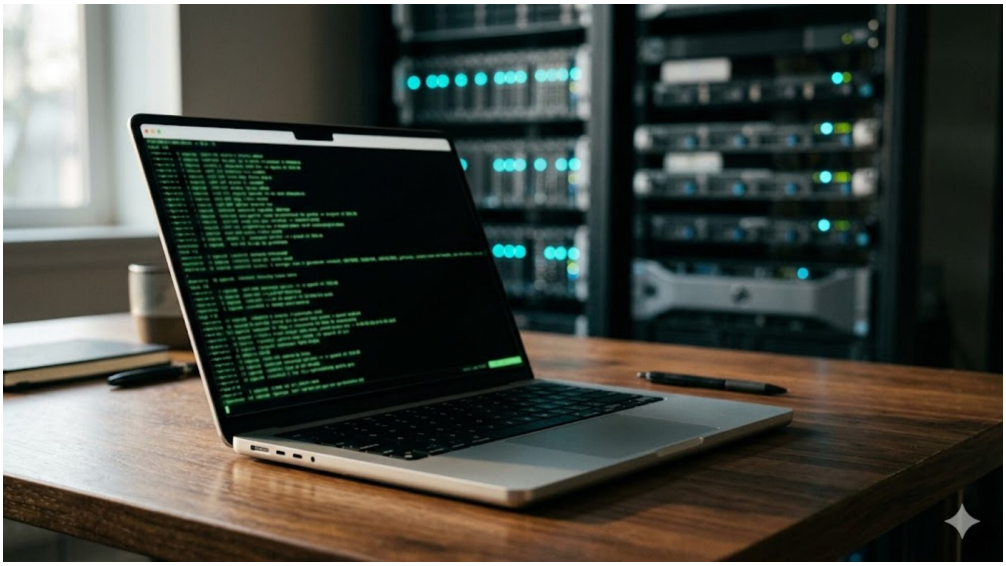
Provisioning AWS EC2 Instances: Console and CLI Methods

Ronald G. Thomas

2026-05-17

2026-05-17 17:08 PDT



Automating cloud infrastructure removes repetitive manual steps and frees attention for the work that matters.

Introduction

I did not really know how virtual servers worked until I had a Shiny app running locally and needed to share it with collaborators over the web. The app was finished, the code was clean, and everything worked on my laptop. 'Works on my machine' is not a deployment strategy.

The challenge had two parts: first, create and launch a virtual server in the cloud; second, configure that server to host the application securely. We cover the first part in full.

There are two well-defined entry points to AWS EC2 provisioning: the interactive web console and the `aws` CLI. The console is a good way to understand what each component does; the CLI is a

better choice once the same configuration has been launched more than twice. We document both paths. They reach the same terminal state (a running Ubuntu instance with a security group, key pair, Elastic IP, and Docker installed). The choice is a matter of how often one expects to repeat the operation.

We cover Layer 11 (Cloud) of the workflow architecture described in A Workflow Construct for the Modern Data Scientist.

Motivations

- A working Shiny application had no deployment path for remote collaborators.
- Managed platforms such as shinyapps.io impose constraints on the software stack; a self-managed server does not.
- Clicking through the EC2 console is error-prone and slow after the first few repetitions; scripted provisioning is reproducible.
- The four bash scripts presented here can be version controlled alongside application code, making the infrastructure part of the project record.

Objectives

1. Understand the four pre-launch components required for any EC2 deployment (key pair, firewall, static IP, domain name).
2. Walk through the console path step by step, including Route 53 domain configuration.
3. Install and configure the AWS CLI, and document the eight environment variables that drive the automation scripts.
4. Write four bash scripts (security group, key pair, instance launch, Docker bootstrap) that replace the manual console workflow.
5. Document connecting to the server, verification, and teardown.

Figure 1: A clean workstation poised at the start of a focused provisioning session.

What is AWS EC2?

