Private RAG Deployment - Live-Scripting Workflow

Andreas Wittmann

Andreas Wittmann IT-Beratung GmbH
andreas.wittmann@anwi.gmbh

2025-11-04

Contents

1 Summary

This document outlines a live-scripting methodology for deploying RagFlow, an open-source RAG platform, on AWS using Terraform, through executable code blocks that ensure reproducible infrastructure setup. It covers HTTPS configuration, Docker containerization, SSH access, and appendices for GitHub Actions runners and remote desktop tools, bridging manual workflows with automated deployments.

- **HTML Version:** private-rag.html
- **PDF Version:** private-rag.pdf

2 Building Privacy-Focused RAG Platform

2.1 Project Overview

This project uses the live-scripting methodology to create an example configuration of the open-source project RagFlow. Live-scripting represents an intermediate approach between manual command-line work and fully automated CI/CD pipelines, enabling the development of documented, shareable, and reproducible solutions.

Using Infrastructure as Code methods with Terraform, an AWS environment is established as a platform for RagFlow installation. The example demonstrates all necessary configuration and software tool installations required for the project. Appendices cover additional topics such as GitHub Runner setup and graphical user interface configuration.

2.2 Live-Scripting Methodology

The deployment uses live-scripting methodology to transform static documentation into executable workflows. Code blocks execute directly in Emacs (F4 key) or copy to any terminal, creating a bridge between documentation and execution.

Based on the [live-scripting methodology](https://github.com/andreaswittmann/live-scripting), this approach provides:

- Executable documentation with current and functional code blocks
- Multi-format publishing from single source to Org, HTML, and PDF
- Reproducible deployments with consistent infrastructure setup

- Version control integration through plain-text format

2.2.1 Implementation Options

Emacs Users Configure live-scripting with this Emacs Lisp function:

```
(defun send-line-to-vterm ()
  "Send region if active, or current line to vterm buffer."
  (interactive)
  (if (region-active-p)
      (send-region "*vterm*" (region-beginning) (region-end))
    (my-select-current-line)
    (send-region "*vterm*" (region-beginning) (region-end)))
  (deactivate-mark))

(global-set-key [f4] 'send-line-to-vterm)
```

Non-Emacs Users Copy code blocks to any terminal - the methodology maintains precision across tools.

The deployment follows phases from environment setup through advanced features, with each step documented using executable code blocks for reproducibility.

2.3 Project Components

- Self-hosted RagFlow with local document processing for privacy control
- Production-ready foundation with SSL/HTTPS, Docker containerization, and infrastructure automation
- Advanced RAG capabilities including document ingestion, vector search, and knowledge base management
- Reproducible methodology using live-scripting workflow for consistent deployment

2.4 Expected Deployment Outcome

The workflow produces a functioning HTTPS-enabled RagFlow instance on AWS, capable of document processing and RAG-based AI responses with full user control.

3 Foundation & Prerequisites

3.1 Environment Setup

These steps establish the development environment and validate prerequisites.

3.1.1 Setting the Project Foundation

This establishes the base project directory and validates AWS network prerequisites.

```
# Listing: Project environment setup
# Establishing the base directory for the RAG project
bash
PROJECT_DIR="/Users/$(whoami)/LocalProjects/RagFlow/private-rag"
cd "$PROJECT_DIR"
```

```
pwd
ls -la

# Verify we're in the right location
echo "Working in: $PROJECT_DIR"

# Set AWS profile for this session
export AWS_PROFILE=default  # Adjust as needed
echo $AWS_PROFILE

## create default VPC if it does not exist
aws ec2 create-default-vpc
## check if the default vpc is created
aws ec2 describe-vpcs --filters "Name=isDefault,Values=true" --query "Vpcs[0].VpcId" --output
↪  text
```

Result: Project environment is configured and AWS default VPC is available for deployment.

3.1.2 Validating AWS Prerequisites

This confirms AWS environment configuration for deployment requirements.

```
# Listing: AWS environment validation
# Ensuring AWS access and basic services availability

# Test basic AWS connectivity
aws s3 ls

# Verify AWS identity and permissions
aws sts get-caller-identity

# Verify SSH key is available for EC2 access
ls -la ~/.ssh/

# Validate required tools are installed
terraform --version
git --version
docker --version
```

Result: AWS credentials and required tools are validated for infrastructure deployment.

