Setting Up a Virtual Server on AWS EC2 Console

A step-by-step walkthrough for launching, securing, and accessing a lightweight cloud server

Ronald 'Ryy' G. Thomas

2026-02-11

Table of contents

Figure 1: AWS EC2 console workflow for launching a virtual server

Setting up a cloud server – the gateway to hosting web applications on your own terms.

1 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. But "works on my machine" is not a deployment strategy.

The challenge broke down into two parts: first, create and launch a virtual server in the cloud; second, configure that server to provide secure access to the Shiny app. This post covers the first part in detail, walking through every step of setting up an AWS EC2 instance from the console.

I found the process surprisingly approachable once I understood the four components that need to be in place before launching an instance. The terminology can be intimidating at first – elastic IPs, security groups, key pairs – but each concept maps to something straightforward once you see what it does in practice.

More formally, this post documents the GUI path of the Cloud layer (Layer 11) of the Workflow Construct described in post 52. The Cloud layer has two well-defined entry points: the `aws` CLI documented in post 22 and the EC2 web console documented here. Both walks reach the same terminal state (a running Ubuntu instance with security group, key pair, and elastic IP); the choice between them is a matter of how often the operator launches new instances, with the CLI scaling better past the first few iterations.

1.1 Motivations

- I had a working Shiny app and no way to share it with collaborators who needed access from different locations.
- I wanted to understand the fundamentals of cloud server management rather than relying on managed platforms like shinyapps.io.
- I needed a server that could also run Docker containers for reproducible research environments.
- I was frustrated by documentation that assumed prior knowledge of AWS infrastructure concepts.
- I wanted a lightweight, low-cost server I could control completely, including the operating system and installed software.
- I needed the ability to host multiple applications on a single server behind a proper domain name with HTTPS.

1.2 Objectives

1. Launch an Ubuntu-based EC2 instance on the AWS free tier, configured with an SSH key pair and firewall rules.
2. Obtain and associate a static IP address and domain name with the server instance.
3. Configure secure SSH access from a local workstation to the remote server.
4. Prepare the instance for post-launch tasks by running a startup script that installs Docker and Docker Compose.

I am documenting my learning process here. If you spot errors or have better approaches, please let me know.

Figure 2: Settling in for infrastructure setup.

The EC2 dashboard is the control centre for managing virtual servers, security groups, and elastic IP addresses.

2 Prerequisites and Setup

Before beginning, I needed the following in place:

- An AWS account (free tier eligible accounts work for this tutorial).
- A local terminal with SSH installed (macOS Terminal or any Linux shell).
- Basic familiarity with the command line.
- A willingness to spend approximately 30 minutes working through the AWS console interface.

The AWS free tier includes 750 hours per month of `t2.micro` instance usage for the first 12 months, which is sufficient for hosting a lightweight Shiny application.

3 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 centre. You choose the operating system, the amount of memory and storage, and the network configuration. Once launched, you connect to it over SSH just as you would connect to any remote Linux machine.

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

4 The Four Pre-Launch Components

No matter what method one uses to host a Shiny app online, the following pre-launch tasks need to be completed:

1. Obtain a static IP address (e.g., `111.222.333.444`).
2. Obtain a domain name (e.g., `rgtlab.org`).
3. Define a security model (firewall) for the server.
4. Create a public and private RSA key pair.
5. Configure a virtual server by selecting the operating system, CPU count, memory, and storage.
6. Generate the virtual server along with the key pair and security group.
7. Associate the domain name with the static IP address.
8. Associate the IP with the server.

After launch, the post-launch tasks include installing a web server, obtaining an SSL certificate, setting up authentication, and configuring a reverse proxy to translate HTTPS (port 443) requests to Shiny (port 3838).

At first glance these requirements may appear daunting, but on closer inspection each can be met with relative ease through the right tools. Technically, if the sole goal is to get the app on the web and security is not an issue, a static IP, domain name, and reverse proxy are not strictly required. However, without them the server will only communicate via unencrypted HTTP, the URL will be something like `111.222.333.444:3838/power1_shiny`, anyone with the URL can reach the site, and the IP address will change every time the server reboots.