Amazon Elastic Compute Cloud (EC2) is a service that provides resizable virtual servers in the cloud. Think of it as renting a computer that lives in an Amazon data center. One chooses the operating system, the amount of memory and storage, and the network configuration. Once launched, one connects over SSH just as one would connect to any remote Linux machine.

The key distinction from a managed hosting platform is control: the operator manages everything on the server, from the operating system packages to the web server configuration. This requires more effort, but it provides complete flexibility over the software stack.

There are several cloud server hosting options: Microsoft Azure, Oracle, Google Cloud, Amazon AWS, Digital Ocean, and Hetzner. Each has its own approach to provisioning, and several offer free or low-cost tiers. AWS is a reasonable choice for a small custom server: the system is well documented and, in my experience, reliable. The AWS free tier includes 750 hours per month of `t2.micro` instance usage for the first 12 months.

The Four Pre-Launch Components

Regardless of whether the console or the CLI is used, four components must be in place before launching an instance:

1. **SSH key pair**: allows encrypted remote login to the server.
2. **Security group (firewall)**: restricts incoming traffic to named ports only.
3. **Static IP address (Elastic IP)**: maintains the link between the domain name and the server across reboots. Without it, a new IP is assigned each time the instance restarts.
4. **Domain name**: provides a human-readable URL rather than a raw IP address.

These components are independent of any specific server instance. You can define multiple instances of each and reuse them across projects.

After launch, post-launch tasks include installing a web server, obtaining an SSL certificate, and configuring a reverse proxy to translate HTTPS (port 443) requests to Shiny (port 3838). Those steps are covered in the companion post on Hosting a Shiny App in Production: Docker Compose, Caddy, and Automatic HTTPS.

Prerequisites

- **Operating system**: macOS 13+ or a Linux distribution with bash 5+.
- **AWS account**: an active account with permission to create IAM users.
- **Already installed (for CLI path)**: Homebrew (macOS) or apt (Debian/Ubuntu), plus `jq` for JSON parsing.
- **Background knowledge**: comfort editing dotfiles and running shell commands; basic familiarity with the AWS console.
- **Time required**: approximately 30 minutes for either path on a first run.

Method A: Console Walkthrough

This section walks through the EC2 web console. It is instructive for understanding the components and their relationships. After two or three manual launches, the CLI path below will be more efficient.

Open the EC2 console at `https://aws.amazon.com/console`, choose the regional service closest to the operator's location (e.g., N. California), and navigate to the EC2 dashboard.

I recommend configuring the four pre-launch components before launching the instance, as it saves back-and-forth with the console.

SSH Key Pair

EC2 supports two approaches: generate the key pair locally and upload the public key, or have EC2 generate the pair and download the private key.

Local generation approach:

```
cd ~/.ssh
ssh-keygen -m PEM
```

Name the key prefix `power1_app.pem` when prompted. The dialog asks for a passphrase; entering one adds additional security but is not required. `ssh-keygen` generates two files: `power1_app.pem` and `power1_app.pem.pub`.

Return to the EC2 dashboard and select **Network & Security > Key Pairs > Actions > Import key pair**. Enter the name `power1_app`, browse to `~/.ssh/power1_app.pem.pub`, and select **Import key pair**.

EC2-generated approach:

Select **Create key pair** in the upper right of the console. Enter a name (e.g., `power1_app`), select RSA key type and `.pem` file format. The private key `power1_app.pem` is offered for download. Place it in `~/.ssh/` and restrict permissions:

```
chmod 400 ~/.ssh/power1_app.pem
```

> **Warning**
>
> If the private key file has permissions broader than 400, SSH will refuse to use it with a 'permissions too open' error.

Firewall (Security Group)

Select **Security Groups** under **Network & Security** in the left panel. Choose **Create security group** and name it `power1_app`. Under **Inbound Rules**, add SSH (port 22) and HTTPS (port 443), each with source **Anywhere IPv4 0.0.0.0/0**.

This creates a firewall that accepts only SSH and HTTPS inbound traffic.

