ChatGPT

SOP: Jenkins CI/CD Setup on AWS EC2 (Ubuntu 24.04) with Docker

1. Purpose

This Standard Operating Procedure (SOP) outlines the step-by-step process to set up a Jenkins CI/CD system using two AWS EC2 instances (t3.micro, Ubuntu 24.04 LTS) with Docker. We will configure one instance as the **Jenkins Master** (controller) and another as the **Jenkins Agent** (worker). The Jenkins Master will orchestrate the pipeline (listening for GitHub webhooks and triggering jobs), while the Jenkins Agent will perform the build, test, and deployment steps in Docker containers. By following this guide, even users with minimal background knowledge will be able to launch the infrastructure, install required software, configure Jenkins, set up a pipeline triggered by GitHub pushes, and handle all necessary networking and persistence configurations.

2. System Architecture

- **Jenkins Master (Controller):** Runs in a Docker container on the first EC2 instance. It hosts the Jenkins web UI, receives GitHub webhook notifications, and coordinates pipeline execution. The master triggers build/test jobs on the agent node and aggregates results. It listens on port **8080** for the web UI and webhook endpoint, and on port **50000** for inbound agent connections (JNLP).
- **Jenkins Agent (Worker):** Runs in a Docker container on the second EC2 instance. The agent connects to the master and executes the pipeline tasks (building Docker images, running the application's test suite, etc.) as instructed by the master. In this setup, the agent will use Docker to build and run the Domain Monitoring System application for testing. The application under test may be exposed on port **8081** (for example, for web UI tests), and the agent uses port **50000** to communicate with the master.

Both instances are in the same AWS Virtual Private Cloud (VPC) and will communicate over the private network for Jenkins traffic. The diagram below illustrates the high-level architecture:

- **GitHub Repo:** Hosts the source code and a `Jenkinsfile` defining the pipeline. A webhook on the repo will notify Jenkins of new commits.
- **AWS EC2 Jenkins Master:** Docker container running Jenkins (exposed on port 8080). Receives webhook, triggers pipeline.
- **AWS EC2 Jenkins Agent:** Docker container running Jenkins agent (connecting on port 50000). Executes pipeline stages (build, test, deploy) inside Docker.

3. Prerequisites

Before starting, ensure you have the following:

- **AWS Account:** An AWS account with rights to create EC2 instances and security groups.
- **Key Pair:** An AWS EC2 Key Pair for SSH access to the instances (you can create one in the AWS console if not already available) [1].

- **EC2 Instances:** Two EC2 virtual machines (VMs) running **Ubuntu 24.04 LTS** (64-bit). Use the **t3.micro** instance type (which is sufficient for Jenkins in a test setup and is free-tier eligible). For clarity, name one instance "jenkins-master" and the other "jenkins-agent". (Ensure at least 10 GB of disk, and 1 GB RAM is a bare minimum for Jenkins, though more is recommended.)
- **Security Group Settings:** Configure AWS security group rules to allow required network traffic:
- **SSH (Port 22)** for remote access to both instances.
- **Jenkins Web (Port 8080)** open to your network (or public internet, if webhooks are coming from GitHub) for the Jenkins Master.
- **Jenkins Agent (Port 50000)** open between the Master and Agent instances for JNLP agent communication. (You can restrict this port to the private IP of the other instance for security.)
- **Application Port (8081)** on the agent instance, if the application under test needs to be reachable (e.g., for a web UI or external testing). This can be open to the master or your network as needed.
- Ensure **outbound internet access** is allowed (for package installation, plugin downloads, and for Jenkins to reach GitHub/DockerHub).
- **Software and Accounts:**
- **GitHub Repository:** Access to the GitHub repo containing the Domain Monitoring System code and Jenkinsfile (in this case: `https://github.com/MatanItzhaki12/domain_monitoring_devops.git`).
- **GitHub Account:** A GitHub account with permissions to set webhooks on the repository.
- **DockerHub Account:** (Optional) If the pipeline will push Docker images to DockerHub, have an account ready and credentials that can be used in Jenkins.
- **Jenkins CLI Tools:** *Not required.* We will do everything via the web UI and CLI on the servers.
- **Local Machine Tools:** SSH client to connect to the EC2 instances, and a text editor to edit files if needed.

Ensure you have these prerequisites sorted out before proceeding. Next, we will launch and configure the Jenkins Master server.

4. Jenkins Master Node Setup

4.1 Launch EC2 Instance for Jenkins Master

1. **Launch Instance:** Log in to the AWS Management Console, go to the EC2 dashboard, and click **"Launch Instance"**. In the launch wizard:
2. **Name and OS:** Enter a name (e.g., "jenkins-master"). Under *Application and OS Images (AMI)*, select an Ubuntu Server **24.04 LTS** AMI (64-bit).
3. **Instance Type:** Choose **t3.micro** (or equivalent) as the instance type.
4. **Key Pair:** Select an existing Key Pair or create a new one for SSH access. *Ensure you have the* `.pem` *file downloaded if new.*
5. **Network Settings:** Choose the VPC and subnet. Attach or create a **Security Group**:
    - Allow **SSH (TCP 22)** from your IP (or a restricted range) so you can connect.
    - Allow **HTTP (TCP 8080)** from the internet or your IP (this will be used for Jenkins Web UI and receiving GitHub webhooks).
    - Allow **TCP 50000** from the Jenkins Agent instance (you can specify the agent's security group or IP as the source) for JNLP agent communications.
    - (Optional) You can open **TCP 8081** here as well if planning to use it for the application test access (not needed on master, typically on agent).
6. **Storage:** Allocate storage (the default 8-10 GB is usually fine for Jenkins, but consider 20+ GB if you'll have many builds).
7. **Launch:** Review and launch the instance. Wait until its state is "running".

