# Automated CI/CD Pipeline for Python Application on AWS EC2

This document outlines the configuration for an automated Continuous Integration (CI) and Continuous Deployment (CD) pipeline for a Python application to an AWS EC2 instance. The pipeline leverages GitHub for source control, AWS CodeBuild for building and packaging, and AWS CodeDeploy for deploying the application.

**1. Project Overview**

* **Application:** A Python application (my\_photo\_app).
* **Source Control:** GitHub (connected to AWS CodePipeline).
* **Build Tool:** AWS CodeBuild.
* **Deployment Tool:** AWS CodeDeploy.
* **Target Environment:** AWS EC2 Instance running Amazon Linux 2 (or similar) with CodeDeploy Agent installed and ec2-user as the primary application user.

**2. Repository Structure**

The application's repository should be structured as follows:

my\_photo\_app/

├── .venv/ (Local virtual environment - NOT deployed)

├── scripts/

│ ├── install\_dependencies.sh

│ ├── start\_app.sh

│ ├── stop\_app.sh

│ └── setup\_deployment\_dir.sh <-- NEW SCRIPT

├── app.py

├── appspec.yml

├── aws\_utils.py

├── bashscript.txt

├── buildspec.yml

├── config.py

└── requirements.txt

**3. Core Configuration Files**

These files dictate the build and deployment process.

**3.1 appspec.yml**

The appspec.yml file is the CodeDeploy specification file. It defines what files to deploy and what scripts to run at various stages of the deployment lifecycle.

**Location:** Root of your repository.

**Final Content:**

YAML

version: 0.0

os: linux

files:

- source: /

destination: /home/ec2-user/my\_photo\_app

hooks:

BeforeInstall:

- location: scripts/setup\_deployment\_dir.sh # Ensure deployment directory is ready

timeout: 60

runas: root # Run as root to manage system-level directory creation/ownership

- location: scripts/install\_dependencies.sh # Install Python dependencies

timeout: 300

runas: ec2-user # Run as ec2-user to manage user-specific virtual environment

ApplicationStart:

- location: scripts/start\_app.sh # Start the Python application

timeout: 60

runas: ec2-user

ApplicationStop:

- location: scripts/stop\_app.sh # Stop the Python application

timeout: 60

runas: ec2-user

**Explanation and Reasoning:**

* **version: 0.0**: Specifies the AppSpec file version.
* **os: linux**: Indicates the target operating system is Linux.
* **files:**: Defines which files from the deployment artifact should be copied to the EC2 instance.
  + **source: /**: Instructs CodeDeploy to take all files from the root of the deployment artifact (created by CodeBuild).
  + **destination: /home/ec2-user/my\_photo\_app**: Specifies that these files should be copied to /home/ec2-user/my\_photo\_app on the EC2 instance. This is the application's base directory.
* **hooks:**: Defines scripts to be executed at different stages of the CodeDeploy lifecycle.
  + **BeforeInstall:**:
    - **location: scripts/setup\_deployment\_dir.sh**: Specifies the path to the script relative to the deployment package root. This script runs *before* the application files are copied to the destination.
    - **timeout: 60**: Sets a maximum execution time of 60 seconds for the script.
    - **runas: root**: **Reason:** This is crucial. To reliably create and set ownership/permissions for /home/ec2-user/my\_photo\_app, the script needs root privileges. This ensures the directory is correctly prepared for the ec2-user before files are copied into it by CodeDeploy.
    - **location: scripts/install\_dependencies.sh**: Specifies the path to the dependency installation script. This runs *after* setup\_deployment\_dir.sh and *before* files are copied (though the script anticipates files being there from a previous deployment step).
    - **runas: ec2-user**: **Reason:** The install\_dependencies.sh script will create a virtual environment and install Python packages. These operations should be performed as the user who will own and run the application (ec2-user) to prevent permission conflicts.
  + **ApplicationStart:**:
    - **location: scripts/start\_app.sh**: Specifies the path to the script that starts your application.
    - **runas: ec2-user**: **Reason:** The application should be run as the ec2-user for proper user context and access to files within /home/ec2-user/my\_photo\_app.
  + **ApplicationStop:**:
    - **location: scripts/stop\_app.sh**: Specifies the path to the script that gracefully stops your application.
    - **runas: ec2-user**: **Reason:** The stop script also needs to run as ec2-user to manage processes owned by that user.