Static IP Address (Elastic IP)

Navigate to **Network and Security > Elastic IPs > Allocate Elastic IP address**. An IP from the EC2 pool is assigned (e.g., `13.57.139.31`).

> ⚠ Warning
>
> Elastic IPs incur charges when allocated but not associated with a running instance. Release unused Elastic IPs immediately to avoid unexpected costs.

Domain Name (Route 53)

From the AWS console, navigate to the Route 53 dashboard (Services > Route 53). Once a domain name is acquired (e.g., `rgtlab.org`), associate it with the static IP:

1. Click **Hosted zones** in the side panel.
2. Click on `rgtlab.org` in the center panel.
3. Check the `rgtlab.org` Type A record.
4. Click **Edit record** in the right panel.
5. Change the IP address in the **Value** field to the allocated Elastic IP.

Selecting and Launching the Instance

From **Instances** in the EC2 dashboard, click **Launch Instances** and follow these steps:

1. **Name the server**: enter `power1_app`.
2. **Select the OS**: choose Ubuntu (a mature Linux distribution).
3. **Choose an instance type**: select `t2.micro` (1 CPU, 1 GiB memory).
4. **Choose a key pair**: select `power1_app` from the dropdown.
5. **Select a security group**: choose the `power1_app` group.
6. **Choose storage**: enter 30 GB of EBS General Purpose SSD (GP2). Thirty gigabytes is the maximum allowed under the free tier. Smaller disk sizes can cause problems during Docker image builds.
7. **Advanced options**: scroll to the bottom and upload the startup script from the Docker Bootstrap section below.
8. Click **Launch Instance**.

After the instance launches, open the Elastic IP dialog under **Network & Security**, select **Actions > Associate Elastic IP address**, and associate the IP with the new instance.

Method B: CLI Scripts

This section documents the CLI approach. Four bash scripts replace the manual console workflow entirely.

Installation

On macOS, Homebrew provides the simplest installation path:

```
brew install awscli jq
aws configure
```

> **Note**
>
> Instructions for installing Homebrew can be found at brew.sh. IAM credential setup is covered in Appendix A below.

The `aws configure` command prompts for four values:

- **AWS Access Key ID** and **AWS Secret Access Key**: from your IAM credentials CSV (see Appendix A).
- **Default region**: e.g., `us-west-1`.
- **Default output format**: `json` is recommended.

Configuration

Nine parameters are required for automated instance generation. Eight are stored as environment variables in `~/.zshrc` (or equivalent) so every script can reference them without hardcoded values.

VPC and Subnet (found on the EC2 dashboard under Your VPCs):

```
export vpc_id="vpc-14814b73"
export subnet_id="subnet-f02c90ab"
```

AMI, instance type, and storage (OS image, server capabilities):

```
export ami_id="ami-014d05e6b24240371"
export instance_type="t2.micro"
export storage_size="30"
```

Key pair name and security group:

```
export key_name="power1_app"
export security_grp="sg-0fef542d93849669c"
```

Static IP (the Elastic IP allocated above):

```
export static_ip="13.57.139.31"
```

A ninth parameter, `proj_name`, can be supplied at call time via the `-p` flag on each script.

Verify after sourcing:

```
aws --version
aws configure list
echo "vpc_id=$vpc_id"
aws ec2 describe-instances \
  --query 'Reservations[].Instances[].InstanceId'
```

If step 3 returns a JSON array (empty or populated), the credentials and region are configured correctly.