Note: For simplicity in testing, you might temporarily allow 0.0.0.0/0 on ports 8080/50000 (open to all). **However, this is not secure for production** – restrict these to specific IPs or ranges. For the webhook to reach Jenkins, port 8080 needs to be accessible from the public internet (GitHub's servers).

1. **Obtain Connection Info:** After launch, note the **Public IPv4 address** or **Public DNS** of the instance (for web access and SSH). Also note the **Private IP** (e.g., 172.31.x.x) – this will be used by the Jenkins agent to communicate with the master internally.

4.2 Connect to the Master via SSH and Install Docker

1. **SSH into Instance:** Open a terminal on your local machine and connect to the new EC2 instance using SSH. For example:

```
ssh -i /path/to/your-key.pem ubuntu@<Master_Public_IP>
```

Replace `<Master_Public_IP>` with the actual public IP or DNS. The default username for Ubuntu AMIs is "**ubuntu**". Ensure your `.pem` key file has restrictive permissions (`chmod 400`) so SSH will accept it. You should now have a shell on the EC2 instance.

1. **Update System Packages:** Once connected, update apt packages to get the latest listings:

```
sudo apt update
```

1. **Install Docker:** Install Docker Engine and other helpful tools:

```
sudo apt install -y docker.io git
```

This installs Docker (from Ubuntu's repositories) and Git. We include Git in case you need it for troubleshooting or if Jenkins needs it (Jenkins will also have its own Git plugin). The Ubuntu `docker.io` package provides the Docker daemon and CLI.

1. **Install Docker Compose (Optional):** If you need Docker Compose, you can install it as well (not strictly required for Jenkins itself). For Ubuntu 24.04, you might install Compose V2 plugin:

```
sudo apt install -y docker-compose
```

(This step is optional, as our Jenkins pipeline will not directly use `docker-compose` unless your Jenkinsfile does so.)

1. **Post-Install Docker Steps:** Start the Docker service and enable it to run on startup:

```
sudo systemctl enable docker
sudo systemctl start docker
```

The `enable` ensures Docker daemon starts on boot, and `start` launches it now (Ubuntu typically starts it upon install, but this ensures it's running) [2]. Verify Docker is active with `sudo systemctl status docker` (it should show "active (running)").

1. **Docker Without Sudo (Optional):** By default, you need `sudo` to run Docker commands. To allow the `ubuntu` user to run Docker without sudo (for convenience):

```
sudo usermod -aG docker ubuntu
```

This adds the **ubuntu** user to the **docker** group. You have to log out and log back in (or use `newgrp docker`) to apply this change. This step is optional for Jenkins usage because we will be running Docker containers via sudo or system services. However, it's useful for manual Docker commands and maintenance.

Gotcha: If you skip this, remember to use `sudo` for all Docker commands on the command line. Jenkins in our setup will be running inside a container and won't be affected by the host user's group.

4.3 Run Jenkins Master in Docker

Now we will run Jenkins on the master instance using Docker. Instead of installing Jenkins via apt or war, the official **jenkins/jenkins** Docker image will be used, which includes Jenkins and runs it on port 8080.

1. **Create Jenkins Volume:** It's good practice to use a Docker volume for Jenkins home directory to persist data. We can create a named volume (Docker will create it on the fly in the run command as well). Let's do it explicitly:

```
sudo docker volume create jenkins_home
```

This volume will store Jenkins configuration and job data so that it isn't lost if the container restarts or is recreated.

1. **Run Jenkins Container:** Launch the Jenkins master container using the official LTS image:

   ```
   sudo docker run -d --name jenkins-master \
     -p 8080:8080 -p 50000:50000 \
     -v jenkins_home:/var/jenkins_home \
     --restart=unless-stopped \
     jenkins/jenkins:lts
   ```

   Breakdown of this command: - `-d` runs the container in detached mode (in the background). - `--name jenkins-master` names the container for easy reference. - `-p 8080:8080` maps container port 8080 to host 8080 (Jenkins web UI). - `-p 50000:50000` maps container port 50000 to host 50000 (JNLP agent port). - `-v jenkins_home:/var/jenkins_home` attaches the volume we created to Jenkins home directory in the container, so Jenkins data persists. - `--restart=unless-stopped` configures Docker to auto-restart this container if it crashes or if the Docker service restarts (e.g., on reboot). - `jenkins/jenkins:lts` specifies the Jenkins Docker image (Long-Term Support version).