3.2 Summary

The foundation phase establishes the development environment and validates prerequisites. Project directory structure is configured, AWS credentials verified, and required tools confirmed operational. This groundwork enables reproducible infrastructure deployment through live-scripting methodology.

4 Infrastructure as Code

4.1 SSH Key Generation for Secure Access

This creates dedicated SSH keypairs for EC2 instance access and stores the public key for Terraform.

```
# Listing: SSH key generation for EC2 access
# Creating dedicated SSH keys for secure instance access

cd "$PROJECT_DIR"

# Generate a new SSH key pair specifically for RagFlow
ssh-keygen -t rsa -b 4096 -f ~/.ssh/ragflow_key -N ""
y
# Display the public key for use in Terraform variables
cat ~/.ssh/ragflow_key.pub

# Store public key in environment variable
export TF_VAR_SSH_PUBLIC_KEY="$(cat ~/.ssh/ragflow_key.pub)"
echo "SSH public key configured: $TF_VAR_SSH_PUBLIC_KEY"
```

Result: SSH keypair is generated and public key is exported for infrastructure provisioning.

4.2 Terraform Configuration and Planning

This prepares the Terraform configuration files and validates the infrastructure setup.

```
# Listing: Terraform environment preparation
# Setting up infrastructure configuration

cd "$PROJECT_DIR/terraform"

# Create terraform variables file with our configuration
cat > terraform.tfvars << EOF
# Generated tfvars file - $(date)
aws_region       = "eu-central-1"
instance_type    = "t3.medium"
root_volume_size = 50
root_volume_type = "gp3"
allowed_ip       = "0.0.0.0/0"
ssh_public_key   = "${TF_VAR_SSH_PUBLIC_KEY}"
ec2_name         = "EC2-RagFlow"
environment      = "development"
project          = "ragflow"
ssh_cidr_blocks  = ["0.0.0.0/0"]
EOF

# Review our configuration
cat terraform.tfvars

# Initialize Terraform
terraform init

# Format and validate our configuration
terraform fmt
terraform validate
```

Result: Terraform configuration is initialized and validated for deployment.

4.3 Infrastructure Deployment Planning

This creates the deployment plan and reviews resource changes before applying them.

```
# Listing: Terraform deployment planning
# Creating and reviewing the deployment plan

# Generate deployment plan
terraform plan -out=tfplan -var-file=terraform.tfvars

# Create human-readable plan
terraform show -json tfplan > tfplan.json
cat tfplan.json | jq > tfplan.pretty.json

# Review what resources will be created
echo "Resources to be created:"
cat tfplan.json | jq '.resource_changes[] | {address: .address, action: .change.actions[0]}'

# Summary of planned changes
cat tfplan.json | jq
↪  '.resource_changes | group_by(.change.actions[0]) | map({action: .[0].change.actions[0], count: l
```

Result: Deployment plan is generated and resource changes are reviewed for approval.

4.4 Infrastructure Deployment Execution

This executes the Terraform plan to create the AWS infrastructure resources.

```
# Listing: AWS infrastructure deployment
# Creating the infrastructure resources

# Execute the deployment plan
terraform apply tfplan
terraform apply
yes

# Capture and display outputs
terraform output

# List our EC2 instances
aws ec2 describe-instances \
  --query
↪  'Reservations[*].Instances[*].[InstanceId, InstanceType, Tags[?Key==`Name`]|[0].Value, State.Name
↪  \
  --output text

# Get our instance details - get only the first/latest instance ID
instance_id=$(aws ec2 describe-instances \
  --filters "Name=tag:Name,Values=EC2-RagFlow"
↪  "Name=instance-state-name,Values=running,pending,stopping,stopped" \
  --query 'Reservations[*].Instances[*].[InstanceId]' \
  --output text | head -n1)
echo "Instance ID: $instance_id"
```

```
public_ip=$(aws ec2 describe-instances \
  --instance-ids "$instance_id" \
  --query 'Reservations[*].Instances[*].[PublicIpAddress]' \
  --output text)
echo "Public IP: $public_ip"
exit
```

Result: AWS infrastructure is deployed and EC2 instance details are captured for subsequent configuration.