**3.2 buildspec.yml**

The buildspec.yml file is used by AWS CodeBuild to define the build steps, including dependency installation and artifact packaging.

**Location:** Root of your repository.

**Final Content:**

YAML

version: 0.2

phases:

install:

runtime-versions:

python: 3.11

commands:

# Reason: Python virtual environment and dependencies are now installed directly on the EC2 instance

# during the CodeDeploy 'BeforeInstall' hook to ensure platform compatibility and simplify artifact size.

- echo "Skipping Python dependency installation in CodeBuild. This will happen on EC2."

build:

commands:

- echo "Build phase started..."

# Reason: Grant execute permissions to all shell scripts used by CodeDeploy.

# This ensures they can be executed on the EC2 instance regardless of how they were created/uploaded.

- chmod +x scripts/install\_dependencies.sh

- chmod +x scripts/start\_app.sh

- chmod +x scripts/stop\_app.sh

- chmod +x scripts/setup\_deployment\_dir.sh # New script needs executable permissions too.

artifacts:

files:

# Reason: These files are essential application components that need to be deployed.

- 'app.py'

- 'aws\_utils.py'

- 'config.py'

- 'requirements.txt'

- 'bashscript.txt'

- 'appspec.yml'

# Reason: Include all scripts in the 'scripts' directory, preserving their path.

# CodeDeploy expects them to be under 'scripts/' as per appspec.yml.

- 'scripts/\*\*/\*'

# Reason: The virtual environment is no longer pre-built in CodeBuild and included in the artifact.

# It is now dynamically created on the EC2 instance during deployment.

# Removed: - '.venv/\*\*/\*'

discard-paths: no # Reason: Crucial to maintain the folder structure (e.g., 'scripts/' folder) within the artifact.

**Explanation and Reasoning:**

* **version: 0.2**: Specifies the BuildSpec file version.
* **phases:**: Defines the stages of the build process.
  + **install:**:
    - **runtime-versions: python: 3.11**: Specifies the Python runtime version CodeBuild should use.
    - **commands:**:
      * We explicitly removed python3 -m venv .venv, . .venv/bin/activate, and pip install -r requirements.txt from here.
      * **Reason:** This decision was made because pre-building the virtual environment in CodeBuild and packaging it in the artifact led to persistent issues with .venv being missing or having incorrect permissions upon CodeDeploy's unpackaging on the EC2 instance. Creating the virtual environment directly on the EC2 instance during deployment is more robust and avoids potential platform incompatibilities or large artifact sizes.
  + **build:**:
    - **commands:**:
      * **chmod +x scripts/install\_dependencies.sh (and others)**: **Reason:** Ensures that all shell scripts that CodeDeploy will execute have the necessary executable permissions when they are packaged into the artifact. Without this, CodeDeploy would report "ScriptFailed" due to permission errors.
* **artifacts:**: Defines what files from the CodeBuild environment should be packaged into the final output artifact for CodeDeploy.
  + **files:**:
    - Lists all individual application files (app.py, appspec.yml, etc.) that need to be deployed.
    - **scripts/\*\*/\***: **Reason:** This wildcard pattern ensures that the entire scripts directory and all its contents (including install\_dependencies.sh, start\_app.sh, stop\_app.sh, and setup\_deployment\_dir.sh) are included in the artifact, preserving their relative path. This is vital for appspec.yml to correctly locate the scripts.
    - **Removed .venv/\*\*/\***: **Reason:** As decided, the virtual environment is no longer built and included here.
  + **discard-paths: no**: **Reason:** This is critical. It ensures that the directory structure (e.g., scripts/) defined in the files list is maintained within the artifact ZIP. If set to yes, all files would be flattened to the root, making appspec.yml unable to find scripts at scripts/.

**3.3 scripts/setup\_deployment\_dir.sh**

This is a new script executed as root early in the CodeDeploy lifecycle to prepare the deployment target directory.