This command will download the Jenkins image (if not already present) and start the Jenkins service inside the container. It may take a minute on first run to download the image.

2. **Verify Container Status:** After a few seconds, check that the container is running:

```
sudo docker ps -l
```

You should see `jenkins/jenkins:lts` in the list with STATUS "Up ...". If it's not running (`docker ps` shows it exited), use `sudo docker logs jenkins-master` to see errors. Common issues might be port conflicts (ensure nothing else is on 8080/50000) or insufficient memory (t3.micro should handle Jenkins, but it's on the lower end; monitor usage if needed).

3. **Allow Jenkins to Initialize:** Jenkins will run an initial setup inside the container. It may take a minute or two on first startup (it installs some default plugins). You can watch the log output:

```
sudo docker logs -f jenkins-master
```

Look for a line that says something like "Jenkins initial setup is required" or "Please use the following password..." – this contains the initial admin password. You will need this for the next step.

Once you see **"Jenkins is fully up and running"** in the logs, the Jenkins Master is ready.

4.4 Unlock and Configure Jenkins Master

With Jenkins running, we need to perform the initial web-based setup: unlocking Jenkins, installing plugins, and creating an admin user.

1. **Access Jenkins Web UI:** In a web browser, navigate to `http://<Master_Public_IP>:8080`. (If you're using an AWS public DNS name, that works too, e.g., `http://ec2-xx-xx-xx-xx.compute-1.amazonaws.com:8080`). You should see the "Unlock Jenkins" screen. This means Jenkins container is running and accessible.

2. **Retrieve Initial Admin Password:** Jenkins, on first run, generates a random initial administrator password. We need to obtain it from the container. In your SSH session on the master instance, run:

```
sudo docker exec jenkins-master cat /var/jenkins_home/secrets/
initialAdminPassword
```

This will output a 32-character alphanumeric password to the console. Copy this value.

3. **Unlock Jenkins:** In the browser, paste the copied password into the "Administrator password" field and click **"Continue"**. If the password is correct, you'll proceed to the next setup screen.

4. **Install Plugins:** Jenkins will ask you to **"Customize Jenkins"**. Choose **"Install suggested plugins"**. Jenkins will then automatically download and install a set of recommended plugins.

These typically include: Git, GitHub, Pipeline, Docker Pipeline, Credentials, etc., which are useful for our CI/CD pipeline. This process may take a few minutes. You'll see a progress screen as plugins install. (Ensure the instance has internet access for this; if behind a proxy, additional config is needed, but on AWS with default settings it should be fine.)

Note: If you prefer a custom set of plugins, you can choose the option to select plugins and include **Git**, **Pipeline**, **Docker Pipeline**, **GitHub**, **SSH Agent**, **JUnit**, etc. But "suggested" covers most of these. You can always install more plugins later via **Manage Jenkins > Manage Plugins**.

5. **Create Admin User:** After plugins install, Jenkins will prompt to create your first admin user. Fill in a username, password, full name, and email for the admin. (For example, user: `admin`, pass: a strong password). This will create a new admin user and disable the default built-in admin (which was using that initial password). If you prefer, you can skip creating a user and continue using the `admin` user with the initial password, but that's not recommended for real setups. Click **"Save and Finish"**.

6. **Confirm Jenkins is Ready:** You should see a "Jenkins is ready" message and a **"Start using Jenkins"** button. Click it to go to the Jenkins Dashboard. Now you have a running Jenkins master, accessible at `http://<Master_IP>:8080`, with an admin account.

7. **Configure Global Settings (Optional):** This is a good time to check a couple of Jenkins settings:

    - Navigate to **Manage Jenkins > Configure System**. Set the **Jenkins URL** to `http://<Master_Public_IP>:8080` (or a domain if you have one) and save. This ensures Jenkins uses the correct URL in things like webhook configuration and links.
    - Under **Manage Jenkins > Manage Nodes**, you will see the master node (often named "Built-In Node" or "Master"). By default, the master is set to handle build executors. However, since we will use a separate agent, you can reduce the master's **# of executors** to 0 to avoid running builds on the master (recommended for security and to dedicate resources to the agent). To do this, click the master node, choose **Configure**, set "Number of executors" to 0, and save. (This step is optional but aligns with best practice to use agents for builds.)

At this point, the Jenkins Master is fully set up. Next, we will set up the Jenkins Agent node.

5. Jenkins Agent Node Setup

The Jenkins Agent node will run builds and tests. We will launch the second EC2, install Docker (and Java), and connect it to the master.

5.1 Launch EC2 Instance for Jenkins Agent

1. **Launch Second Instance:** In AWS EC2 console, launch another instance for the Jenkins Agent:
2. Use Ubuntu 24.04 LTS (same AMI type) and **t3.micro** instance type.
3. Name it "jenkins-agent" for clarity.
4. In the Security Group settings, you can use the same security group as the master if it was configured to allow needed ports, or create a new one. The key requirements:
    - Allow **SSH (22)** from your IP (to manage the agent server).
    - Ensure this agent can communicate with the master on port **50000**. If using the same security group, you could have an inbound rule on the master allowing port 50000 from

    the security group itself (intra-security-group traffic). Otherwise, allow port 50000 inbound on master from the agent’s IP or subnet.
    - Allow **8081** inbound on the agent if the application under test will be accessed externally (for example, if you want to view the test application’s UI in a browser or allow Selenium on another machine to reach it). This can be optional – if tests run entirely within the agent, you might not need 8081 open to the world.
    - (Opening 8080 on the agent is not necessary for Jenkins, since the agent doesn’t run a web UI; any 8080 use on agent would be for the app itself if at all.)