4.5 SSH Configuration for Easy Access

This establishes convenient SSH access configuration and tests connectivity to the deployed instance.

```
# Listing: SSH configuration setup
# Establishing convenient SSH access
pwd
# Test initial SSH connection
ssh -i ~/.ssh/ragflow_key ubuntu@"$public_ip" 'whoami && pwd'
yes
# Create SSH config entry for easy access
open ~/.ssh/config # we may delete an old entry first, manually
cat >> ~/.ssh/config << EOF
# SSH over EC2 - RagFlow Privacy RAG Project
Host EC2-RagFlow
    HostName $public_ip
    User ubuntu
    IdentityFile ~/.ssh/ragflow_key
EOF
## use a perl one-liner to grep these lines from the ssh-config file.
perl -nE 'print if /Host EC2-RagFlow/ .. /yesEOF/' ~/.ssh/config
#bbedit ~/.ssh/config

# Test SSH with hostname
ssh EC2-RagFlow 'whoami && date'
```

Result: SSH configuration is established and remote access to the EC2 instance is verified.

4.6 EC2 Instance Selection and Upgrading

4.6.1 Recommended EC2 Instance Types for RAG Workloads

Here is a list of recommended EC2 instance types suitable for RAG (Retrieval-Augmented Generation) workloads like RagFlow, which typically require significant memory for model inference, vector embeddings, and data processing. I've focused on instances with at least 20 GB RAM, prioritizing memory-optimized types (R series) for better performance. All recommendations are for the `eu-central-1` region, with approximate on-demand pricing (as of late 2024; prices may vary—check AWS Pricing Calculator for current rates). Network speed refers to the maximum bandwidth.

Instance Type	RAM (GB)	CPU (vCPUs)	Network Speed	Approx. Price/Hour (USD)	Notes
r5.xlarge	32	4	Up to 10 Gbps	$0.25	Balanced mem
r6g.xlarge	32	4	Up to 10 Gbps	$0.21	Graviton-based
r7g.xlarge	32	4	Up to 15 Gbps	$0.20	Latest Gravito
r5.2xlarge	64	8	Up to 10 Gbps	$0.50	Doubles RAM
m5.2xlarge	32	8	Up to 10 Gbps	$0.38	General-purpos

Recommendations

- **For most RAG setups**: Start with `r5.xlarge` or `r6g.xlarge` for a balance of memory, performance, and cost.
- **For high-performance needs**: Opt for `r7g.xlarge` if network speed is critical (e.g., for real-time inference).
- **Scaling up**: If 32 GB isn't enough, `r5.2xlarge` provides 64 GB without overkill.
- **Considerations**: R series are optimized for memory-intensive tasks. Use Spot instances for cost savings (~50-70% off on-demand). Monitor with CloudWatch for actual usage. If your workload involves GPUs (e.g., for advanced ML), consider P or G series, but they exceed basic RAG needs and are pricier.

4.6.2 Upgrading to a Larger Instance Type with Terraform

The best and fastest way to upgrade your EC2 instance to one with more memory is to update your Terraform configuration and apply the changes. This approach is automated, ensures consistency, and avoids manual errors.

Steps to Upgrade

1. **Update the Terraform Variables**:
   - Edit `terraform/terraform.tfvars` and change `instance_type = "t3.medium"` to your desired instance type (e.g., `instance_type = "r5.xlarge"`).
   - Alternatively, update the default in `terraform/variables.tf` if you prefer not to modify `tfvars`.
2. **Apply the Changes**:
   - Navigate to the `terraform/` directory in your terminal.
   - Run `terraform plan` to preview the changes (it will show the instance type update).
   - Run `terraform apply` to execute the upgrade. Terraform will:
     - Stop the existing instance.
     - Change the instance type.
     - Restart the instance with the new configuration.
   - This process typically takes 5-10 minutes, depending on AWS availability.