**Location:** my\_photo\_app/scripts/

**Final Content:**

Bash

#!/bin/bash

set -e

APP\_DIR="/home/ec2-user/my\_photo\_app"

APP\_USER="ec2-user"

echo "Running setup\_deployment\_dir.sh as $(whoami)"

# Reason: Ensure the /home/ec2-user directory exists and is correctly owned.

# This provides a stable base for the application directory.

echo "Ensuring /home/ec2-user exists..."

mkdir -p /home/ec2-user || { echo "ERROR: Could not create /home/ec2-user. Exiting."; exit 1; }

chown $APP\_USER:$APP\_USER /home/ec2-user || { echo "ERROR: Could not chown /home/ec2-user. Exiting."; exit 1; }

chmod 755 /home/ec2-user || { echo "ERROR: Could not chmod /home/ec2-user. Exiting."; exit 1; }

# Reason: Ensure the application's root directory is created, owned by the application user,

# and has correct permissions for writing and traversal.

echo "Ensuring application directory $APP\_DIR exists and is owned by $APP\_USER..."

mkdir -p "$APP\_DIR" || { echo "ERROR: Could not create $APP\_DIR. Exiting."; exit 1; }

chown -R $APP\_USER:$APP\_USER "$APP\_DIR" || { echo "ERROR: Could not chown $APP\_DIR. Exiting."; exit 1; }

chmod -R 755 "$APP\_DIR" || { echo "ERROR: Could not chmod $APP\_DIR. Exiting."; exit 1; }

echo "Deployment directory setup complete."

**Explanation and Reasoning:**

* **#!/bin/bash & set -e**: Standard shebang and error handling for robust shell scripting.
* **APP\_DIR & APP\_USER**: Variables for clarity and easy modification.
* **runas: root in appspec.yml**: This script is designed to run with root privileges. **Reason:** Only root can guarantee the creation and ownership change of directories under /home (like /home/ec2-user) if they don't exist or have incorrect ownership. This resolves potential permission issues that prevented .venv from being unpacked by CodeDeploy in earlier attempts.
* **mkdir -p /home/ec2-user & chown $APP\_USER:$APP\_USER /home/ec2-user**: Ensures the home directory of ec2-user is properly set up.
* **mkdir -p "$APP\_DIR" & chown -R $APP\_USER:$APP\_USER "$APP\_DIR" & chmod -R 755 "$APP\_DIR"**: Ensures the main application deployment directory exists and is fully owned by ec2-user with read, write, and execute/traverse permissions for the owner, and read/execute for others. This is critical for subsequent scripts (run as ec2-user) to function correctly.

**3.4 scripts/install\_dependencies.sh**

This script is responsible for setting up the Python virtual environment and installing application dependencies directly on the EC2 instance.

**Location:** my\_photo\_app/scripts/

**Final Content:**

Bash

#!/bin/bash

set -e # Exit immediately if a command exits with a non-zero status

APP\_DIR="/home/ec2-user/my\_photo\_app"

echo "Current user: $(whoami)"

echo "Deployment target directory: $APP\_DIR"

# Reason: Change to the application directory where files are deployed.

# All subsequent relative paths depend on this.

echo "Navigating to app directory: $APP\_DIR"

cd "$APP\_DIR" || { echo "ERROR: Failed to change directory to $APP\_DIR. Exiting."; exit 1; }

echo "Current working directory: $(pwd)"

# Reason: Create a new Python virtual environment named .venv.

# This isolates application dependencies from system-wide Python packages.

echo "Creating virtual environment..."

python3 -m venv .venv || { echo "ERROR: Failed to create venv. Exiting."; exit 1; }

echo "Virtual environment created."

# Reason: Activate the virtual environment.

# This modifies the shell's PATH and other variables so that 'python' and 'pip'

# commands refer to the executables within the virtual environment.

# Using '.' instead of 'source' for broader shell compatibility (e.g., sh vs bash).

ACTIVATE\_SCRIPT=".venv/bin/activate"

echo "Activating virtual environment..."

