Scripts Explanation

NOTE: REFER GITHUB FOR SCRIPTS

cloudformation-template.yml Explanation:

🧱 PARAMETERS

```
Parameters:
  AmazonLinuxAMIID: ...
  MyIP: ...
  KeyName: ...
```

💡 Purpose:

- **Dynamic & reusable template.** You don't hardcode values like the AMI ID, IP, or KeyName — good CI/CD hygiene.
- `MyIP` is used to limit SSH access securely using `/32` (single IP access).

🌐 NETWORK INFRA

✅ VPC, IGW, Subnet, Routing

```
VPC, InternetGateway, VPCGatewayAttachment, PublicSubnetA
```

Creates:

- **Custom VPC**: `10.11.0.0/16`
- **One Public Subnet:** `10.11.0.0/20`

- **Internet Gateway** + routing to `0.0.0.0/0` for full internet access.

📦 Route Table + Subnet Association

```
PublicRouteTable + PublicInternetRoute + SubnetRouteTableAssociation
```

- **Makes the subnet public**
- Adds default route to internet

🔒 SECURITY GROUP

```
PublicSecurityGroup
```

Allows:

- **HTTP (80)** & **HTTPS (443)** to **everyone**
- **Your app (8000)** to **everyone**
- **SSH (22)** to only `MyIP` → 👏 this is 🔐 best practice

🧙 IAM ROLES

🪄 CodeDeployServiceRole

```
AWSCodeDeployRole
```

- Allows CodeDeploy to orchestrate deployment

🧠 ServerRole

```
AmazonSSMManagedInstanceCore, AmazonEC2RoleforAWSCodeDeploy, S3, CloudWatch
```

- **SSM** → Terminal access (Session Manager)

- **S3** + CodeDeploy access
- **CloudWatch** for logs

Then wrapped in:

```
DeployRoleProfile
```

→ Bound to the EC2 instance via `IamInstanceProfile`.

🚀 EC2 INSTANCE

```
WebServer
```

Key Features:

- `t2.micro` → Free-tier eligible 💸
- **AMI from SSM** — Always latest Amazon Linux 2023
- Auto-attaches public IP
- Uses the subnet, security group, and IAM profile properly

✅ UserData Highlights:

```
yum install ruby, python3.12, cloudwatch agent
wget & install CodeDeploy Agent
pip install virtualenv
```

- Prepares instance for CodeDeploy
- Ensures Python 3.12 is available (critical for your app)
- Logs finish status to `/var/log/userdata.log` – smart debug spot.

📦 CodeDeploy Resources

```
CodeDeployApplication
CodeDeployDeploymentGroup
```

- Sets up **named deployment app + group**
- Filters EC2 by **tag** `role=webserver`
- Uses default config: `AllAtOnce` (can be updated to rolling if needed)

🚪 Outputs — 👑 Beautiful UX

Output	Purpose
`PublicAppURL`	App access on port 8000
`PublicIP`	For raw debugging/SSH
`CodeDeployAppName`	Useful for CLI deploys
`CodeDeployGroup`	CLI deploy helper
`SSHAccess`	Readable `ssh` command format for devs

🧠 Final Thoughts & Suggestions

✅ This Template:

- **Launches infra + app pipeline in one go**
- Is **production-ready for solo EC2 use case**
- Properly secures SSH
- Installs all dependencies, including Python 3.12 and CodeDeploy agent
- Smart use of tags, outputs, and policies

🔧 Suggestions (Optional):

Suggestion	Why
Use `UpdatePolicy` and `CreationPolicy`	For auto rolling EC2 updates

Add a second subnet & NAT Gateway (later)	For DB/private EC2 segregation
Export logs to CloudWatch in UserData	For tracking UserData failures
Add `Metadata` for CloudFormation Designer	Helps visualize infra
Hook in CodePipeline	For full CI/CD pipeline (GitHub → Build → Deploy)

🧠 TL;DR

This is a **clean, modular, reusable**, and security-conscious CloudFormation template to deploy a SaaS app with a CI/CD pipeline using EC2 + CodeDeploy. You're orchestrating the whole show from **infra to app startup** with zero hand-holding.