Why This Method Is Best/Fastest

- **Automation**: Terraform handles the entire process, ensuring consistency and avoiding manual errors.
- **Minimal Downtime**: The stop/change/start cycle is efficient for instance type changes.
- **No Data Loss**: Your EBS root volume (50 GB) and Elastic IP remain attached.
- **Cost**: Monitor via AWS Cost Explorer. R series instances are reasonably priced.
- **Alternatives Considered**:
  - Manual AWS Console change: Faster for one-off updates but not recommended for Terraform-managed infrastructure, as it can cause drift.

– Creating a new instance: Slower and requires data migration.

If you encounter issues (e.g., instance family incompatibility), Terraform will prompt you. Ensure your AWS credentials are configured and the instance is not in use during the upgrade.

```
# Listing: Upgrading EC2 instance type with Terraform
# Updating instance configuration and applying changes

# Navigate to the Terraform directory
cd "$PROJECT_DIR/terraform"

# Update the instance type in terraform.tfvars (replace 'r5.xlarge' with your desired type)
sed -i 's/instance_type    = "t3.medium"/instance_type    = "r5.xlarge"/g' terraform.tfvars

# Review the updated configuration
cat terraform.tfvars

# Generate and review the deployment plan
terraform plan -out=tfplan -var-file=terraform.tfvars

# Apply the changes to upgrade the instance
terraform apply tfplan

# Verify the instance has been upgraded
aws ec2 describe-instances \
  --filters "Name=tag:Name,Values=EC2-RagFlow" \
  --query 'Reservations[*].Instances[*].[InstanceType, State.Name]' \
  --output text
```

Result: EC2 instance type has been successfully upgraded to the new configuration with increased memory capacity.

4.7 Infrastructure Live-Cycle Operations

This section provides procedures for managing the EC2 instance lifecycle - starting and stopping the instance when returning to work sessions after periods of inactivity.

```
exit
pwd


# Initialize project environment
PROJECT_DIR="/Users/$(whoami)/LocalProjects/RagFlow/private-rag"
cd "$PROJECT_DIR"
export AWS_PROFILE=default

# Get instance ID
instance_id=$(aws ec2 describe-instances --filters "Name=tag:Name,Values=EC2-RagFlow" --query
↪  'Reservations[*].Instances[*].[InstanceId]' --output text | head -n1)
echo $instance_id

### STATUS: Get current instance state
aws ec2 describe-instances --instance-ids "$instance_id" --query
↪  'Reservations[*].Instances[*].[State.Name]' --output text
```

```
### START:  Start and wait for running state
aws ec2 start-instances --instance-ids "$instance_id"
while [ "$(aws ec2 describe-instances --instance-ids "$instance_id" --query
↪  "Reservations[0].Instances[0].State.Name" --output text)" != "running" ]; do echo -n ".";
↪  sleep 1; done; echo "   running"


# Get public IP
public_ip=$(aws ec2 describe-instances --instance-ids "$instance_id" --query
↪  'Reservations[*].Instances[*].[PublicIpAddress]' --output text)
echo "Instance running at: $public_ip"

# Test SSH connection
ssh EC2-RagFlow 'whoami && date && uptime'


### STOP: stop  the instance and wait for stopped state
aws ec2 stop-instances --instance-ids "$instance_id"
while [ "$(aws ec2 describe-instances --instance-ids "$instance_id" --query
↪  "Reservations[0].Instances[0].State.Name" --output text)" != "stopped" ]; do echo -n ".";
↪  sleep 1; done; echo "   stopped"
```

Result: EC2 instance is started and ready for use. Stop commands are provided as well.

4.8 Summary

Infrastructure automation through Terraform provisions the AWS environment. The EC2 instance operates with proper network security configuration. SSH access is established through dedicated keypairs. The infrastructure exists as version-controlled code, enabling consistent reproduction across deployments.

5 Development Environment Setup

5.1 Complete Docker Engine and Tools Installation

This section establishes the complete development environment on the AWS EC2 instance, including Docker Engine, container management tools, GitHub CLI, and Python development utilities. These tools form the foundation for both RagFlow deployment and CI/CD testing.