5 Selecting a Hosting Service

There are several cloud-based server hosting options: for example, Microsoft Azure, Oracle, Google Cloud, Amazon AWS EC2, Digital Ocean, and Hetzner. Each has its own approach to setting up a custom virtual server, and several offer free or low-cost service tiers.

I found AWS to be a reasonable choice for setting up a small custom server. It is not the cheapest option, but the system is well documented and, in my experience, reliable.

To start, I opened the EC2 console by visiting:

```
https://aws.amazon.com/console
```

In the console window I chose the regional service closest to my location (N. California). I then created an account, signed in, and navigated to the EC2 dashboard. It is through this dashboard that we define the parameters for the type of server to launch and the mechanisms for communicating with it.

6 Setting Up the Working Environment

Along with selecting a server, I needed to set up a working environment. I recommend setting up these components before launching the server, as it saves some back-and-forth with the console.

The working environment consists of four main components:

1. **Security credential (RSA key pair):** Allows remote and secure login to the virtual server once launched.
2. **Firewall (security group):** Restricts incoming server access by closing all incoming traffic except through specifically named ports.
3. **Static IP address:** 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:** Facilitates collaborator access by providing a human-readable URL rather than a raw IP address.

These components are not directly tied to any specific server. You can define multiple instances of each and associate one of each with each server.

6.1 SSH Key Pair

To securely communicate with the server, I needed to exchange an RSA key pair with AWS. The pair consists of a private key and a public key. EC2 supports two approaches: generate the pair locally and upload the public key, or have EC2 generate the pair and download the private key.

For local generation, I created a directory on my workstation to hold the keys and navigated to it:

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

The `PEM` flag defines the key format. More information on public key authentication is available from the SSH Academy and from the AWS documentation.

In the interactive dialog that follows, I named the key prefix `power1_app.pem`. The dialog asks for a passphrase; entering one adds an additional level of security, but it is not required. The `ssh-keygen` program generates two files: `power1_app.pem` and `power1_app.pem.pub`.

To complete the process, I returned to the EC2 dashboard and selected **Network & Security > Key Pairs > Actions > Import key pair** in the left panel. I entered the name `power1_app`, clicked **Browse**, navigated to the `~/.ssh/power1_app.pem.pub` file, and selected **Import key pair**.

For the EC2-generated approach, I selected **Create key pair** in the upper right of the console page. A form appears asking for a name (e.g., `power1_app`). I selected RSA for key pair type and `.pem` for key file format. The keys were created and the private key `power1_app.pem` was offered for download. I placed it in the `~/.ssh` directory and changed the permissions so only I could access it:

```
sudo chmod 400 power1_app.pem
```

