Hosting a Shiny App in Production: Docker Compose, Caddy, and Automatic HTTPS

Ronald G. Thomas

2026-08-28

2026-08-28 11:00 PDT

A reverse proxy is a lighthouse: one fixed, well-lit entrance standing in front of whatever is actually running behind it.

Introduction

I did not really know what stood between a working Shiny app and a 'Go to app' link a collaborator could actually click until I had both pieces separately and no path connecting them. Provisioning AWS EC2 Instances covers the first half: a running Ubuntu instance with a static IP, a firewall, and Docker installed. That post explicitly defers the second half, 'installing a web server, obtaining an SSL certificate, and configuring a reverse proxy,' to a companion post. This is that post.

The gap matters because the two halves fail in different ways. A misconfigured security group returns a connection timeout; a missing reverse proxy returns a working app that only its own operator can reach, over plain HTTP, with no certificate and no authentication. The server exists.

Nothing points a browser at it correctly. Getting from 'the app runs on port 3838' to 'the app is available at `https://example.org/`, encrypted, authenticated, and unattended' is its own set of decisions, not a footnote to provisioning.

This post documents that path end to end: a minimal Dockerfile for the Shiny app, a `docker-compose.yml` that wires the app together with a Caddy reverse proxy and a Watchtower auto-updater, and a Caddyfile that handles TLS certificate acquisition and basic authentication in about a dozen lines. We build a small proof-of-concept app locally, then walk it through to a production deployment reachable at a custom domain.

We cover the remainder of Layer 11 (Cloud) of the workflow architecture described in A Workflow Construct for the Modern Data Scientist, continuing directly from Provisioning AWS EC2 Instances.

Motivations

- A Shiny app that only runs on `localhost` has no path to collaborators; managed platforms such as shinyapps.io impose constraints on the package stack that a self-managed server does not.
- Manually installing a certificate and configuring a webserver by hand does not survive a server rebuild; the configuration needs to live in version control next to the application code.
- Pushing a code change should not require an SSH session and a manual `docker pull`; the deployment should update itself.
- A reverse proxy is the one piece of infrastructure that has to be right before anything else is reachable, so it deserves its own careful treatment rather than an afterthought at the end of a provisioning post.

Objectives

1. Build and test a minimal Shiny app locally, using only base R and a handful of reactive widgets.
2. Containerize the app with a single-purpose Dockerfile and push the image to a container registry.
3. Write a three-file configuration set (Dockerfile, Caddyfile, `docker-compose.yml`) that a CI workflow can rebuild and redeploy without manual intervention.
4. Configure Caddy to handle automatic TLS certificate acquisition, HTTPS-to-Shiny reverse proxying, and HTTP basic authentication.
5. Add Watchtower so a pushed image update propagates to the running server without an SSH session.

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

What is a Reverse Proxy?

A reverse proxy is a server that sits in front of one or more backend services, accepts incoming client requests, and forwards them to the appropriate backend before returning the response to the client. From the outside, the proxy is the only thing visible; the Shiny process behind it never receives a direct connection from the internet.

> **Note**
>
> A reverse proxy is the mirror image of a forward proxy. A forward proxy sits in front of *clients* and hides their identity from the servers they contact. A reverse proxy sits in front of *servers* and hides their identity (and their internal ports) from clients.

Three responsibilities land on the reverse proxy in this setup:

1. **TLS termination**: the proxy holds the certificate and decrypts incoming HTTPS traffic; the Shiny container never sees TLS at all.
2. **Port translation**: browsers expect HTTPS on port 443; Shiny Server listens on port 3838. The proxy maps one to the other.
3. **Authentication**: HTTP basic auth checked at the proxy layer blocks unauthenticated requests before they ever reach the Shiny process.

We use Caddy rather than nginx or Apache because Caddy automates certificate acquisition and renewal through Let's Encrypt with no separate `certbot` step. Supplying a domain name in the Caddyfile is sufficient; Caddy requests, installs, and renews the certificate on its own.