5.1.1 Docker Engine Installation

Install Docker Engine with proper repository configuration and user permissions.

```
# Listing: Docker Engine installation on EC2
# Setting up containerization platform for applications

ssh EC2-RagFlow
bash
# Remove any old Docker versions that may exist
sudo apt remove docker docker-engine docker.io containerd runc 2>/dev/null || true

# Update system packages
```

```
sudo apt update
sudo apt install -y ca-certificates curl gnupg lsb-release

# Add Docker's official GPG key
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

# Verify GPG key installation
cat /etc/apt/keyrings/docker.gpg | gpg --show-keys

# Add Docker repository
echo \
  "deb [arch=$(dpkg
↪  --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/u
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

# Install Docker packages
sudo apt update
sudo apt install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin
↪  docker-compose-plugin

# Configure Docker to run without sudo (important for CI/CD)
sudo usermod -aG docker $USER

# Apply group changes without logout
newgrp docker

# Test Docker installation
docker run hello-world

# Verify Docker Compose
docker compose version

exit
```

Result: Docker Engine is installed with proper GPG key verification and user permissions configured for passwordless operation.

5.1.2 Docker Service Configuration

Enable Docker as a system service for automatic startup and management.

```
# Listing: Docker service setup
# Configuring Docker for automatic startup and monitoring

ssh EC2-RagFlow

# Enable Docker service to start on boot
sudo systemctl enable docker

# Start Docker service
```

```
sudo systemctl start docker

# Verify service status
sudo systemctl status docker  --no-pager

exit
```

Result: Docker service is configured to start automatically on EC2 instance boot.

5.1.3 Lazydocker Installation

Install Lazydocker for convenient terminal-based Docker container management.

```
# Listing: Lazydocker installation
# Installing terminal UI for Docker management

ssh EC2-RagFlow

# Download and install Lazydocker
curl
↪  https://raw.githubusercontent.com/jesseduffield/lazydocker/master/scripts/install_update_linux.sh
↪  | bash

# Add Lazydocker to PATH
echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.bashrc

# Reload bashrc to apply PATH changes
source ~/.bashrc

# Verify installation
lazydocker --version

exit
```

Result: Lazydocker is installed and available in the PATH for terminal-based Docker management.

5.1.4 GitHub CLI Installation

Install the GitHub CLI tool for repository and runner management.

```
# Listing: GitHub CLI installation
# Installing gh command-line tool for GitHub integration

ssh EC2-RagFlow

# Update package list
sudo apt update

# Install dependencies
sudo apt install -y curl unzip

# Add GitHub CLI GPG key
```

```
curl -fsSL https://cli.github.com/packages/githubcli-archive-keyring.gpg | sudo dd
↪  of=/usr/share/keyrings/githubcli-archive-keyring.gpg
sudo chmod go+r /usr/share/keyrings/githubcli-archive-keyring.gpg

# Add GitHub CLI repository
echo "deb [arch=$(dpkg
↪  --print-architecture) signed-by=/usr/share/keyrings/githubcli-archive-keyring.gpg] https://cli.gi
↪  | sudo tee /etc/apt/sources.list.d/github-cli.list > /dev/null

# Install GitHub CLI
sudo apt update
sudo apt install -y gh

# Verify installation
gh --version

exit
```

Result: GitHub CLI is installed and ready for repository and runner management operations.

5.1.5 Docker and Docker Compose Verification

Verify all Docker components are properly installed and functional.

```
# Listing: Docker installation verification
# Confirming all Docker components are operational

ssh EC2-RagFlow

# Check Docker version
docker --version

# Check Docker Compose version (new recommended format)
docker compose version

# Verify Docker daemon is running
docker ps

# List available Docker images
docker images

exit
```

Result: Docker Engine and Docker Compose are verified as fully operational.

5.1.6 Python Development Tools Installation

Install uv (fast Python package installer) and development libraries required for Python projects.

```
# Listing: Python development tools installation
# Installing uv and development dependencies
```

```
ssh EC2-RagFlow

# Install uv (fast Python package installer and runner)
curl -LsSf https://astral.sh/uv/install.sh | sh

# Verify uv installation
uv --version
uvx --version

# Install ICU development libraries and build tools
sudo apt-get update
sudo apt-get install -y \
    libicu-dev \
    pkg-config \
    build-essential

# Verify build tools
gcc --version
pkg-config --version

exit
```

Result: Python development tools including uv, ICU libraries, and build essentials are installed and verified.