5. Use the **same Key Pair** for SSH, or a new one. Launch the instance and wait for it to start.

6. **SSH into Agent:** Just like with the master, SSH into the agent instance:

```
ssh -i /path/to/your-key.pem ubuntu@<Agent_Public_IP>
```

Use the agent instance’s public IP/DNS. You should get a shell on the agent VM.

5.2 Install Docker and Java on Agent

1. **Update and Install Packages:** On the agent EC2, update apt and install Docker, Java, and Git:

```
sudo apt update
sudo apt install -y docker.io openjdk-11-jre-headless git
```

- **Docker:** This installs Docker engine on the agent (so the agent can run Docker containers for builds/tests).
- **Java:** Jenkins agents require Java to run the agent program. We're installing OpenJDK 11 JRE headless which is sufficient for the Jenkins agent process. (Jenkins inbound agent is basically a Java program that connects to the master.)

- **Git:** Installing Git (for completeness, if the agent needs to run any git commands or if you want to use it manually; the Jenkins pipeline will usually handle SCM checkout via the master’s plugins).

- **Add ubuntu to docker group (Optional):**

```
sudo usermod -aG docker ubuntu
```

Similar to master, this is for convenience. Since we'll likely run the agent container with sudo or as a system service, it's not strictly necessary, but it can help if you run Docker commands manually on the agent.

1. **Start Docker and Enable on Boot:**

```
sudo systemctl enable docker
sudo systemctl start docker
```

Ensure Docker daemon is running on the agent as well. The agent will be running a Docker container for the Jenkins agent process, and also using Docker to run builds, so the Docker service must be active.

1. **Verify Java:** Check that Java is installed by running `java -version`. You should see an OpenJDK version (11.x). This confirms the agent can run Java programs (the Jenkins agent will run as a Java process inside a container or directly, but having Java on host is a good backup and is needed if we run the agent as a jar outside of Docker).

At this stage, the agent machine has Docker and Java set up – ready to run the Jenkins agent.

5.3 Register Agent Node in Jenkins Master

Before we run the agent, we must inform the Jenkins master about this new agent and obtain the secret for connection. We'll do this via Jenkins's web interface:

1. **Create Node in Jenkins UI:** On the Jenkins master web UI, go to **Manage Jenkins > Manage Nodes and Clouds** (or it might just say "Manage Nodes"). Click **"New Node"**. In the dialog:
2. Enter a node name, e.g., **"agent1"** (or any identifier you like).
3. Select **Permanent Agent** (not a cloud or ephemeral).
4. Click **OK** to proceed to node configuration.
5. **Configure Agent Node:**
6. **Remote root directory:** set this to a path where the agent's workspace will live *inside the agent container*. A common value is `/home/jenkins` or `/var/jenkins` (the container's default working directory for the Jenkins user). For simplicity, use `/home/jenkins/agent` or `/home/jenkins` – we will ensure the container uses the same.
7. **Labels:** (Optional) add a label like `docker` or `agent` or `linux`. If your Jenkinsfile uses a node label (e.g., `agent { label 'docker' }`), put that here. Otherwise, you can leave it blank or put a generic label.
8. **# of executors:** leave it as 1 (meaning this agent can run one build at a time, which is fine for a t3.micro).
9. **Launch method:** This is crucial. Choose **"Launch agent by connecting it to the controller"** (formerly called "JNLP" or inbound launch). This means we will start the agent manually, and it will initiate the connection to Jenkins (as opposed to Jenkins trying to SSH into the agent, which we are not using here).
10. You can leave other settings (work directory, environment variables) default for now.

Click **Save**. Jenkins will add the node. Initially, it will show as **offline** (which is expected, since we haven't started the agent process yet).

1. **Obtain Agent Secret:** After saving, Jenkins will show an agent page. Since we chose "launch by connecting it", Jenkins provides a **secret key** for the agent and instructions. On the node's page, look for something like:
2. A command line snippet and the **secret**. It might say: *"Use the following command to connect this agent:"* and show a command with `-secret <long_hex_value> -jnlpUrl http://<master>:8080/computer/agent1/jenkins-agent.jnlp` etc. The key part is the `<secret>` (a long alphanumeric string). Copy this secret value.

3. If it doesn't show a full command, you will definitely see the **secret** listed on this page (or click "Secret" or "Agent.jar" link). Copy the secret to a safe place temporarily. Also note the agent name you used (e.g., agent1).

We will use this secret to authenticate the agent when it connects to the master. (Jenkins uses this secret to ensure an agent is allowed to connect as that node.)