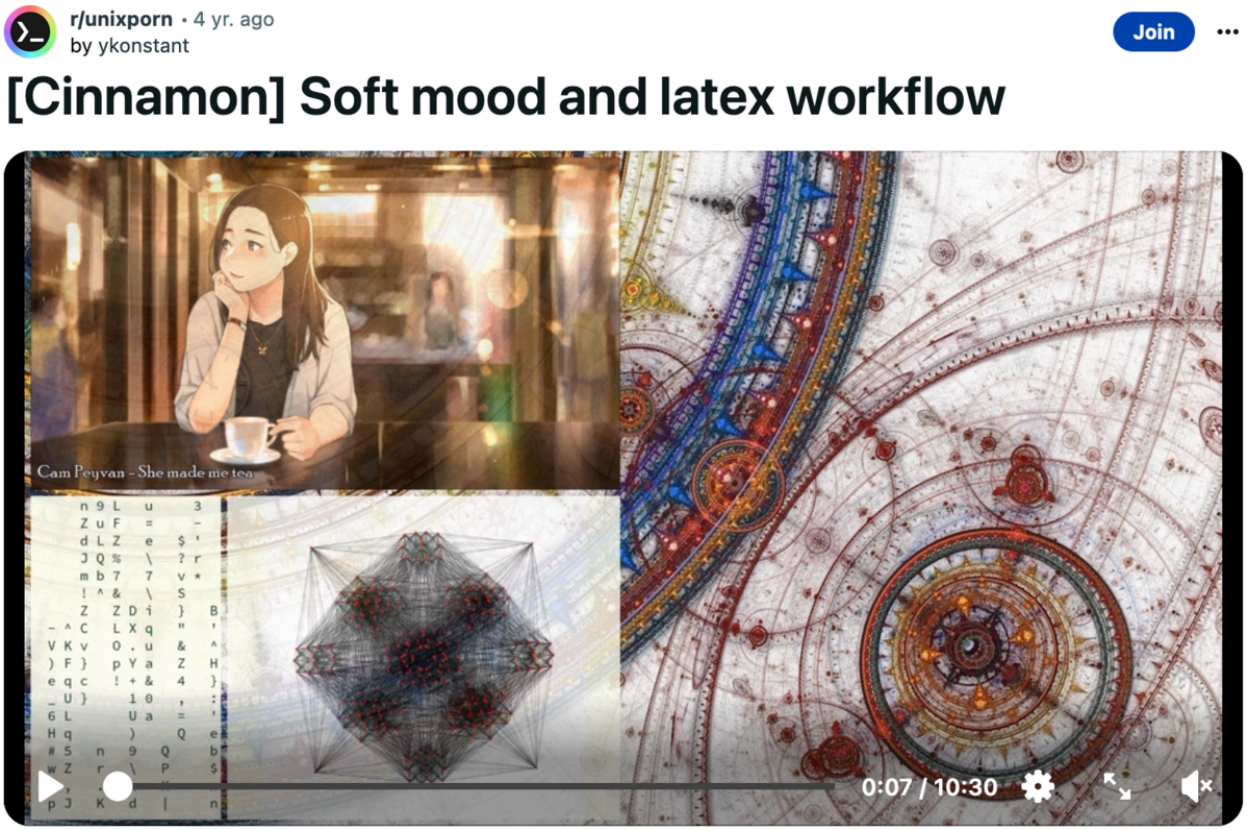

Figure 3: Soft-focus photograph suggesting the calm of working through infrastructure setup

Taking time to set up the working environment properly pays dividends when the server is running in production.

6.2 Firewall (Security Group)

To create a firewall, I clicked on **Security Groups** under **Network & Security** in the left panel. I chose **Create security group** and named it `power1_app`. Under **Inbound Rules**, I selected SSH and HTTPS from the **Type** dropdown menu, and selected **Anywhere IPv4 0.0.0.0/0** for both.

This creates a firewall that leaves open only ports 22 and 443 for SSH and HTTPS incoming traffic, respectively.

6.3 Static IP Address

I used the EC2 Elastic IP service to obtain a static IP. I navigated to **Network and Security**, selected **Allocate Elastic IP**, and an IP was assigned from the EC2 pool of available IPv4 addresses (e.g., 13.57.139.31).

6.4 Domain Name

To obtain a dedicated domain name, I left the EC2 dashboard and went to the Amazon Route 53 dashboard (select **Services** at the top and search for **Route 53**).

Once I acquired a domain name (e.g., `rgtlab.org`), I associated it with my static IP through Route 53:

- Click on **Hosted zones** in the side panel.
- Click on `rgtlab.org` in the centre panel.
- Click the checkbox for the `rgtlab.org` Type=A record.
- Click **Edit record** in the right panel.
- Change the IP address to the assigned static IP (e.g., 111.222.333.444) in the **Value** field.

7 Selecting and Launching the Instance

From **Instances** in the EC2 dashboard, I clicked **Launch Instances** and followed these steps:

1. **Name the server:** I entered `power1_app`.
2. **Select the operating system:** I chose Ubuntu, a mature Linux distribution based on Debian.
3. **Choose an instance type:** I selected `t2.micro`, which provides 1 CPU and 1 GiB of memory. Different instance types offer varying combinations of processors, memory, storage, and network performance.
4. **Choose a key pair:** I selected `power1_app` from the dropdown list.
5. **Select a security group:** I used the `power1_app` security group from my environment.
6. **Choose storage:** I entered 30 GB of EBS General Purpose SSD (GP2). Thirty gigabytes is the maximum allowed in the free tier. In my experience, smaller disk sizes can lead to problems during Docker image builds.
7. **Advanced options:** Under advanced options, I scrolled to the bottom and uploaded my startup script (see the Startup Script section below) to run one time after server generation.
8. **Launch:** I clicked **Launch Instance**.

After the instance launched, I opened the Elastic IP dialog under **Network & Security**, selected the **Actions** button, and associated the IP address with the new instance.