5.1.7 Development Environment Summary

Verify the complete development environment is ready for deployment.

```
# Listing: Development environment verification
# Confirming all tools are installed and operational

ssh EC2-RagFlow

# Display installed versions
echo "=== Development Environment Status ==="
echo "Docker:"
docker --version
echo ""
echo "Docker Compose:"
docker compose version
echo ""
echo "GitHub CLI:"
gh --version
echo ""
echo "Python Tools:"
uv --version
echo ""
echo "Build Tools:"
gcc --version | head -1
echo ""
echo "System Information:"
uname -a
```

```
echo ""
echo "Available Disk Space:"
df -h /

exit
```

Result: Complete development environment is verified and ready for RagFlow deployment and CI/CD operations.

5.2 Summary

The development environment setup establishes necessary tools for containerized application deployment and CI/CD workflows. Docker Engine provides the containerization platform, Lazydocker offers terminal management, GitHub CLI enables repository integration, and Python development tools support application requirements. This foundation enables RagFlow deployment and GitHub Actions runner functionality on the EC2 instance.

6 Core RagFlow Deployment

6.1 Basic Ragflow Setup

This section deploys RagFlow on the EC2 instance using Docker Compose following the QuickStart guide.

```
# Listing: RagFlow deployment on EC2
# Installing and starting RagFlow RAG platform

# SSH into the EC2 instance
ssh EC2-RagFlow

# Clone the RagFlow repository
cd ~; pwd
git clone https://github.com/infiniflow/ragflow.git

# Navigate to the RagFlow directory
cd ~/ragflow/docker

# Start RagFlow using Docker Compose
docker compose up -d

# Verify the deployment
docker ps

# Getting SERVICE_IP
public_ip=$(curl -s http://checkip.amazonaws.com)
echo $public_ip

# RagFlow URL
echo "RagFlow is accessible at: http://$public_ip:80"

# Shutting down RagFlow (optional)
docker compose stop
# Clean up RagFlow (optional)
```

```
docker compose down -v # removes volumes as well
cd ~; rm -rf ragflow


exit
```

Result: RagFlow is successfully deployed and running on the EC2 instance.
Url http://18.184.181.110:80 URL https://18.184.181.110

6.2 HTTPS Setup with Self-Signed Certificate

This section provides step-by-step instructions for setting up HTTPS access to RAGFlow using a self-signed SSL certificate. The setup allows the domain to be reachable via IP address over HTTPS, reusing the existing Nginx configuration files with minimal modifications.

Prerequisites

- Docker and Docker Compose installed on your server
- Ports 80 and 443 open on your server
- Server's public IP address (replace <SERVER$_{IP}$> with your actual IP throughout these instructions)
- Basic knowledge of command-line operations

6.2.1 Step 1: Generate Self-Signed SSL Certificate**

Generate a self-signed certificate for your server's IP address. This certificate will be used by Nginx for HTTPS connections.

```
bash
pwd
ssh EC2-RagFlow
cd
cd ~/ragflow/docker/nginx

# Retrieve the SERVER_IP
curl -s http://checkip.amazonaws.com
export SERVER_IP=$(curl -s http://checkip.amazonaws.com)
echo $SERVER_IP


# Create a directory for SSL certificates
mkdir -p ssl

# Generate a private key
openssl genrsa -out ssl/privkey.pem 2048

# Generate a certificate signing request (CSR) for your IP
openssl req -new -key ssl/privkey.pem -out ssl/cert.csr -subj
↪  "/C=US/ST=State/L=City/O=Organization/CN=localhost"
openssl req -new -key ssl/privkey.pem -out ssl/cert.csr -subj
↪  "/C=US/ST=State/L=City/O=Organization/CN=<$SERVER_IP>"

# Generate the self-signed certificate valid for 365 days
openssl x509 -req -days 365 -in ssl/cert.csr -signkey ssl/privkey.pem -out ssl/fullchain.pem
```

```
# Set appropriate permissions
chmod 600 ssl/privkey.pem
chmod 644 ssl/fullchain.pem

# Verify the certificate was created
ls -la ssl/
cat ssl/fullchain.pem
cat ssl/privkey.pem
cat ssl/cert.csr
```

After running these commands, you should see the certificate files in the ssl directory. The fullchain.pem contains the certificate, and privkey.pem contains the private key.