---

buildspec.yml Explanation:

⚙️🔍 What is `buildspec.yml`?

This file **tells CodeBuild what to do** at each stage of your build. Think of it like a chef's recipe card for building, testing, and prepping your app for deployment.

It is used **only in the "Build" phase of CodePipeline**, and specifically during the **CodeBuild** stage. So it's not EC2, not deploy, just the **building/testing/packing** part.

🧱 File Structure Breakdown

```
version: 0.2
```

- Version of the `buildspec` schema.

- `0.2` supports multiple phases, environment variables, artifact controls, etc.

🔷 `phases:` – Like workflow stages in your kitchen

🧰 `install:` – Setup the environment

Used to set up your build environment before anything runs.

```
runtime-versions:
  python: 3.12
```

➡️ Use **Python 3.12** runtime in the CodeBuild environment.

```
- echo "Initializing environment..."
- pip install --upgrade pip
- pip install awscli
```

➡️ Standard stuff: log Python version, upgrade pip, install AWS CLI.

```
- export CODEARTIFACT_AUTH_TOKEN=...
- pip config set ...
```

➡️ This authenticates your pip with **AWS CodeArtifact**:

- Gets an **auth token**
- Sets up **pip** to install packages from your CodeArtifact Python repo (instead of PyPI)

🏗️ `build:` – Build the app and run tests

```
- echo "Build started..."
- python3.12 -m venv venv
- source venv/bin/activate
```

➡️ Creates and activates a **Python virtual environment**.

```
- pip install --retries 3 -r requirements.txt
```

➡️ Installs app dependencies from `requirements.txt`.

```
- python -m unittest discover -s tests || echo "Tests failed but continuing"
```

➡️ Runs all unit tests in `tests/`. It won't fail the build if tests fail (due to the `|| echo`), just logs it.

```
- mkdir -p instance
- chmod 775 instance
- touch instance/saasight.db
- chmod 664 instance/saasight.db
```

➡️ Sets up a placeholder **SQLite database file** (`saasight.db`) for your Flask app inside the `instance/` folder.

🧼 `post_build:` – Sanity checks after build

```
- echo "Build completed..."
- test -f run.py || (echo "Missing run.py!" && exit 1)
- test -f config.py || (echo "Missing config.py!" && exit 1)
```

➡️ Double-checks that critical files exist (like your app runner and config). Fails if they're missing.

📦 `artifacts:` – What files should be packed and sent to the next stage

```
files:
  - '**/*'
exclude-paths:
  - 'venv/**/*'
  - '.git/**/*'
  - '**/*.pyc'
  - '**/__pycache__/**/*'
discard-paths: no
```

➡️ Sends **everything except**:

- Your venv folder
- Git metadata
- Python cache files

This artifact is what **CodeDeploy** (or S3/next stage) gets next.

🧩 Where This Fits in CI/CD Pipeline

```
[ GitHub Push ] → CodePipeline → [ CodeBuild ] → (your buildspec.yml here)
→ Artifact → CodeDeploy → EC2
```

- Your `buildspec.yml` runs **only in CodeBuild**.
- It builds your app, runs tests, and prepares artifacts.
- **Output artifacts** from here are passed to CodeDeploy.

🧠 TL;DR – What's This Script Doing?

> "Hey CodeBuild, install Python, setup pip to use CodeArtifact, install dependencies, run tests (even if they fail), prep the app with a dummy DB, check core files exist, then send everything (minus venv/git/cache) to the next stage."

install_dependencies.sh Explanation:

### 🔰 `#!/bin/bash`

> Tells the system to use Bash to execute the script.

`set -e`

> Fail fast: If any command exits with a non-zero status, the script stops right there.
>
> Saves your app from half-broken deployments.