Script 1: Create Security Group

Creates a firewall and opens specified ports. The `-n` flag sets the group name; `-p` adds a port. Default behavior opens ports 22 and 443 only.

```
aws_create_security_group.sh -n power1_app -p 22 -p 80 -p 443
```

```
#!/usr/bin/env bash
Help()
{
echo "The script generates a new security group."
echo "The group name is given with the -n flag."
echo "Ports are specified with the -p flag."
echo "Anticipated incoming ports: 22 ssh, 80 http,"
echo "  3838 shiny, 443 https."
echo "Script will fail if group name already exists."
echo "Reads vpc_id from environment variables."
echo "Example:"
echo "  aws_create_security_group.sh \\"
echo "    -n power1_app -p 22 -p 80 -p 443"
}
sg_grp_name=$(basename "$PWD")
while getopts ":hp:n:" opt; do
    case $opt in
        p ) ports+=("$OPTARG") ;;
        n ) sg_grp_name=$OPTARG ;;
        h ) Help
            exit ;;
        * ) echo \
            'error in command line parsing' >&2
            exit 1
    esac
done
echo "sg group name = $sg_grp_name"

aws ec2 create-security-group \
    --group-name "$sg_grp_name" \
```

```
    --description "security group" \
    --tag-specifications \
    "ResourceType=security-group,\
Tags=[{Key=Name,Value=$sg_grp_name}]" \
    --vpc-id "$vpc_id" > temp.txt
wait
security_grp=$(jq -r .GroupId temp.txt)
wait
echo "security group ID = $security_grp"

for i in "${ports[@]}"
do
  aws ec2 authorize-security-group-ingress \
    --group-id "$security_grp" \
    --protocol tcp \
    --port "${i}" \
    --cidr "0.0.0.0/0" > /dev/null
done
```

Script 2: Create Key Pair

Generates an SSH key pair and stores the private key in `~/.ssh/`. The `-k` flag sets the key pair name; if omitted, the current directory name is used.

```
aws_create_keypair.sh -k power1_app
```

```
#!/usr/bin/env bash
Help()
{
echo "The script generates a new key pair."
echo "The key pair name is given with the -k flag."
echo "Script will fail if name already exists."
echo "Example:"
echo "  aws_create_keypair.sh -k power1_app"
}
while getopts 'hk:' flag; do
  case "${flag}" in
    h) Help
      exit;;
    k) key_pair_name=${OPTARG};;
  esac
done
base=$(basename "$PWD")
if [ -z "$key_pair_name" ]
then
  key_pair_name=$base
```

```
fi
echo "key_pair_name is $key_pair_name"

cd ~/.ssh
rm -f "$HOME/.ssh/$key_pair_name.pem"
aws ec2 create-key-pair \
  --key-name "$key_pair_name" \
  --query 'KeyMaterial' \
  --output text > "$HOME/.ssh/$key_pair_name.pem"

wait
chmod 400 "$HOME/.ssh/$key_pair_name.pem"
```