. "$ACTIVATE\_SCRIPT" || { echo "ERROR: Failed to activate venv using '$ACTIVATE\_SCRIPT'. Exiting."; exit 1; }

echo "Virtual environment activated successfully."

# Reason: Install Python packages listed in requirements.txt into the active virtual environment.

# This ensures all application dependencies are available.

echo "Installing Python dependencies (pip install -r requirements.txt)..."

pip install -r requirements.txt || { echo "ERROR: Failed to install dependencies. Exiting."; exit 1; }

echo "Python dependencies installed successfully."

# Reason: Verify that Python and Pip are now pointing to the virtual environment.

# Useful for debugging.

echo "Python executable in venv: $(which python)"

echo "Pip executable in venv: $(which pip)"

echo "Dependencies installation complete."

**Explanation and Reasoning:**

* **runas: ec2-user in appspec.yml**: This script runs as ec2-user. **Reason:** The virtual environment and installed packages should be owned by and accessible to the user who will run the application.
* **cd "$APP\_DIR"**: Ensures the script operates within the deployed application's root directory.
* **python3 -m venv .venv**: **Reason:** Creates the virtual environment. This is now done on the EC2 instance, resolving issues with CodeDeploy not unpacking .venv from the artifact.
* **. .venv/bin/activate**: **Reason:** Activates the newly created virtual environment. The . command is used for portability across different shell environments. This modifies the current shell session to use the virtual environment's Python and pip.
* **pip install -r requirements.txt**: **Reason:** Installs all required Python packages into the activated virtual environment. This ensures the application has all its dependencies to run.

**3.5 scripts/start\_app.sh (Example)**

This script starts your application.

**Location:** my\_photo\_app/scripts/

**Example Content:**

Bash

#!/bin/bash

set -e

APP\_DIR="/home/ec2-user/my\_photo\_app"

echo "Starting application..."

cd "$APP\_DIR" || { echo "ERROR: Failed to change directory to $APP\_DIR. Exiting."; exit 1; }

# Reason: Activate the virtual environment before running the application.

# This ensures the application uses its isolated dependencies.

. .venv/bin/activate || { echo "ERROR: Failed to activate venv before starting app. Exiting."; exit 1; }

# Reason: Start your Python application.

# Use 'nohup' and '&' to run in the background and detach from the shell.

# Redirect output to a log file for debugging.

nohup python app.py > app.log 2>&1 &

echo $! > app.pid # Store the process ID for stopping later

echo "Application started."

**Explanation and Reasoning:**

* **runas: ec2-user**: Ensures the application runs as the correct user.
* **. .venv/bin/activate**: Activates the virtual environment.
* **nohup python app.py > app.log 2>&1 &**: Runs the Python application in the background, redirects all output to app.log, and detaches it from the shell, allowing the CodeDeploy hook to complete successfully.
* **echo $! > app.pid**: Saves the Process ID (PID) of the background process to app.pid. This is essential for the stop\_app.sh script to identify and terminate the running application.

**3.6 scripts/stop\_app.sh (Example)**

This script stops your application.

**Location:** my\_photo\_app/scripts/

**Example Content:**

Bash

#!/bin/bash

set -e

APP\_DIR="/home/ec2-user/my\_photo\_app"

echo "Stopping application..."

cd "$APP\_DIR" || { echo "ERROR: Failed to change directory to $APP\_DIR. Exiting."; exit 1; }

PID\_FILE="app.pid"

if [ -f "$PID\_FILE" ]; then

PID=$(cat "$PID\_FILE")

echo "Found PID $PID in $PID\_FILE. Attempting to kill process."

kill -TERM "$PID" || { echo "WARNING: Could not gracefully kill process $PID. Trying harder..."; kill -KILL "$PID"; }

rm "$PID\_FILE"

echo "Application stopped."

else

echo "No PID file ($PID\_FILE) found. Application may not be running or was stopped manually."

fi

**Explanation and Reasoning:**

* **runas: ec2-user**: Ensures the script has permissions to kill processes owned by ec2-user.
* **Checks app.pid**: Looks for the PID file created by start\_app.sh.
* **kill -TERM "$PID"**: Attempts to gracefully terminate the application.
* **kill -KILL "$PID"**: If graceful termination fails, forces termination.
* **rm "$PID\_FILE"**: Cleans up the PID file.