5.4 Start Jenkins Agent as a Docker Container

Now we have the agent node defined and a secret. We'll run the Jenkins agent on the agent EC2 instance inside a Docker container.

1. **On the agent EC2, run the Jenkins inbound agent container:**

   We will use the official Jenkins inbound agent image (`jenkins/inbound-agent`). This image contains the Jenkins agent software (Remoting library) and will connect to the master. Run the following on the **agent VM**:

   ```
   sudo docker run -d --name jenkins-agent --init -u root \
     -v /var/run/docker.sock:/var/run/docker.sock \
     --restart=unless-stopped \
     jenkins/inbound-agent:latest \
     -url http://<Master_Private_IP>:8080 \
     <agent-secret> <agent1>
   ```

   Replace: - `<Master_Private_IP>` with the private IP address of the master instance (e.g., `http://172.31.x.x:8080`). We use the private IP because the agent and master are within the same AWS network, and the security group allows port 50000 internally. *(If for some reason the agent cannot reach the master's private IP, you could use the public IP, but then ensure port 50000 is open to the agent's public IP.)* - `<agent-secret>` with the secret key you copied from Jenkins. - `<agent1>` with the agent node name you configured (it must exactly match the name in Jenkins). For example, if your node is "agent1", put `agent1` here.

   Let's break down this command: - `jenkins/inbound-agent:latest` is the Docker image for Jenkins agent. (It's effectively a Java program that will connect to the master). - We pass `-url http://<master>:8080` to tell the agent where the Jenkins controller is. - Then we provide the `<secret>` and `<agent name>` as arguments as required by the image entrypoint. - We include `--init` which runs an init process inside the container. This is recommended for proper signal handling and to avoid zombie processes when the agent runs builds. - We run as `-u root`. This ensures the container's process runs as root user. We do this because we will mount the Docker socket and need permissions to run Docker inside, and also to easily install Docker CLI inside the container. (The image's default user might be `jenkins` which may not have permission on `/var/run/docker.sock`.) - We mount the host's Docker socket into the container: `-v /var/run/docker.sock:/var/run/docker.sock`. This is **critical**: it allows the agent (inside container) to use the host machine's Docker daemon. That way, any Docker commands executed by Jenkins on the agent will actually run on the agent host (enabling the agent to build images or run containers as needed). - We set `--restart=unless-stopped` so this agent container will restart automatically on reboot or if it crashes.

After running this, Docker will start the agent container. You can verify with `sudo docker ps` that `jenkins/inbound-agent:latest` is running.

2. **Check Jenkins Agent Connection:** Go to the Jenkins web UI, **Manage Jenkins > Manage Nodes**. The agent node you created ("agent1") should now show a green indicator or "Connected" and "Online". In the agent's status page, you might see some system info appear, and it should say **"agent1 is online"**.

   If it still shows offline after a minute: - Check the logs of the agent container: `sudo docker logs -f jenkins-agent` on the agent VM. Look for errors such as inability to resolve the master hostname or authentication issues. Common fixes: - If it says "unknown host" or "connection refused", ensure the URL is correct and reachable. For private IP, the agent's VPC DNS must resolve the master's IP or you can just use the IP numbers. - If it says "Unauthorized" or bad secret, double-check the secret and agent name. - If networking is an issue, you can try using the master's public DNS in the URL instead (with proper security group open). - Verify the master is listening on port 50000. (Jenkins Docker image by default exposes it. In Jenkins UI, under **Configure Global Security**, the "TCP port for inbound agents" is by default 50000 – it should be enabled by our Docker run which mapped that port.)

   Once connected, the master's UI should list the agent with some details (like number of executors, etc.). Great – we have an operational Jenkins agent!

3. **Install Docker CLI inside Agent Container:** *This is a crucial step for CI with Docker.* The Jenkins agent container we launched includes Java (to run the agent) but it **does not include the Docker CLI** by default. Our pipeline will likely use Docker commands (like `docker build`, `docker run`, etc.) on the agent. Since we mounted the Docker socket, any Docker CLI call inside the agent container will control Docker on the host. But if the `docker` command is not present inside the container, those steps will fail.

   To install Docker CLI in the running agent container, execute:

   ```
   sudo docker exec -u root jenkins-agent apt-get update
   sudo docker exec -u root jenkins-agent apt-get install -y docker.io
   ```

   What this does: - `docker exec` runs a command inside an existing container. We run as root user (we started the container as root, but we include `-u root` to be explicit). - We update package lists and install the `docker.io` package inside the container. The `jenkins/inbound-agent` image is based on a Linux distribution that supports apt (currently it's a Debian-based image), so this installs the Docker CLI within the container.

   After this, you can test by running: `sudo docker exec jenkins-agent docker version`. It should execute the Docker CLI inside the container and output version info, indicating the container can invoke Docker on the host.

   Alternative: Instead of this manual install, one could use a custom agent Docker image that already has Docker installed (or use the official `jenkins/inbound-agent` as a base and `Dockerfile` to add Docker CLI). But the above approach is a quick solution.

