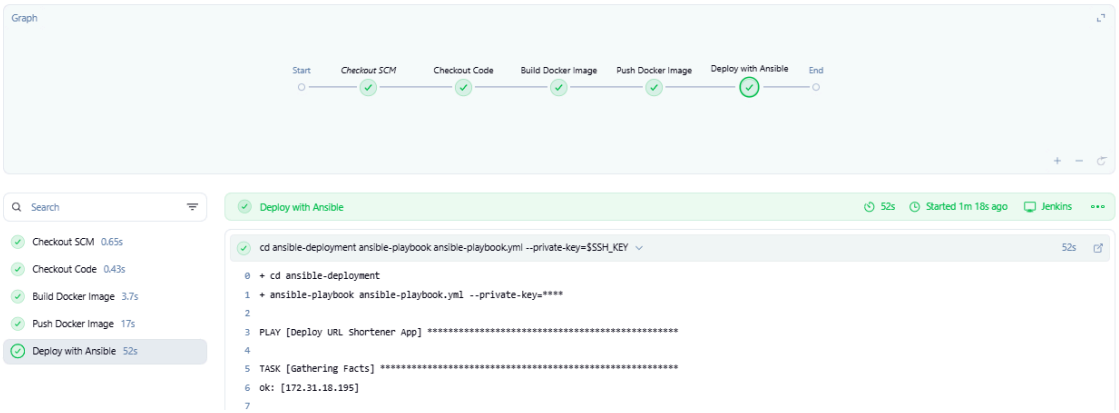
Go to Jenkins:



The pipeline executed successfully. We can view the command line outputs checking `Console Output`:



You can see stages that `Jenkins` went through and that were also specified in `Jenkinsfilee` navigating to `Pipeline Overview` and analyze which of them were completed successfully:



Take the **public IP address of your deployment server** that you recorded during infrastructure setup. Open a web browser and enter:

```
http://<deployment-server-public-ip>:3000
```

This URL will load the deployed application in your browser.

6. Future Enhancements

Add Automated Tests in the Pipeline

- Integrate unit and integration tests into the Jenkins pipeline to automatically validate code changes. This ensures only stable, working code is deployed to production.

Configure Monitoring with Prometheus + Grafana

- Set up Prometheus to collect metrics from your application and infrastructure. Use Grafana dashboards to visualize performance and receive alerts on critical issues.

Enable HTTPS with Nginx Reverse Proxy + Certbot

- Deploy Nginx as a reverse proxy in front of the application and secure it with Let's Encrypt certificates via Certbot. This provides encrypted communication and automatic certificate renewal.

Add Auto-Scaling with AWS Auto Scaling Groups (ASG)

- Configure an Auto Scaling Group to dynamically adjust the number of EC2 instances based on traffic or resource usage. This improves availability while optimizing costs.