`echo "==== Starting dependency installation at $(date) ===="`

> Logs the start time of deployment for debugging and log visibility.

🐍 Python 3.12 Installation Block

```
if ! command -v python3.12 &> /dev/null; then
```

> Checks if Python 3.12 is already installed on the machine.
>
> If **not** found → proceeds to install it.

```
if command -v dnf &> /dev/null; then
```

> If you're on Amazon Linux 2023, dnf will be present.
>
> Uses `dnf install` for Python 3.12 and related packages.

```
elif command -v yum &> /dev/null; then
```

> If it's Amazon Linux 2, uses yum. But there's no official Python 3.12 —
>
> so it downloads and builds it from **source** (🐌 slower, but stable).

```
sudo ln -sf /usr/local/bin/python3.12 /usr/bin/python3.12
```

> Creates symbolic links so python3.12 and pip3.12 are available globally in terminal.

```
python3.12 --version
```

> Verifies that Python 3.12 was successfully installed and accessible.

📍 Find CodeDeploy Unpacked Directory

```
DEPLOYMENT_DIR=$(find /opt/codedeploy-agent/deployment-root -name "appspec.yml" | head -1 | xargs dirname)
```

> Finds the path where CodeDeploy dropped the app build (artifacts from CodeBuild).
>
> Looks for `appspec.yml`, assumes its folder is the root of your code.

```
if [ ! -d "$DEPLOYMENT_DIR" ]; then
```

> If it can't find that path → kill the script. Means CodeDeploy didn't unpack properly.

🧹 Clean Old App Files

```
rm -rf /home/ec2-user/{app,scripts,static,templates,tests,venv,__pycache__,migrations}
rm -f /home/ec2-user/*.{py,txt,yml}
```

> Deletes existing app code and junk from last deploy.
>
> Prevents conflict with new deploy.

⚠️ **Hard delete**, no backup. Anything custom you left in home directory = gone.

📂 Copy New App Files

```
for item in app scripts static ...; do
  if [ -e "$DEPLOYMENT_DIR/$item" ]; then
    cp -r "$DEPLOYMENT_DIR/$item" /home/ec2-user/
  fi
done
```

> Iterates over key files and folders and copies them from the deployment folder to home.

Only copies if the file/folder **exists** (fail-safe).

📀 Handle Database (saasight.db)

Checks 4 scenarios:

1. Database is part of deployment → copy it ✅

2. Already exists on EC2 → reuse it ✅
3. Legacy location → move it ✅
4. Nothing → create an empty `.db` file

Also sets proper:

```
chown ec2-user:ec2-user ...
chmod 664 ...
```

> Permissions to avoid runtime errors.

📄 Verify Critical Files Exist

```
for file in config.py requirements.txt run.py; do
  if [ -f "/home/ec2-user/$file" ]; then
```

> Verifies if must-have files exist.
>
> If any is missing → error out. Deployment incomplete otherwise.

🧪 Set up Virtual Environment

```
python3.12 -m venv venv
source venv/bin/activate
```

> Creates a clean virtual environment and activates it.

```
pip install -r requirements.txt
```

> Installs all packages needed for your app.

🛡️ Run Migrations + DB Backup

```
cp saasight.db backups/saasight.db.backup-<timestamp>
flask db upgrade
```

> Backs up the existing DB, runs migrations using Flask-Migrate (Alembic).

```
if ! flask db upgrade; then
  cp backup_file ...
```

> If migration fails, restore the backup. Very smart safety net.

✅ Verify App and Gunicorn

```
python -c "from app import create_app; ..."
```

> Makes sure the app can be created from your create_app() factory.
>
> Early crash detection.

```
gunicorn --version
```

> Verifies Gunicorn is available before setting it in systemd.

⚙️ Set Up systemd Service

```
sudo tee /etc/systemd/system/saasight.service <<EOF
...
EOF
```

> Writes a custom systemd unit file:

- Starts Gunicorn with 3 workers on port 8000
- Sets env vars (like DB path, Flask mode)
- Runs the app as user `ec2-user`

```
sudo systemctl daemon-reload
sudo systemctl enable saasight
```

> Makes systemd aware of your service and ensures it starts on boot.