7.1 Accessing the Server

From my laptop, I logged into the server with:

```
ssh -i "~/.ssh/power1_app" ubuntu@rgtlab.org
```

7.2 SSH Config File for Convenience

For convenience, I constructed a `config` file in `~/.ssh`:

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

With this configuration in place, I can SSH into the server with the abbreviated command:

```
ssh rgtlab.org
```

8 Startup Script for Docker Installation

The following startup script installs Docker and Docker Compose on the Ubuntu instance. I uploaded this script in the **Advanced options** section of the instance launch process. It runs once, automatically, when the server first boots.

```
#!/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
```

This script installs the Docker engine, CLI, containerd runtime, and Docker Compose plugin. The final two lines add the default `ubuntu` user to the `docker` group so that Docker commands can be run without `sudo`.

9 Verification

After launching the instance and associating the elastic IP, confirm that the server is accessible and Docker is installed.

```
ssh rgtlab.org "echo 'SSH connection successful'"

ssh rgtlab.org "docker --version"

ssh rgtlab.org "docker compose version"
```

If all three commands return expected output, the instance is correctly configured with SSH access and Docker installed.

10 Daily Workflow

Task	Command
Connect to server	`ssh rgtlab.org`
Check instance status	AWS Console > EC2 > Instances
View Docker containers	`ssh rgtlab.org "docker ps"`
Check disk usage	`ssh rgtlab.org "df -h"`
Review startup log	`ssh rgtlab.org "cat /var/log/cloud-init-output.log"`
Reboot instance	AWS Console > Instance > Actions > Reboot

10.1 Things to Watch Out For

1. **Key pair permissions matter.** If the private key file has permissions broader than 400, SSH will refuse to use it. I learned this the hard way after a "permissions too open" error.
2. **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.
3. **Elastic IPs incur charges when unassociated.** AWS charges for elastic IPs that are allocated but not associated with a running instance. I always release unused elastic IPs.
4. **Security group rules default to deny all inbound.** Forgetting to add SSH (port 22) to the inbound rules means you cannot connect to your own server.
5. **The startup script runs as root.** The `userdata` script in the advanced options runs with root privileges on first boot only. If it fails, you need to check `/var/log/cloud-init-output.log` on the instance for debugging.

11 Uninstall / Rollback

To tear down the EC2 instance and release all associated resources:

1. **Terminate the instance:** EC2 Console > Instances > select instance > Actions > Terminate.
2. **Release the elastic IP:** EC2 Console > Elastic IPs > select IP > Actions > Release.
3. **Delete the security group:** EC2 Console > Security Groups > select group > Actions > Delete.
4. **Remove local SSH config:** Delete the entry from `~/.ssh/config` and remove the key file:

```
rm ~/.ssh/power1_app.pem
```

AWS does not charge for terminated instances. Elastic IPs incur charges only while allocated, so releasing them immediately avoids ongoing costs.

Figure 4: UCSD Geisel Library at sunset, a place for focused technical work

Cloud infrastructure, like library architecture, benefits from careful planning and solid foundations.

12 What Did We Learn?

12.1 Lessons Learnt

Conceptual Understanding:

- Cloud servers are ordinary Linux machines running in someone else's data centre; the mental model of "remote laptop" is surprisingly accurate.
- The four pre-launch components (key pair, firewall, static IP, domain name) are independent of each other and can be configured in any order.
- Security groups act as stateful firewalls: if inbound traffic is allowed, the corresponding outbound response is automatically permitted.
- Elastic IPs provide stability across instance reboots, which is essential for maintaining DNS records.

Technical Skills:

- Generating and managing RSA key pairs with `ssh-keygen` for both local and EC2-generated workflows.
- Configuring SSH `config` files to simplify repeated connections to remote servers.
- Writing `userdata` startup scripts that run automatically on first boot to install Docker and other dependencies.
- Associating elastic IPs, security groups, and key pairs with EC2 instances through the console interface.

Gotchas and Pitfalls:

- Forgetting to change key file permissions to 400 results in an SSH connection refusal that can be confusing if you have not seen it before.
- The EC2 console interface changes periodically; screenshots in tutorials may not match the current layout exactly.
- Running out of disk space during Docker builds is a common issue on instances with less than 30 GB of storage.
- The `userdata` startup script only runs on first boot; if you need to re-run it, you must either recreate the instance or run the commands manually.