Prerequisites

- A running server with a static IP, an open firewall on ports 22, 80, and 443, and Docker installed. See Provisioning AWS EC2 Instances for the console and CLI paths to reach this state.
- A registered domain name with its `A` record pointed at the server's static IP.
- A GitHub (or GitLab) account with a repository to hold the application code and deployment configuration, and permission to push container images to that account's container registry.
- Local R and Shiny installed for development and local testing.
- Docker installed locally, primarily to build and test the image before pushing.

Method

Step 1: The Shiny Application

We start with a small, self-contained example: a Shiny app that calculates statistical power for a two-sample t-test as a function of the standardized effect size. The example is intentionally minimal, using only base R functions with a handful of reactive widgets, so that later steps are about deployment mechanics rather than application complexity.

```
ui <- fluidPage(
  titlePanel("Power Calculator for Two Group Parallel Designs"),
  sliderInput("N", "Total Sample Size:", min = 0, max = 300, value = 100),
  plotOutput("plot"),
  verbatimTextOutput("eff")
)

server <- function(input, output, session) {
  delta <- seq(0, 1.5, .05)
  pow <- reactive(sapply(delta, function(x) power.t.test(input$N, d = x)$power))
  eff <- renderText(power.t.test(input$N, power = .8)$d)
  output$plot <- renderPlot({
    plot(delta, pow(), cex = 1.5, ylab = "power")
    abline(h = .8, col = "red", lwd = 2.5, lty = 4)
    abline(v = eff(), col = "blue", lwd = 2.5, lty = 4)
  })
  output$eff <- renderText(
    paste0("Std. effect detectable with power 80% = ", eff())
  )
}
shinyApp(ui, server)
```

Test the app locally before containerizing anything:

```
R -e "library(shiny); runApp('shiny_app/app.R', launch=TRUE)"
```

Confirm the widget and plot behave as expected in a browser before moving on. Debugging application logic is far easier on a local workstation than inside a container running on a remote server.

Step 2: The Dockerfile

A minimal Dockerfile based on the `rocker/shiny` image builds the app into a container that starts Shiny Server automatically:

```
FROM rocker/shiny:4.2.0
RUN rm -rf /srv/shiny-server
COPY shiny_app/* /srv/shiny-server/
USER shiny
CMD ["/usr/bin/shiny-server"]
```

> **Note**
>
> Placing `app.R` at `/srv/shiny-server/`, the default location Shiny Server watches, means the container needs no further configuration to find and serve the app. The `USER shiny` line drops root privileges before the server starts.

For a new deployment, pin the base image to whatever `rocker/shiny` tag is current at build time rather than reusing an old pin verbatim.

Build and test the image locally:

```
docker build -t power1_shiny --platform linux/amd64 .
docker run -d -p 3838:3838 --platform linux/amd64 power1_shiny
```

> **Tip**
>
> On Apple Silicon, `rocker` images are built for `linux/amd64`. Always include `--platform linux/amd64` on both `build` and `run`, or the container will fail to start with a 'no matching manifest' error.

Visit `http://localhost:3838` to confirm the containerized app behaves the same as the local development version.

Step 3: Push the Image to a Registry

Once the image builds and runs correctly, push it to a container registry so the production server can pull it without needing the source tree. GitHub Container Registry (`ghcr.io`) is a reasonable default when the application repository already lives on GitHub:

```
echo $GITHUB_TOKEN | docker login ghcr.io -u <github-username> --password-stdin

docker build -t ghcr.io/<github-username>/power1-shiny:v1.0 \
  --platform linux/amd64 .
docker push ghcr.io/<github-username>/power1-shiny:v1.0
```

`$GITHUB_TOKEN` here is a personal access token with `write:packages` scope, not the account password. GitLab's container registry works identically; substitute `registry.gitlab.com` for `ghcr.io` and a GitLab personal access token for `$GITHUB_TOKEN`.