🏁 Final Echo

```
echo "✅ Dependency install finished at $(date)"
```

> Confirms deployment completion with timestamp.

Summary: This Script Does All of This

- Ensures Python 3.12 exists
- Prepares a clean EC2 environment
- Copies new app files

- Sets up virtualenv & dependencies
- Protects your DB (with backup + restore)
- Sets up systemd service to run Gunicorn server
- Verifies app health before starting

---

start_server.sh Explanation:

🔷 Purpose:

This script ensures:

1. **The Flask app can be started cleanly**
2. **Gunicorn is valid**
3. **Systemd service is restarted**
4. **Health checks pass**
5. **Logs show up if it fails**

🔍 Line-by-Line Breakdown

```
#!/bin/bash
set -e
```

- Use Bash to run
- **Exit immediately** if a command fails (no silent fails)

```
echo "Starting SaaSight service..."
cd /home/ec2-user
source venv/bin/activate
```

- Log starting message
- Move into project directory
- Activate virtual environment

✅ Health Check Before Boot

```
python -c "from app import create_app; print('✓ app import OK')"
```

- Ensures that `create_app()` exists and imports properly.

  If this fails → **app is broken**.

```
gunicorn --check-config -w 1 -b 127.0.0.1:8001 run:app
```

- Dry-run Gunicorn startup using `-check-config`
- Runs with:
  - 1 worker (`w 1`)
  - on `127.0.0.1:8001` (a test port)
- Doesn't actually start the server — just checks if config is valid

💡 This is **brilliant** — better to catch Gunicorn failures **before** systemd starts it.

🔄 Restart systemd Service

```
sudo systemctl daemon-reload
sudo systemctl stop saasight || true
sleep 2
sudo systemctl start saasight
```

- Reloads any systemd config changes
- Stops the service **without crashing the script if already stopped**
- Waits 2s

- Starts fresh

```
sleep 5
```

- Give the app time to warm up before checking its pulse.

🩺 Health Verification

```
if systemctl is-active --quiet saasight; then
```

> If service is running:

```
curl -fs http://localhost:8000 && echo "✓ HTTP OK" || echo "⚠ HTTP FAIL"
```

- Sends an HTTP request to the **real port (8000)**.
- If request passes → all green.
- If it fails → logs a warning.

```
else
  echo "✗ SaaSight failed"
  sudo journalctl -u saasight --no-pager -l --since "5 min ago"
  exit 1
fi
```

- If service is **not active**, show last 5 minutes of logs and **fail the deploy**.

```
echo "✓ Server startup completed!"
```

- End message: all good. App is up. Gunicorn blessed. Deploy complete.

💎 TL;DR – This Script Ensures:

Step	Purpose
Import app	Sanity check for app factory
Gunicorn config test	Avoid runtime Gunicorn failures
Restart `saasight`	Clean launch via systemd
Health check on `:8000`	Ensure server is responding to HTTP
Print logs on failure	Fast debugging

---

stop_server.sh Explanation:

🧱 Full Script Breakdown

```
#!/bin/bash
set -e
```

- Tells Linux to use Bash to run this script.
- `set -e` means: *"If anything fails, stop the script immediately."*

```
echo "Stopping SaaSight service..."
```

- Basic log message — gives visibility in logs/CodeDeploy console.

🛑 Check if systemd service is active

```
if systemctl is-active --quiet saasight; then
    echo "Service is running, stopping it..."
    sudo systemctl stop saasight
    echo "Service stopped successfully"
```

- Checks if `saasight` systemd service is currently **running**
- If yes:

    - Stops it cleanly using `systemctl stop`
    - Confirms successful shutdown

```
else
    echo "Service is not running or doesn't exist"
fi
```

- Handles cases where:
    - Service isn't running
    - Or it's not installed yet (e.g. first-time deploy)

🧹 Kill Leftover Gunicorn Processes (just in case)

```
pkill -f "gunicorn.*run:app" || echo "No gunicorn processes found"
```

- Forcefully kills **any rogue Gunicorn** processes running your app (`run:app`)
- Uses `pkill -f` to match full command line
- If no match found → shows fallback message

```
echo "Stop server completed!"
```

- Final confirmation. Marks completion of stop sequence.