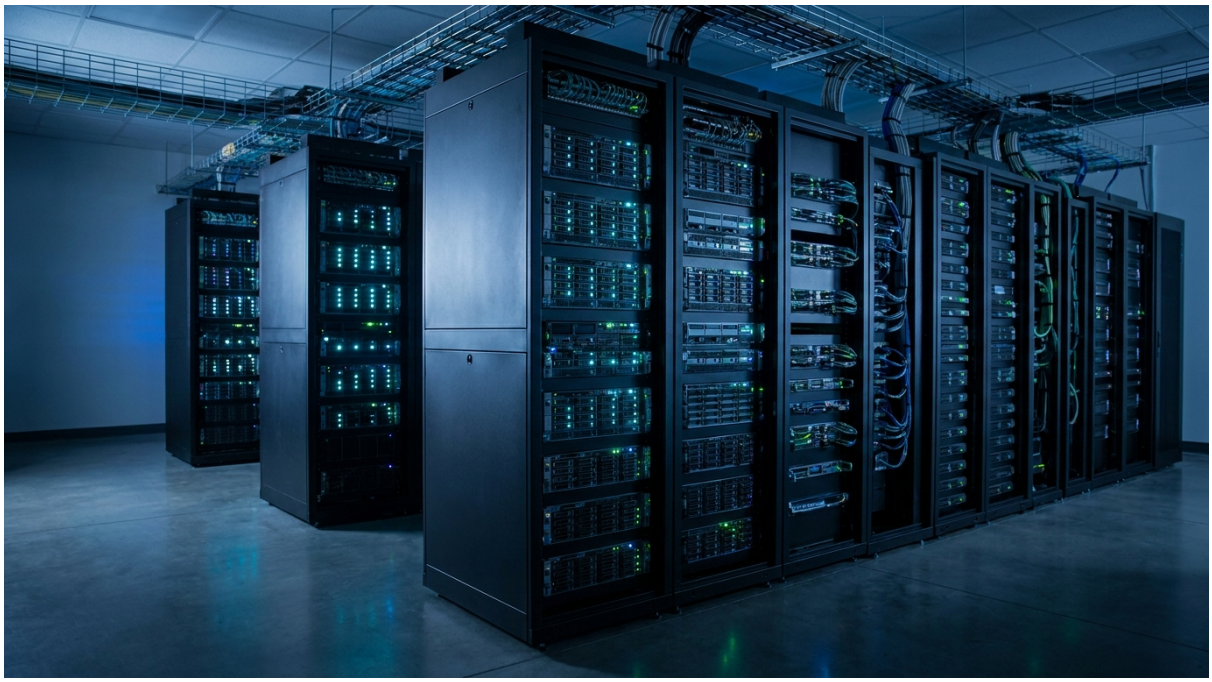


Figure 2: Server racks in a data center conveying the physical infrastructure behind virtual cloud instances.

Script 3: Launch the EC2 Instance

This is the core script. It reads all eight environment variables, launches the instance, and associates the Elastic IP. The `-p` flag sets the project name used for tagging.

```
aws_create_instance.sh -p power1_app
```

```
#!/usr/bin/env bash
Help()
{
```

```
echo "Notes on current parameters:"
echo "Security group should already exist."
echo "  If not, run aws_create_security_group.sh."
echo "Key pair should already exist."
echo "  If not, run aws_create_keypair.sh."
echo "AMI ID is for Ubuntu Linux 22.04 LTS."
echo "Check static IP: nslookup <IP>"
echo ""
echo "Usage:"
echo "  aws_create_instance.sh -p power1_app"
}
while getopts 'hp:' flag; do
  case "${flag}" in
    h) Help
      exit;;
    p) proj_name=${OPTARG};;
  esac
done
base=$(basename "$PWD")
if [ -z "$proj_name" ]
then
  proj_name=$base
fi

aws ec2 run-instances \
  --image-id "$ami_id" \
  --count 1 \
  --instance-type "$instance_type" \
  --key-name "$keypair_name" \
  --security-group-ids "$security_grp" \
  --subnet-id "$subnet_id" \
  --block-device-mappings \
  "[{\"DeviceName\":\"/dev/sda1\",\
\"Ebs\":{\"VolumeSize\":$storage_size}}]" \
  --tag-specifications \
  "ResourceType=instance,\
Tags=[{Key=Name,Value=$proj_name}]" \
  --user-data \
  file://~/Dropbox/prj/c060/aws_startup_code.sh

iid0=$(aws ec2 describe-instances \
  --filters "Name=tag:Name,Values=$proj_name" | \
  jq -r \
  '.Reservations[].Instances[].InstanceId')
echo "$iid0"
read -p "enter instance id:" iid
echo "instance id: $iid"
```

```
aws ec2 associate-address \
  --public-ip "$static_ip" \
  --instance-id "$iid"
```

Script 4: Docker Bootstrap

This user-data script runs automatically on first boot. It installs Docker and Docker Compose on the new Ubuntu instance, then adds the default `ubuntu` user to the `docker` group. Upload it in the **Advanced options** section of the console launch dialog, or pass it via `--user-data` in Script 3.

```
#!/bin/bash
apt update
apt-get install curl -y
apt-get install gnupg -y
apt-get install ca-certificates -y
apt-get install lsb-release -y
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL \
  https://download.docker.com/linux/ubuntu/gpg | \
  sudo gpg --dearmor \
  -o /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg
echo \
  "deb [arch="$(dpkg --print-architecture)" \
  signed-by=/etc/apt/keyrings/docker.gpg] \
  https://download.docker.com/linux/ubuntu \
  "$(. /etc/os-release \
  && echo "$VERSION_CODENAME")" stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list \
  > /dev/null
apt-get update
apt-get install docker-ce docker-ce-cli \
  containerd.io docker-compose-plugin -y
su ubuntu -
usermod -aG docker ubuntu
```

> **Note**
>
> The startup script runs as root on first boot only. If it fails, check `/var/log/cloud-init-output.log` on the instance for debugging output.