6.2.2 Step 2: Modify Nginx Configuration**

Update the existing ragflow.https.conf file to use your IP address instead of a domain name.

```
# Go to the ragflow project directory
cd ~/ragflow
pwd


# Copy the existing HTTPS config as a backup
cp docker/nginx/ragflow.https.conf docker/nginx/ragflow.https.ip.conf


# Edit the configuration file to replace domain with IP and proxy path
sed -i "s/your-ragflow-domain.com/localhost/g" docker/nginx/ragflow.https.ip.conf
sed -i 's|http://ragflow:|http://ragflow-cpu:|' docker/nginx/ragflow.https.ip.conf
# Only if using GPU version, uncomment the following line
#sed -i 's|http://ragflow:|http://ragflow-gpu:|' docker/nginx/ragflow.https.ip.conf


# Verify the changes
cat docker/nginx/ragflow.https.ip.conf
```

The modified configuration now listens on your IP address for both HTTP (redirecting to HTTPS) and HTTPS connections.

6.2.3 Step 3: Update Docker Compose Configuration**

Modify the docker-compose.yml file to mount the self-signed certificates and use the updated Nginx configuration.

```
pwd
cd ~/ragflow/docker

# Add certificate volumes to the ragflow service in docker-compose.yml
# Insert these lines under the existing volumes section for ragflow-cpu and ragflow-gpu
↪  services
```

```
# For ragflow-cpu service (around line 37-44):
# Add after: - ./entrypoint.sh:/ragflow/entrypoint.sh
      - ./nginx/ssl/fullchain.pem:/etc/nginx/ssl/fullchain.pem:ro
      - ./nginx/ssl/privkey.pem:/etc/nginx/ssl/privkey.pem:ro
      - ./nginx/ragflow.https.ip.conf:/etc/nginx/conf.d/ragflow.confls


# For ragflow-gpu service (around line 86-93):
# Add after: - ./entrypoint.sh:/ragflow/entrypoint.sh
      - ./nginx/ssl/fullchain.pem:/etc/nginx/ssl/fullchain.pem:ro
      - ./nginx/ssl/privkey.pem:/etc/nginx/ssl/privkey.pem:ro
      - ./nginx/ragflow.https.ip.conf:/etc/nginx/conf.d/ragflow.conf

# Verify the docker-compose.yml file syntax
docker compose config
```

The docker-compose.yml now mounts your self-signed certificates and the modified Nginx configuration into the container.

6.2.4 Step 4: Restart RAGFlow Services**

Stop the existing services and restart them with the new HTTPS configuration.

```
cd ~/ragflow/docker
# Stop all running services
docker compose down

# Start the services (use --profile cpu or gpu as needed)
docker compose --profile cpu up -d

# Wait for services to start
sleep 30

# Check that services are running
docker compose ps

# Verify Nginx is listening on ports 80 and 443
docker compose exec ragflow-cpu netstat -tlnp | grep :80
docker compose exec ragflow-cpu netstat -tlnp | grep :443
```

After restarting, RAGFlow should be accessible via HTTPS at https://<SERVER_IP>. Note that browsers will show a security warning due to the self-signed certificate.

6.2.5 Step 5: Test HTTPS Access**

Verify that the HTTPS setup is working correctly.

```
# Test HTTP to HTTPS redirect
curl -I http://$SERVER_IP

# Test HTTPS connection (ignore certificate warnings)
curl -k -I https://$SERVER_IP
```

```
# Check certificate details
openssl s_client -connect $SERVER_IP:443 -servername $SERVER_IP < /dev/null 2>/dev/null |
↪  openssl x509 -noout -dates -subject

# Verify RAGFlow API is accessible over HTTPS
curl -k https://$SERVER_IP/v1/system/healthz
```

The HTTP request should redirect to HTTPS, and the HTTPS connection should succeed despite certificate warnings. The API status endpoint should return a valid response.