**4. AWS Services Setup**

**4.1 IAM Roles**

Ensure the following IAM roles have the necessary permissions:

* **CodeBuild Service Role:**
  + **Permissions:** AmazonS3FullAccess (or more restrictive S3 permissions for your artifact bucket), CloudWatchLogsFullAccess (for logging), and permissions to invoke CodeDeploy deployments. A managed policy like AWSCodeBuildDeveloperAccess might suffice, but often custom policies are needed.
* **CodeDeploy Service Role:**
  + **Permissions:** AWSCodeDeployRole (managed policy) is typically sufficient. This allows CodeDeploy to interact with EC2 instances, S3, and CloudWatch.
* **EC2 Instance Profile Role (for the EC2 instance where the app is deployed):**
  + **Permissions:** AmazonEC2RoleforAWSCodeDeploy (managed policy). This allows the CodeDeploy agent on the EC2 instance to communicate with the CodeDeploy service, download artifacts from S3, and write logs to CloudWatch.

**4.2 EC2 Instance Preparation**

* **Operating System:** An Amazon Linux 2 AMI is recommended.
* **Python:** Ensure Python 3 (e.g., Python 3.11) is installed and accessible via python3 and pip commands. sudo yum install python3 or sudo apt-get install python3 might be necessary.
* **CodeDeploy Agent:** The AWS CodeDeploy agent must be installed and running on the EC2 instance. Follow AWS documentation for agent installation (e.g., sudo yum install -y ruby then wget https://aws-codedeploy-eu-west-1.s3.eu-west-1.amazonaws.com/latest/install && chmod +x ./install && sudo ./install auto).
* **User:** The ec2-user (default on Amazon Linux) is generally suitable and recommended for running your application.

**5. CodePipeline Setup**

The CodePipeline orchestrates the entire process.

* **Source Stage:** Configured to pull code from your GitHub repository (e.g., master branch).
* **Build Stage:** Linked to your AWS CodeBuild project, which uses the buildspec.yml file to build and package the artifact.
* **Deploy Stage:** Linked to your AWS CodeDeploy application and deployment group, which uses the appspec.yml and the deployment scripts to deploy the application to your EC2 instance.

# Deployment Troubleshooting & Service Management Summary

Today, we successfully resolved several issues to get your my\_photo\_app Streamlit application running reliably as a systemd service on your EC2 instance.

#### 1. Deployment File Folder Structure on EC2:

Your application and its dependencies are organized on the EC2 instance as follows:

/home/ec2-user/

└── my\_photo\_app/

├── .venv/ # Python virtual environment (contains `bin/activate`, `bin/streamlit`, etc.)

│ ├── bin/

│ └── lib/

├── my\_photo\_app/ # Your Python package (actual source code)

│ ├── \_\_init\_\_.py

│ ├── app.py # Main Streamlit application file

│ ├── aws\_utils.py # AWS utility functions

│ └── config.py # AWS configuration (S3 bucket name, DynamoDB table name)

├── requirements.txt # Python dependencies

└── install\_dependencies.sh # Script to install dependencies and activate venv

The Systemd service file for managing this application is located at:

* /etc/systemd/system/my\_photo\_app.service

#### 2. Key Learnings & Resolutions:

1. **Initial NameError: name 'custom\_css' is not defined**:
   * **Problem:** This was a Python error in app.py where the custom\_css variable was used by st.markdown() before it was actually assigned its value.
   * **Resolution:** Reordered the code in app.py to ensure all variables are defined before they are used in st.markdown() calls. This was confirmed by CodePipeline passing and initial systemctl status showing status=0/SUCCESS (though the app still wasn't staying active).
2. **Streamlit App Exiting (Service activating (auto-restart) with status=0/SUCCESS)**:
   * **Problem:** When running python app.py directly (not streamlit run), Streamlit processes the script and then exits, even if cleanly. Systemd, expecting a long-running web server, would then continuously try to restart it.
   * **Resolution:** Modified the ExecStart command in /etc/systemd/system/my\_photo\_app.service to use streamlit run /home/ec2-user/my\_photo\_app/app.py --server.port 8501 --server.enableCORS false --server.enableXsrfProtection false. This instructs Streamlit to start its internal web server and keep the process alive. We also learned to always run sudo systemctl daemon-reload after modifying the service file.
3. **Port 8501 is already in use (Service activating (auto-restart) with status=1/FAILURE)**:
   * **Problem:** After changing to streamlit run, the service started failing because another instance of Streamlit (or a zombie process from previous attempts) was still holding port 8501. streamlit run exits with a failure code if it cannot bind to its port.
   * **Resolution:** Identified and manually killed the conflicting process.