4. **Persistence & Auto-start Considerations:** We already set `--restart=unless-stopped` on the agent container, so it will try to come back after reboots. However, note that if Jenkins master is not up when the agent starts, it will continuously retry to connect (which is fine). The agent container will run as a background service on the agent VM. There's no further action needed, but for completeness:

    - The agent's **work directories** (Jenkins workspace, etc.) are inside the container (by default under `/home/jenkins` or `/agent` depending on image). If you want to persist workspace or cache, you could mount a volume for `/home/jenkins` too. This is optional; builds typically can be ephemeral.
    - We installed Docker CLI inside the container; that layer will disappear if the container is removed. If you ever recreate the agent container, remember to reinstall Docker CLI or build it into the image.
    - We have effectively "wrapped" the JNLP agent in a system service via Docker. If you prefer a more traditional approach, you could run the agent as a systemd service running `java -jar agent.jar ...` on the host. The provided design mentioned using a `.service` file for the agent, but since we use Docker's restart mechanism, we do not necessarily need an external service file for the agent container.

At this point, we have Jenkins master and agent up and connected. Now we will configure a Jenkins pipeline job and the GitHub webhook to trigger it.

6. Jenkins Pipeline Job Configuration

With the infrastructure in place, let's set up the actual CI/CD pipeline job in Jenkins and integrate it with GitHub.

6.1 Create a Pipeline Job in Jenkins

1. **Create New Pipeline Job:** On the Jenkins dashboard (master UI), click **"New Item"**. Enter an item name, for example **"DomainMonitoring_CI_Pipeline"**. Select **"Pipeline"** as the project type (since our Jenkinsfile defines a pipeline) and click **OK**.

2. **General Settings (optional):** In the new job configuration page, you can add a description like "Pipeline for Domain Monitoring System CI/CD". If your Jenkins has the GitHub plugin installed (it should, from suggested plugins), you'll see a "GitHub project" checkbox. You can check it and paste the GitHub repository URL (`https://github.com/MatanItzhaki12/domain_monitoring_devops.git`) for reference. This isn't strictly necessary but provides a direct link to the repo from Jenkins.

3. **Source Code Management:** Under **Pipeline** settings (scroll down), find the **Pipeline script from SCM** option:

4. Choose **SCM: Git**.
5. **Repository URL:** put the GitHub repo URL (`https://github.com/MatanItzhaki12/domain_monitoring_devops.git`). Jenkins will attempt to fetch this. If the repo is public, no credentials are needed. If private, you'd add credentials in Jenkins and select them here.
6. (If Jenkins complains about Git not being available, ensure the Git plugin is installed. The suggested plugins usually include it. Alternatively, Jenkins can use JGit internally. Our earlier

installation of Git on the OS was more for convenience, as the Jenkins container has its own environment.)
7. **Branch Specifier:** you can leave as `*/master` or `*/main` depending on the repo's main branch. Let's assume the main branch is `main` (adjust if needed). You can put `*/main` or the specific branch name.
8. **Script Path:** this is the path to the Jenkinsfile in the repository. If the Jenkinsfile is at the root of the repo and named "Jenkinsfile", leave it as `Jenkinsfile` (default). If it's in a subfolder, specify the path (e.g., `ci/Jenkinsfile`).

This configuration tells Jenkins to pull the Jenkinsfile from your GitHub repo, so our pipeline logic is source-controlled.

1. **Build Triggers:** To have Jenkins respond to GitHub pushes, scroll to **Build Triggers** section:
2. Check the box **"GitHub hook trigger for GITScm polling"**. This enables Jenkins to be triggered by the GitHub webhook. (Under the hood, the GitHub plugin will listen for the webhook call and then schedule a build without needing to poll.)
3. You do not need to enable polling (the webhook replaces it), and you don't need other triggers for now.

Note: Ensure that in **Manage Jenkins > Manage Plugins**, the "GitHub Plugin" (for webhooks) is installed, and in **Manage Jenkins > Configure System**, a GitHub server may be added. The default configuration usually works for public repos via webhooks. If the webhook doesn't trigger builds, you might need to configure a GitHub API token in Jenkins and add GitHub under Configure System – but often just the plugin and the checkbox is enough for basic push events.

1. **Save the Job:** Click **Save** at the bottom of the job configuration. You should now see the pipeline job created on the dashboard.

At this point, Jenkins knows about the repo and is ready to run the pipeline defined in the Jenkinsfile. Next, we'll set up the GitHub webhook so pushes to the repo notify Jenkins.

6.2 Configure GitHub Webhook

1. **Set up Webhook in GitHub Repo:** Go to the GitHub repository page (https://github.com/MatanItzhaki12/domain_monitoring_devops.git) in your browser. You'll need admin access to the repo to add a webhook.
2. Click on **"Settings"** (the Settings tab of the repository).
3. In the left sidebar, choose **"Webhooks"**, then click **"Add webhook"**.
4. **Payload URL:** This should point to your Jenkins master's webhook endpoint. Format: `http://<Master_Public_IP>:8080/github-webhook/`. (Make sure to use the correct protocol (`http` or `https` if you set up SSL). For our setup, it's `http`. If you had a custom domain or DNS, you could use that. Ensure there's a trailing slash and **"github-webhook"** exactly, as this is the endpoint Jenkins GitHub plugin listens on.)
5. **Content type:** Select **application/json** (GitHub will send JSON payloads).
6. **Secret:** (Optional) You can leave this blank for simplicity. If you fill it, you'd have to configure the same secret in Jenkins' GitHub plugin settings. For now, blank means no signature verification.
7. **Which events:** Choose **"Just push events"**. This will trigger on pushes to any branch (you can further restrict in webhook settings if needed, but generally push is fine).
8. Click **"Add webhook"** to save. GitHub will immediately try to send a ping event to the URL.
9. After adding, you should see the new webhook in the list. A green check mark or "Last delivery was successful" indicates Jenkins responded (or at least returned HTTP 200). If you see a red X,