Connecting to the Server

Construct a `config` file in `~/.ssh/` to connect with a simple hostname rather than remembering the IP address and key path:

```
Host rgtlab.org
  HostName 13.57.139.31
  User ubuntu
  Port 22
  IdentityFile ~/.ssh/power1_app.pem
```

Then connect with:

```
ssh rgtlab.org
```

> Tip
>
> Set the private key permissions to 400 if not already done: `chmod 400 ~/.ssh/power1_app.pem`

Verification

After launching the instance and associating the Elastic IP, confirm the server is accessible and Docker is running.

CLI path:

```
aws --version
aws configure list
echo "vpc_id=$vpc_id"
aws ec2 describe-instances \
  --query 'Reservations[].Instances[].InstanceId'
```

Server access:

```
ssh rgtlab.org "echo 'SSH connection successful'"
ssh rgtlab.org "docker --version"
ssh rgtlab.org "docker compose version"
```

If all three server checks return expected output, the instance is correctly configured.

Daily Workflow

Command	Action
`aws_create_security_group.sh -n NAME -p P`	Create firewall, open port `P`
`aws_create_keypair.sh -k NAME`	Generate `.pem` key under `~/.ssh/`
`aws_create_instance.sh -p NAME`	Launch tagged EC2 instance
`ssh rgtlab.org`	Connect to the running instance

Command	Action
`ssh rgtlab.org "docker ps"`	View running containers
`ssh rgtlab.org "df -h"`	Check disk usage
`aws ec2 describe-instances`	List all instances in current region
`aws ec2 terminate-instances --instance-ids ID`	Terminate by instance ID
AWS Console > EC2 > Instances	Check instance status via GUI

Things to Watch Out For

1. **Security group names must be unique within a VPC.** If a group with the same name already exists, the create script will fail. Check with `aws ec2 describe-security-groups` before running the script.
2. **Key pair names must also be unique.** A duplicate name causes a CLI error. Delete the old pair first if regeneration is needed.
3. **The Elastic IP must be allocated before association.** If a static IP has not yet been allocated, the associate command in Script 3 will fail silently.
4. **The startup script runs as root on first boot only.** If the bootstrap script has an error, the instance must be terminated and a new one launched. There is no way to re-run user-data on an existing instance.
5. **Environment variables are session-scoped.** Opening a new terminal without sourcing `.zshrc` means the scripts will fail because the variables are not set. Verify with `echo $vpc_id` before running any script.
6. **AWS CLI commands are region-specific.** The `aws configure` region must match the region where the VPC and Elastic IP were allocated.
7. **The free tier storage limit is 30 GB.** Choosing less than 30 GB may seem sufficient initially, but Docker images and system packages accumulate quickly.
8. **The `--cidr "0.0.0.0/0"` rule opens a port to the entire internet.** Restrict this to the operator's IP in production environments.

Uninstall / Rollback

To remove all AWS resources associated with a project, follow these steps in order:

1. **Terminate the instance.** EC2 Console > Instances > select instance > Actions > Terminate instance.
2. **Release the Elastic IP.** EC2 Console > Elastic IPs > select IP > Actions > Release Elastic IP address.
3. **Delete the security group.** EC2 Console > Security Groups > select group > Actions > Delete security groups.
4. **Delete the SSH key pair.** EC2 Console > Key Pairs > select pair > Actions > Delete.

5. **Remove local SSH config.** Delete the entry from `~/.ssh/config` and remove the key file: `rm ~/.ssh/power1_app.pem`

To remove the AWS CLI from the workstation:

```
rm -i ~/.aws/credentials ~/.aws/config
brew uninstall awscli   # macOS
```

> **Warning**
>
> Orphaned Elastic IPs and unused EBS volumes continue to accrue charges. Always run the full teardown when a project is complete.