#### 3. Essential Commands:

Here are the commands you'll use regularly to manage your application:

**A. Managing the my\_photo\_app.service:**

* **Check Status:** See if your app is running.

Bash

sudo systemctl status my\_photo\_app.service

(Look for Active: active (running))

* **Stop the App:** Gracefully shuts down the Streamlit server.

Bash

sudo systemctl stop my\_photo\_app.service

* **Start the App:** Launches the Streamlit server.

Bash

sudo systemctl start my\_photo\_app.service

* **Restart the App:** Stops and then starts the app (useful after code updates).

Bash

sudo systemctl restart my\_photo\_app.service

* **Enable Auto-Start on Reboot:** Ensures the app starts automatically when the EC2 instance boots up.

Bash

sudo systemctl enable my\_photo\_app.service

* **Reload Systemd Daemon:** **CRUCIAL** if you ever modify the /etc/systemd/system/my\_photo\_app.service file. Must be run before start, stop, or restart to apply service file changes.

Bash

sudo systemctl daemon-reload

**B. Viewing Application Logs:**

* **View Real-time Logs:** Essential for debugging startup issues or seeing application output.

Bash

sudo journalctl -u my\_photo\_app.service -f

(Press Ctrl+C to exit the follow mode)

**C. Handling Port Conflicts (the "Kill Section"):**

* **Identify Process Using a Port (e.g., 8501):**

Bash

sudo lsof -i :8501

(Look for the PID in the output, e.g., streamlit 12345 ec2-user...)

* **Kill a Specific Process by PID:** (Replace <PID> with the actual process ID found above)

Bash

sudo kill -9 <PID>

(Use this with caution, only when you're sure it's the correct process to kill.)

what is the role of Systemd service file?

Show thinking

The **Systemd service file** (like your /etc/systemd/system/my\_photo\_app.service) acts as the instruction manual for the Systemd init system on Linux. Its primary role is to **define and manage how a specific application or daemon runs as a background service** on the operating system.

Here's a breakdown of its key roles:

1. **Defining the Service:** It tells Systemd everything it needs to know to start, stop, and manage your application. This includes:
   * **The command to execute:** What command Systemd should run to start your app (e.g., streamlit run ...).
   * **Working directory:** Where the command should be run from.
   * **User/Group:** Which user and group the service should run as (important for permissions and security).
   * **Environment variables:** Any specific environment variables needed by the application.
2. **Automated Management:**
   * **Auto-start at Boot:** You can configure it to automatically launch your application every time the server starts up (using sudo systemctl enable).
   * **Restart on Failure:** It can be configured to automatically restart your application if it crashes or stops unexpectedly (as you experienced with auto-restart). This is crucial for maintaining uptime.
3. **Standardized Control:**
   * It provides a **consistent and standardized way** to control services using systemctl commands (start, stop, restart, status). This means you don't need to remember specific, complex commands for each individual application.
4. **Dependency Management:**
   * You can specify dependencies (e.g., "start only after networking is up," or "start only after database X is running"). This ensures your application starts in the correct order relative to other system services.
5. **Logging Integration:**
   * It directs the application's standard output and error streams to Systemd's journal, allowing you to easily view logs with journalctl, which is invaluable for debugging.

In essence, the Systemd service file takes your application, which might otherwise just be a simple Python script, and turns it into a robust, manageable, and automatically-restarting background service that's fully integrated with the operating system's lifecycle.

Sources