12.2 Limitations

- This tutorial covers only the AWS EC2 console approach; other providers (Azure, GCP, Digital Ocean) have different workflows.
- The free tier `t2.micro` instance has limited CPU and memory, which may be insufficient for computationally intensive Shiny applications.
- The startup script installs Docker using a specific method that may need updating as Docker's installation process evolves.
- This post covers only the pre-launch and basic launch steps; configuring a web server, SSL certificates, and reverse proxy are addressed in a separate post.
- AWS pricing and free tier eligibility change over time; readers should verify current terms before proceeding.

- The security group configuration shown here allows SSH and HTTPS from any IP address, which may not meet institutional security requirements.

12.3 Opportunities for Improvement

1. Automate the entire server provisioning process using Terraform or AWS CloudFormation instead of manual console clicks.
2. Use AWS Systems Manager Session Manager for SSH access without opening port 22, improving the security posture.
3. Implement a more restrictive security group that limits SSH access to specific IP addresses rather than 0.0.0.0/0.
4. Create an AMI (Amazon Machine Image) with Docker pre-installed to eliminate the need for a startup script.
5. Explore spot instances for non-production workloads to reduce costs below the free tier threshold.
6. Set up CloudWatch monitoring and billing alerts to avoid unexpected charges.

13 Wrapping Up

Setting up an AWS EC2 instance is a foundational skill for anyone who needs to host web applications, run Docker containers, or manage reproducible research environments in the cloud. The process involves four pre-launch components – key pairs, security groups, elastic IPs, and domain names – followed by a straightforward instance launch sequence.

What I found most valuable in this process was understanding that each component serves a distinct, understandable purpose. The terminology is more intimidating than the actual configuration. Once the working environment is set up, it can be reused across multiple server instances.

The next step from here is configuring the server for Shiny app hosting, which involves installing a web server (Caddy), obtaining an SSL certificate, and setting up a reverse proxy. That workflow is covered in a companion post on Dockerizing Shiny applications.

Main takeaways:

- The four pre-launch components (key pair, firewall, static IP, domain name) are independent and reusable across multiple server instances.
- The free tier `t2.micro` instance with 30 GB storage is sufficient for lightweight Shiny app hosting.
- A startup script can automate Docker installation on first boot, reducing manual configuration.
- SSH config files simplify repeated connections and reduce typing errors.

14 See Also

14.0.1 Related Posts

- Dockerizing a Shiny App for Production – the companion post covering post-launch server configuration.
- Creating a GitHub Dotfiles Repository for Configuration Management – managing configuration files across machines.
- Configure the Command Line for Data Science Development – terminal and shell setup for productive development.

14.0.2 Key Resources

- AWS EC2 User Guide – official documentation for EC2 instances.
- SSH Public Key Authentication – reference for key pair generation and management.
- Docker Installation on Ubuntu – official Docker installation guide.
- AWS Free Tier Details – current free tier eligibility and usage limits.

15 Reproducibility

This tutorial does not involve R code execution. The commands shown are shell commands run either on a local macOS/Linux workstation or on the remote Ubuntu server.

To reproduce the server setup:

```
ssh-keygen -m PEM
sudo chmod 400 ~/.ssh/power1_app.pem
ssh -i "~/.ssh/power1_app" ubuntu@rgtlab.org
```

The startup script can be saved as `aws_startup_code.sh` and uploaded during instance launch, or run manually after SSH access is established.

```
sessionInfo()
```

```
R version 4.5.3 (2026-03-11)
Platform: aarch64-apple-darwin20
Running under: macOS Tahoe 26.4.1

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.5-arm64/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.5-arm64/Resources/lib/libRlapack.dylib;  LAPACK version

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
```

```
time zone: America/Los_Angeles
tzcode source: internal

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

loaded via a namespace (and not attached):
 [1] compiler_4.5.3  fastmap_1.2.0   cli_3.6.5       tools_4.5.3
 [5] htmltools_0.5.9 parallel_4.5.3  otel_0.2.0      yaml_2.3.12
 [9] rmarkdown_2.31  knitr_1.51      jsonlite_2.0.0  xfun_0.57
[13] digest_0.6.39   rlang_1.1.7     evaluate_1.0.5
```

16 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.