Figure 3: A calm workspace with a laptop and notebook, representing the planning that goes into infrastructure automation.

What Did We Learn?

Lessons Learned

Conceptual Understanding:

- The EC2 console and the AWS CLI are two interfaces to the same API. Anything clickable in the browser has a corresponding `aws ec2` subcommand.
- A security group is a stateful firewall at the instance level, not the subnet level. Each rule opens a single port to a specified CIDR range.

- Elastic IPs are free while associated with a running instance but incur charges when allocated but unused.
- User-data scripts provide a one-shot bootstrap mechanism. For ongoing configuration management, tools such as Ansible or cloud-init are more appropriate.
- Cloud servers are ordinary Linux machines running in someone else's data center. The mental model of 'remote laptop' is surprisingly accurate.

Technical Skills:

- Writing bash scripts with `getopts` for flag parsing produces reusable, self-documenting CLI tools.
- The `jq` utility is essential for extracting fields from the JSON responses returned by AWS CLI commands.
- Storing infrastructure parameters as environment variables decouples configuration from code, following the twelve-factor app methodology.
- SSH `config` files simplify repeated connections and reduce typing errors.

Gotchas and Pitfalls:

- Forgetting to `chmod 400` the PEM file will cause SSH to reject the connection with a permissions error.
- Deleting a security group while an instance references it will cause the deletion to fail. Terminate the instance first.
- Running out of disk space during Docker builds is a common issue on instances with less than 30 GB of storage.
- The EC2 console interface changes periodically. Screenshots in tutorials may not match the current layout exactly.

Limitations

- These scripts assume a single-instance deployment and do not address auto-scaling groups, load balancers, or multi-AZ redundancy.
- The security group rules open ports to all IPv4 addresses, which is acceptable for development but inappropriate for production.
- The bootstrap script installs Docker but does not configure TLS certificates, reverse proxies, or application-level security.
- IAM credentials stored in `~/.aws/credentials` are long-lived. Rotating to short-lived credentials via IAM roles would be more secure.
- The scripts do not include error recovery or rollback logic. A failure mid-sequence can leave orphaned resources.

Opportunities for Improvement

1. Replace long-lived IAM credentials with IAM roles attached to the EC2 instance profile.
2. Add `set -euo pipefail` to each script for stricter error handling.
3. Migrate the bootstrap script to a cloud-init configuration file for better logging and idempotency.

4. Restrict security group ingress to the operator's current public IP rather than `0.0.0.0/0`.
5. Wrap all four scripts in a single orchestration script with a `--teardown` flag to reverse the entire setup.
6. Explore AWS CloudFormation or Terraform for declarative infrastructure that can be version controlled and reviewed.
7. Use AWS Systems Manager Session Manager for SSH access without opening port 22, improving the security posture.

Wrapping Up

Provisioning an AWS EC2 instance requires the same four components regardless of method: a key pair, a security group, an Elastic IP, and an optional domain name. The console walkthrough reveals what each component does; the CLI scripts make the process repeatable in under two minutes.

The most transferable lesson is that the AWS console and the CLI are two windows into the same API. Once that relationship becomes clear, every console action suggests a corresponding scriptable command, and the scope for automation expands considerably.

In conclusion, four points merit emphasis. First, four bash scripts replace the entire EC2 console provisioning workflow, reducing a multi-step manual process to a single command sequence. Second, eight environment variables parameterize the scripts, making them reusable across projects without hardcoded values. Third, the teardown procedure is equally important: orphaned Elastic IPs and unused volumes accumulate charges that are easy to overlook. Fourth, the console path is valuable for learning the component relationships, while the CLI path is more efficient for any repeated provisioning.