Step 4: The Caddyfile

The Caddyfile is the entire reverse-proxy configuration. It names the domain, maps HTTPS traffic to the Shiny container's port, and adds basic authentication, in about a dozen lines:

```
# Generate a bcrypt password hash first:
# > caddy hash-password --plaintext <password>

example.org {
    basicauth /power1/* {
        analyst $2a$14$Zkx19XLiW6VYouLHR5NmfOFU0z2GTNmpkT/5qqR7hx4IjWJPDhjvG
    }
    root * /srv
    handle_path /power1/* {
        reverse_proxy power1:3838
    }
    file_server
}
```

Three things happen here:

1. **`example.org`** is the site block. Supplying a bare domain name is sufficient for Caddy to request a certificate from Let's Encrypt on first start and renew it automatically thereafter.
2. **`basicauth /power1/*`** requires HTTP basic auth credentials for any request under `/power1/`, checked against a bcrypt hash generated once with `caddy hash-password`.
3. **`handle_path /power1/*`** strips the `/power1` prefix and forwards the remaining path to `power1:3838`, the Shiny container's hostname and port inside the Compose network.

> **Warning**
>
> Never commit a plaintext password to the Caddyfile. Generate the bcrypt hash with `caddy hash-password` and commit only the hash.

Step 5: docker-compose.yml

Docker Compose wires the Shiny container, the Caddy reverse proxy, and a Watchtower auto-updater into a single multi-container application, started and stopped as a unit:

```
version: "3.7"

services:
  power1:
    image: ghcr.io/<github-username>/power1-shiny:v1.0
    restart: unless-stopped
    expose:
      - "3838"

  caddy:
    image: caddy:2.6.4-alpine
    restart: always
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - $PWD/Caddyfile:/etc/caddy/Caddyfile
      - $PWD/site:/srv
      - caddy_data:/data
      - caddy_config:/config
    depends_on:
      - power1
    environment:
      - HOST="example.org"
      - EMAIL="admin@example.org"

  watchtower:
    image: containrrr/watchtower
    container_name: watchtower
    restart: always
    environment:
      WATCHTOWER_POLL_INTERVAL: 3600
      WATCHTOWER_CLEANUP: "true"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - /home/ubuntu/.docker/config.json:/config.json

volumes:
  caddy_data:
  caddy_config:
```

The `power1` service exposes port 3838 internally only; it is never published to the host directly, only reachable through `caddy`'s reverse proxy on the internal Compose network. `caddy` publishes 80 and

443 to the host, terminates TLS, and forwards to `power1:3838` exactly as the Caddyfile specifies. `watchtower` polls the registry every hour (`WATCHTOWER_POLL_INTERVAL: 3600`) and, on finding a newer image tag than the one currently running, pulls it and restarts the `power1` container automatically.

> **Note**
>
> Watchtower authenticates to a private registry using the same Docker config the host used to log in manually. Mounting `~/.docker/config.json` read-only into the Watchtower container gives it that credential without duplicating it anywhere.

Step 6: A Minimal Landing Page

A single static `index.html` under a `site/` directory gives the Caddy `file_server` directive something to serve at the domain root, alongside the `/power1/` path handled by the reverse proxy:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Power Calculators</title>
  </head>
  <body>
    <h1>Power Calculators</h1>
    <p>Power for a two-sample t-test</p>
    <a href="./power1/">Go to app</a>
  </body>
</html>
```

At this point the deployment repository holds four files alongside the application code:

```
.
├── Caddyfile
├── Dockerfile
├── docker-compose.yml
├── shiny_app/
│   └── app.R
└── site/
    └── index.html
```

Step 7: Deploy

Connect to the server, clone the repository holding the four configuration files, authenticate to the registry, and bring the stack up:

```
ssh example.org
git clone https://github.com/<github-username>/power1-shiny-deploy.git
cd power1-shiny-deploy
echo $GITHUB_TOKEN | docker login ghcr.io -u <github-username> --password-stdin
docker compose up -d
```

The app is now available at `https://example.org/power1/`, over HTTPS, behind basic authentication, with a certificate that renews itself.

A subsequent code change follows a shorter path: build and push a new image tag from the workstation, and Watchtower picks it up on the server within the polling interval, with no SSH session required:

```
docker build -t ghcr.io/<github-username>/power1-shiny:v1.1 \
  --platform linux/amd64 .
docker push ghcr.io/<github-username>/power1-shiny:v1.1
```

> **Tip**
>
> Watchtower matches on the image reference, not the tag alone. Pushing a new tag (`v1.1`) requires updating `docker-compose.yml` on the server and re-running `docker compose up -d` once; pushing to the same tag (`latest`) lets Watchtower redeploy without any server-side edit, at the cost of losing an explicit version history.

Daily Workflow

Task	Command
Rebuild and push a new image	`docker build ... && docker push ...`
Check running containers	`ssh example.org "docker compose ps"`
Tail Caddy logs	`ssh example.org "docker compose logs -f caddy"`
Force an immediate redeploy	`ssh example.org "docker compose pull && docker compose up -d"`
Regenerate a basic-auth hash	`caddy hash-password --plaintext <password>`
Stop the stack	`ssh example.org "docker compose down"`

Things to Watch Out For

1. **The Shiny container's port must stay internal.** Publishing `3838` directly to the host bypasses TLS termination and basic auth entirely. Only `caddy` should have published ports.

2. **Caddy needs port 80 open, not only 443.** Let's Encrypt's HTTP-01 challenge, and Caddy's own HTTP-to-HTTPS redirect, both use port 80 as well. Restricting the firewall to 443 only breaks certificate issuance.
3. **The domain's `A` record must resolve before the first Caddy start.** Caddy requests a certificate on startup; if DNS has not propagated yet, the request fails and Caddy will keep retrying with exponential backoff rather than serving over HTTPS immediately.
4. **A stale bcrypt hash locks everyone out, including the operator.** Regenerate it with `caddy hash-password` and confirm the new hash is copied correctly; a single mistyped character produces a silent authentication failure with no useful error message.
5. **Watchtower needs read access to the registry credentials, not write access.** Mount `config.json` read-only; Watchtower only pulls images, never pushes them.
6. **`docker-compose.yml` and the Caddyfile both hardcode the domain name.** Renaming the domain requires editing both files and restarting the stack; there is no single source of truth for the hostname in this minimal setup.
7. **An `.pem` platform mismatch produces a container that starts and immediately exits.** On Apple Silicon, forgetting `--platform linux/amd64` during the build produces an image that runs locally under emulation but may behave differently once deployed to an amd64 server; always build for the target platform explicitly.

Uninstall / Rollback

To tear down the deployment while keeping the underlying server:

```
ssh example.org
cd power1-shiny-deploy
docker compose down -v
```

The `-v` flag also removes the `caddy_data` volume, which discards the acquired TLS certificate; a fresh `docker compose up -d` will request a new one from Let's Encrypt.

> **Warning**
>
> Let's Encrypt rate-limits certificate issuance per domain. Repeatedly tearing down and rebuilding `caddy_data` during testing can trigger that limit; use Caddy's staging CA during development to avoid it.

Figure 2: A calm workspace with a laptop and notebook, representing the planning that goes into a production deployment.

What Did We Learn?

Lessons Learnt

Conceptual Understanding:

- A reverse proxy is the single point of contact between the internet and every backend service behind it; getting its configuration right is a precondition for everything else working, not a final polish step.
- TLS termination at the proxy means the application container never handles certificates at all, which keeps the Dockerfile focused on the application alone.
- Docker Compose's internal network means service names (`power1`, `caddy`) resolve to each other without any manual IP bookkeeping.

Technical Skills:

- Caddy's automatic HTTPS collapses what used to be a multi-tool `certbot` plus nginx configuration into a domain name and a handful of directives.
- `docker compose` treats a multi-container application as a single deployable unit, with `up`, `down`, `pull`, and `logs` operating across all defined services at once.
- Watchtower turns 'push an image, redeploy manually' into 'push an image,' removing the SSH step from routine updates.

Gotchas and Pitfalls:

- Publishing the application port directly, even temporarily for debugging, defeats both TLS and authentication and is easy to forget to revert.
- A DNS record that has not propagated yet silently blocks certificate issuance rather than producing an obvious error.
- Watchtower's polling interval means a redeploy is not instantaneous; a one-hour interval, appropriate for most projects, is too slow for same-day iteration during active development.

Limitations

- This setup handles a single Shiny app behind a single domain. Hosting multiple independent apps with per-app authentication and isolated containers is closer to what ShinyProxy provides, as noted in Sharing R Code via Docker.
- Basic auth over HTTPS is adequate for a small group of named collaborators; it is not a substitute for a real identity provider for a larger or more sensitive user base.
- The domain name and the registry image reference are both hardcoded in two separate files, which does not scale past one or two deployments without templating.
- Watchtower's polling model introduces deployment latency and, when pointed at a mutable tag, loses the explicit version history that a pinned tag provides.

Opportunities for Improvement

1. Move the domain name and image reference into a single `.env` file referenced by both `docker-compose.yml` and a templated Caddyfile.
2. Replace polling-based Watchtower with a GitHub Actions workflow that pushes a new image and then triggers an SSH-based redeploy directly, removing the up-to-one-hour delay.
3. Add a `HEALTHCHECK` to the Shiny Dockerfile so Compose can detect and restart a hung container automatically.
4. Evaluate ShinyProxy for deployments that need to host more than one app or isolate sessions per authenticated user.
5. Add structured logging and a log-shipping sidecar so Caddy and Shiny logs are queryable outside the container filesystem.

Wrapping Up

Getting a Shiny app from 'runs on my laptop' to 'reachable at a secure URL' comes down to three configuration files sitting next to the application code: a Dockerfile that packages the app, a Caddyfile that handles TLS and authentication in about a dozen lines, and a `docker-compose.yml` that wires the two together with an auto-updater. None of the three files is long; the value is in how cleanly they separate concerns.

In conclusion, four points merit emphasis. First, the reverse proxy, not the application container, is the correct place for TLS termination and authentication, which keeps the Dockerfile focused purely on the application. Second, Caddy's automatic HTTPS removes an entire category of manual certificate management that used to require a separate tool. Third, Watchtower converts routine updates into a single `docker push`, eliminating the SSH step for the common case. Fourth, this

three-file pattern generalizes past a single power calculator: any containerized Shiny app can reuse the same Caddyfile and Compose structure with only the image reference and domain changed.

See Also

Related posts:

- Provisioning AWS EC2 Instances: the prerequisite server, firewall, and static IP setup this post continues from.
- Sharing R Code via Docker: R Markdown and Shiny: the Dockerfile fundamentals for R Markdown and Shiny images, and where ShinyProxy is discussed as an alternative for multi-user deployments.
- A Workflow Construct for the Modern Data Scientist: the reference architecture that contextualizes the Cloud layer.

Key resources:

- Caddy Documentation
- Docker Compose File Reference
- Watchtower Documentation
- Let's Encrypt: How It Works
- ShinyProxy

Reproducibility

Tested configuration:

Component	Version
Base image	rocker/shiny:4.2.0
Caddy	2.6.4-alpine
Docker Compose	v2.x
Watchtower	containrrr/watchtower (latest)
Last verified	2026-08-28

> **!** Important
>
> **Epistemic status.** The application build, the Caddyfile syntax, and the `docker-compose.yml` structure in this post are adapted from a working deployment I ran previously against GitLab's container registry and a `rgtlab.org` domain. For this post the registry examples were updated to GitHub Container Registry to match this site's own infrastructure, and the domain and credentials were genericized to `example.org` placeholders. The adapted configuration has not been re-run end to end against a fresh server as part of writing this post; verify each step against a current server before relying on it for a real deployment.

To reproduce the deployment from scratch:

```
docker build -t ghcr.io/<github-username>/power1-shiny:v1.0 \
  --platform linux/amd64 .
docker push ghcr.io/<github-username>/power1-shiny:v1.0
ssh example.org "cd power1-shiny-deploy && docker compose up -d"
```

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 2026-08-28 at 11:00 PDT. Source: ~/prj/rgtlab/posts/cln-shiny-docker-caddy-hosting/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
3. **Post 97: Hosting a Shiny App in Production** (this post)
4. Post 98: Running ZZedc Independently for Clinical Trials