🧠 TL;DR – What This Script Does

Step	Purpose
Stops systemd service `saasight`	Ensures app isn't running
Cleans up zombie Gunicorns	Avoids port bind errors
Fails early on unexpected issues	Thanks to `set -e`
Provides meaningful logs	Easy debugging in CodeDeploy console

---

🔗 Where to Use This in `appspec.yml`

```
hooks:
  ApplicationStop:
    - location: scripts/stop_server.sh
```

🧠 Suggestions (Optional):

💡 Enhancement	Why
`systemctl disable saasight`	Prevents auto-start before full install
Log output to file	Add `tee -a /var/log/saasight-stop.log`
Add a `sleep 1` after `stop`	Give systemd a second to fully settle

⚠️ Danger Zones

- `pkill` is forceful — it will kill any `gunicorn run:app` regardless of context. If you somehow run multiple apps, this will nuke all of them.
- If someone renames the app entry point from `run:app` to something else — this won't catch it.

---

appspec.yml Explanation:

```
version: 0.0
```

> This tells AWS you're using CodeDeploy for EC2/On-Prem deployments (not Lambda/ECS).
>
> `0.0` is **mandatory** for this deployment type.

```
os: linux
```

> This confirms you're deploying to a Linux-based instance (e.g., Amazon Linux 2/2023, Ubuntu, etc.)

🎯 `hooks:` — The Real Magic

This is the **step-by-step lifecycle** of the deployment. Each script will be executed in this order (if present):

🔧 `BeforeInstall`

```
BeforeInstall:
  - location: scripts/install_dependencies.sh
    timeout: 300
    runas: root
```

Field	Purpose
`location`	Path to your shell script relative to root of CodeDeploy artifact
`timeout`	Maximum time (in seconds) to run the script
`runas`	Which user to run the script as (root here)

> ✅ This installs Python 3.12, sets up virtualenv, installs packages, prepares DB, systemd config — everything before the app runs.

🛑 `ApplicationStop`

```
ApplicationStop:
  - location: scripts/stop_server.sh
```

```
    timeout: 300
    runas: root
```

> ✅ Stops running Gunicorn/app processes safely before the new code is laid down.

🚀 `ApplicationStart`

```
ApplicationStart:
  - location: scripts/start_server.sh
    timeout: 300
    runas: root
```

> ✅ Starts systemd service for the app, verifies startup success, and does health checks.

🧠 TL;DR – What Each Hook Does

Hook	Script	What It Handles
`BeforeInstall`	`install_dependencies.sh`	Python, venv, packages, DB prep, systemd service config
`ApplicationStop`	`stop_server.sh`	Stop service + kill leftover gunicorns
`ApplicationStart`	`start_server.sh`	Restart systemd, test health, print logs on fail

💡 Suggestions to Enhance

Enhancement	Benefit
Add `ValidateService` hook	Final post-start sanity check (can call a custom script or test endpoint)
Add log output to S3/CloudWatch in scripts	Easier debugging, especially after failure

Use `runas: ec2-user` for ApplicationStart	Unless you're binding to ports <1024, you don't need root — **more secure**
Add `AfterInstall` if you want post-copy logic	This hook runs **after files are unpacked** but **before app starts**

🧩 Deployment Flow Visualized

```
[CodeDeploy Agent on EC2]
    |
    |--- ApplicationStop       → stop_server.sh
    |--- BeforeInstall         → install_dependencies.sh
    |--- [Files copied to /home/ec2-user]
    |--- ApplicationStart      → start_server.sh
```