See Also

Related posts:

- Hosting a Shiny App in Production: post-launch server configuration and application deployment.
- Sharing R Code via Docker: R Markdown and Shiny: Dockerfile fundamentals for R Markdown and Shiny images.
- Multi-Laptop macOS Bootstrap: managing configuration files across machines.
- Unix Command-Line Workspace Setup for Data Science: terminal and shell setup.
- A Workflow Construct for the Modern Data Scientist: the reference architecture that contextualizes the Cloud layer.

Key resources:

- AWS CLI Command Reference: EC2
- AWS CLI Getting Started Guide
- Docker Installation on Ubuntu
- jq Manual
- AWS Free Tier Details

Reproducibility

Tested configuration:

Component	Version
Operating system	macOS 13.x
AWS CLI	2.x
Shell	zsh 5.9
jq	1.6+
Homebrew	4.x
Last verified	2026-05-17

The scripts in this post do not require R or Quarto. They require macOS or Linux with bash, Homebrew (macOS) for AWS CLI installation, an active AWS account with IAM credentials, and `jq` installed (`brew install jq`).

To reproduce the full provisioning workflow:

```
aws configure
aws_create_security_group.sh -n power1_app -p 22 -p 80 -p 443
aws_create_keypair.sh -k power1_app
aws_create_instance.sh -p power1_app
```

Appendix A: IAM Credential Setup

1. Log into the AWS console.
2. Search for the **IAM** service and navigate to the IAM dashboard.
3. Select **User groups** and create a group based on the Power User profile. Name it `admin` and include your IAM user in the group.
4. Select **Users** in the left panel, then click **Create User**.
5. Enter a username (e.g. `zenn`) and click **Next**, then **Create User**.
6. Click on the new username. Select the **Security Credentials** tab.
7. Under **Access Keys**, click **Create access key**.
8. Select **Command Line Interface (CLI)** and check the acknowledgement box.
9. Click **Create access key** and then **Download .csv file**.
10. Save the CSV to `~/.aws/`.

Now configure the CLI:

```
aws configure
```

Paste the Access Key ID and Secret Access Key from the CSV. Enter your region (e.g. `us-west-1`) and output format (`json`).

> **Note**
>
> Never commit AWS credentials to version control. Treat them as passwords.

Appendix B: Sample Work Session

A complete CLI provisioning session starting from scratch, assuming `aws configure` has been run and VPC/subnet IDs are set as environment variables.

Step 1. Create a security group:

```
aws_create_security_group.sh -n power1_app -p 22 -p 80 -p 443
```

Step 2. Create the key pair:

```
aws_create_keypair.sh -k power1_app
```

Step 3. Allocate a new Elastic IP via the console and update shell configuration. If the new IP is `204.236.167.50`:

Add to `~/.config/zsh/.zsh_export`:

```
export static_ip='204.236.167.50'
export security_grp='sg-0fda72c2879d6b2ad'
```

Step 4. Launch the instance:

```
aws_create_instance.sh -p power1_app
```

Step 5. Update SSH config with the new IP:

```
# Edit ~/.ssh/config and update the HostName line
# to the new Elastic IP address.
```

Let's Connect

- **GitHub:** rgt47
- **Twitter/X:** @rgt47
- **LinkedIn:** Ronald Glenn Thomas
- **Email:** rgtlab.org/contact

I would enjoy hearing from you if:

- You spot an error or a better approach to any of the code in this post.

- You have suggestions for topics you would like to see covered.
- You want to discuss R programming, data science, or reproducible research.
- You have questions about anything in this tutorial.
- You just want to say hello and connect.

Rendered on 2026-05-17 at 17:08 PDT. Source: ~/prj/rgtlab/posts/cln-aws-ec2-provisioning/index.qmd

Related posts in this cluster

This post is part of the *Clinical Trials and Cloud Deployment* series. Recommended reading order:

1. Post 95: Clinical Trial Data Validation Across Languages
2. **Post 96: Provisioning AWS EC2 Instances** (this post)
3. Post 97: Hosting a Shiny App in Production
4. Post 98: Running ZZedc Independently for Clinical Trials