click the webhook to see the delivery details. Jenkins might not respond to the test ping with content, but as long as it's a 200 OK, GitHub will mark it green.

10. **Verify Jenkins Received Webhook:** In Jenkins, go to **Manage Jenkins > System Log** (or look at the logs via `docker logs` on the master). You might see an entry that a payload was received. Another way: in the GitHub webhook configuration page, there is a **Recent Deliveries** section. Select the latest delivery (the ping or a test push) and check the response. It should show a response code **200** if Jenkins endpoint received it.

Jenkins is now set to trigger the pipeline whenever a push happens on the repository.

Tip: If you want Jenkins to automatically manage webhooks (instead of creating manually), you could configure the GitHub integration with an API token and set **Manage Hooks** to let Jenkins create them. But manual setup as we did is straightforward for a single pipeline.

6.3 Initial Pipeline Run and Testing

1. **Run a Test Build:** Back in Jenkins, open the pipeline job we created. Since this is the first run, it's often good to run it manually to ensure everything is in place (and in some cases Jenkins registers the webhook after first run). Click **"Build Now"** for the pipeline job. This should queue a build and start it within a few seconds.
2. The build will likely go to the agent (if your Jenkinsfile has `agent any` or a specific label, Jenkins will use the appropriate node). Because we set master executors to 0, it should choose the agent.
3. Click the build in the build history to see the console output. Jenkins will: check out the repository (you should see "Checking out Git ..."), then proceed through the stages defined in the Jenkinsfile (e.g., build Docker image, run tests, etc.). This confirms that Jenkins can fetch the code and that the agent is functioning.
4. If the build fails at checkout stage with an authentication issue, the repo might be private – you'll need to add credentials in Jenkins for Git. If it fails later, debug according to the error (maybe Docker commands, etc., see Troubleshooting below).

5. If the build passes, great. If it fails due to tests (which could happen if something in the app or tests is not correct), that's application-specific – Jenkins is set up correctly if it at least ran the stages.

6. **Trigger via Git Push:** Now test the webhook trigger. Make a trivial commit to the GitHub repo (e.g., edit README or a minor change in code) and push it to the repository's main branch. Within a few seconds of pushing, Jenkins should detect the webhook and queue a new build automatically (you'll see it on the Jenkins dashboard with an incremented build number). This confirms the end-to-end automation: GitHub -> Jenkins webhook -> Jenkins agent runs the pipeline.

7. **Monitor and Review:** Watch the pipeline run on Jenkins. The console output will show each step. For the Domain Monitoring System project, you should see steps corresponding to those outlined in the design:

    - Docker image build (temporary image tagging perhaps with commit ID).
    - Spinning up a container of the app.
    - Running the pytest and selenium test suite.

    - If tests pass, tagging and pushing a new Docker image to DockerHub (make sure to configure DockerHub credentials if this part is in the Jenkinsfile).
    - Cleaning up containers/images.
    - The Jenkins console will mark the build **SUCCESS** or **FAILURE** accordingly.

  All network and file configurations we set up (ports open, Docker available, volumes) should facilitate these steps. For example, the agent's Docker can pull base images and push to DockerHub (it has internet access), the test container can reach any external services if needed, etc.

7. Network and Firewall Considerations

- **Security Group Recap:** The master node's security group should allow inbound 8080 (web UI) from wherever you need (your IP for UI, and GitHub's servers for webhook). GitHub's webhook IP ranges are broad, so often 0.0.0.0/0 on 8080 is used for simplicity. In a tighter security setup, you might front Jenkins with a proxy or limit by source IP (GitHub publishes a list of hook IPs). The agent's security group should allow inbound from the master on 50000 (or both could be in same SG with a rule to allow SG-to-SG traffic on 50000). We also opened 8081 on the agent for potential external access to the test application (if not needed, you can omit that). Always lock down SSH (port 22) to your IP or use AWS Session Manager for access.
- **Internal Traffic:** The master and agent communicate over port 50000. We used the master's **private IP** in the agent configuration, avoiding exposing 50000 publicly. As long as the security groups and VPC settings allow it, this is more secure and faster. The agent container will continuously try to reconnect if it loses connection. This is normal, just ensure it can reach the master.
- **Firewall (Ubuntu UFW):** Ubuntu by default has UFW off on AWS instances. If you enable it, remember to `ufw allow 22,8080,50000,8081` as needed. Otherwise, the AWS security group is your firewall.
- **DNS and IPs:** Using IP addresses in webhook and agent config is fine if they are static or long-lived. Note AWS EC2 public IPs can change on stop/start (unless using Elastic IP). You might consider using an Elastic IP for the Jenkins master so the webhook URL doesn't break. Or update the webhook if IP changes. Alternatively, use a domain name (and maybe a dynamic DNS) for convenience.
- **HTTPS for Jenkins:** This SOP didn't cover securing Jenkins with SSL. For production, consider putting Jenkins behind an HTTPS reverse proxy or using Amazon's ALB with SSL. At minimum, set up credentials and matrix security if multiple users.
- **Jenkins Master Persistence:** All Jenkins data (jobs, config, plugins) is on the `jenkins_home` Docker volume. It's good to routinely back this up (you can snapshot the volume or copy data out). If the container or instance is lost, you can re-launch Jenkins with that volume to restore state.

8. Additional Configuration and Troubleshooting

- **DockerHub Credentials:** If the pipeline pushes Docker images to DockerHub (as indicated by the project, pushing `matanitzhaki/domain-monitoring` images), you need to provide credentials. In Jenkins, go to **Manage Jenkins > Manage Credentials** (or directly in the pipeline job config under credentials). Add your DockerHub username/password (or token) as a credential. In the Jenkinsfile, ensure `docker login` is performed (or use withCredentials). The example pipeline likely expects certain credentials IDs or environment variables (like `DOCKERHUB_USER` and `DOCKERHUB_PASS`). Set those up in Jenkins (either in credentials or as global env vars in the node config).

- **Email/Notifications:** If you want Jenkins to send emails on failures, configure an SMTP server in **Manage Jenkins > Configure System > E-mail Notification** and use the Email Extension Plugin in the pipeline.
- **Agent Offline Issues:** If the agent goes offline frequently:
- Ensure the agent container is running (`docker ps` on agent). Docker's restart policy should keep it running.
- Check for network connectivity issues between agent and master (security group changes, etc.).
- Ensure the master's TCP inbound port is set to 50000 (default) and not changed after restart.
- You can also enable the **Work Dir** for the agent: for durability, run agent with `-workDir=/home/jenkins/agent` and mount a volume. This can help if the connection drops; the agent can store state. (This was mentioned in the Jenkins documentation as an option).
- **Resource Considerations:** t3.micro is small (1 vCPU, ~1GB RAM). Running Jenkins and Docker builds/tests on these might be slow. The master on t3.micro is usually fine for light use. The agent running tests (especially with Selenium, maybe launching a browser headless, etc.) could be constrained. Monitor CPU/RAM usage. If needed, consider t3.small or larger for the agent to speed up tests. You can also use swap on AWS or adjust test parallelism.
- **Selenium Tests:** If the pipeline includes Selenium UI tests, ensure the environment can support it. Often, projects use a headless browser (Chrome/Firefox headless). Make sure the Docker image used for tests has the necessary drivers and maybe a virtual display if not headless. If tests fail due to browser issues, you might need to tweak the Dockerfile or Jenkinsfile (this is outside the SOP's main scope, but worth noting).
- **Pipeline Debugging:** You can always test parts of the pipeline by logging into the agent and manually running the Docker build or test commands to see what's wrong. The Jenkins workspace on the agent (inside the container) can be found under the container's file system (for example, `/home/jenkins/workspace/<job_name>`). If a build fails, Jenkins usually prints the error and stacktrace. Use that to pinpoint issues (e.g., test assertion failed, or Docker build failed).
- **Jenkins Logs and Restart:** The master Jenkins logs are viewable via `docker logs jenkins-master`. If Jenkins becomes unresponsive (e.g., high load or memory), you can restart the container: `docker restart jenkins-master`. It will reapply the `--restart` policy (so it will come back up). Agents will reconnect automatically. Frequent restarts might indicate the instance is underpowered.

9. Summary

You have now set up a robust Jenkins CI/CD environment on AWS: - A Jenkins Master in a Docker container on Ubuntu (with persistent storage for config and an admin user configured). - A Jenkins Agent in a Docker container on a second Ubuntu instance (with Docker installed to handle build workloads). - Both instances properly networked (ports 8080, 50000, etc.) and secured via AWS Security Groups. - A Pipeline job that pulls code from GitHub and runs a test/build/deploy pipeline defined in a Jenkinsfile, with triggers in place for automatic builds on `git push`. - Docker is leveraged for Jenkins itself and for the build steps, ensuring environment consistency and isolation.

This SOP is designed to be foolproof: following the numbered steps will set up the system from scratch. By adhering to it, even users with minimal background can deploy an industry-standard CI/CD pipeline. Always remember to monitor your CI system and keep Jenkins and plugins up to date (especially for security patches). Good luck with your Jenkins pipeline and happy DevOps!

---

1 Jenkins on AWS
https://www.jenkins.io/doc/tutorials/tutorial-for-installing-jenkins-on-AWS/

2 How to Install Docker on Ubuntu 24.04: Step-by-Step Guide | Cherry Servers
https://www.cherryservers.com/blog/install-docker-ubuntu