6.3 Use Case: Access RagFlow via SSH-Tunnel

This creates an SSH tunnel to access RagFlow securely from a local browser.

```
# Listing: Testing RagFlow via SSH tunnel

ssh -L 9380:localhost:80 EC2-RagFlow
# Open RagFlow in your browser at http://localhost:9380
exit
```

When the SSH tunnel is established, RagFlow can be accessed via local browser at: http://localhost:9380

Result: RagFlow is accessible via SSH tunnel and ready for document processing and RAG operations.

6.4 Summary

RagFlow deployment establishes the core RAG platform. The application configures with Docker containerization, HTTPS setup with self-signed certificates, and SSH tunnel access for secure local browser connectivity. The system integrates document processing and knowledge base management.

7 Advanced Features - Document Processing

7.1 Document Ingestion and Processing

This demonstrates uploading and processing documents for RAG functionality.

7.1.1 Upload Documents

Access RagFlow via browser and upload sample documents for processing.

7.1.2 Configure Knowledge Base

Create a knowledge base and configure document parsing settings.

7.1.3 Test RAG Queries

Test retrieval-augmented generation with uploaded documents.

7.2 Summary

Document processing enables advanced knowledge management while preserving privacy. Local processing handles sensitive documents without transmitting data to external services.

8 Infrastructure Cleanup

8.1 Cleanup Philosophy

Infrastructure as Code enables rapid deployment, but equally important is the ability to cleanly tear down resources to avoid unnecessary costs. This section provides procedures for complete cleanup of all deployed resources.

8.2 Stop RagFlow Services

Before destroying infrastructure, gracefully stop all running services.

```
# Listing: Stopping RagFlow services
# Gracefully shutting down all containers

ssh EC2-RagFlow

# Stop RagFlow Docker containers
cd ~/ragflow
docker-compose down

# Verify containers are stopped
docker ps -a

# Optional: Remove Docker images to free disk space
docker system prune -a -f

# Exit SSH session
exit
```

Result: RagFlow services are stopped and Docker containers are removed from the EC2 instance.

8.3 Terminate EC2 Instance with Terraform

Use Terraform to cleanly destroy all AWS infrastructure resources.

```
# Listing: Destroying AWS infrastructure
# Removing all Terraform-managed resources

cd "$PROJECT_DIR/terraform"

# Review what will be destroyed
terraform plan -destroy -var-file=terraform.tfvars

# Destroy all infrastructure
terraform destroy -var-file=terraform.tfvars
yes
```

```
# Verify destruction
terraform show

# List remaining EC2 instances (should show none with EC2-RagFlow tag)
aws ec2 describe-instances \
  --filters "Name=tag:Name,Values=EC2-RagFlow" \
  --query 'Reservations[*].Instances[*].[InstanceId, State.Name]' \
  --output text
```

Result: All AWS infrastructure resources are destroyed, including EC2 instance, security groups, and key pairs.

8.4 Clean Up Local SSH Configuration

Remove SSH configurations and keys associated with the destroyed instance.

```
# Listing: Removing local SSH configuration
# Cleaning up SSH keys and config entries

# Remove SSH config entry for EC2-RagFlow
# Manual edit recommended to preserve other configurations
echo "Removing SSH config entry for EC2-RagFlow..."
perl -i.bak -ne 'print unless /^Host EC2-RagFlow$/../(^Host / && !/^Host EC2-RagFlow$/)'
↪  ~/.ssh/config

# Verify removal
grep -A 5 "EC2-RagFlow" ~/.ssh/config || echo "SSH config entry removed successfully"

# Optional: Remove SSH keys (only if no longer needed)
# Uncomment the following lines if you want to delete the keys:
# rm -f ~/.ssh/ragflow_key
# rm -f ~/.ssh/ragflow_key.pub

# List remaining SSH keys
ls -la ~/.ssh/ragflow_key* 2>/dev/null || echo "SSH keys still present (not removed)"
```

Result: SSH configuration entries are removed. SSH keys remain available for potential future deployments (remove manually if no longer needed).

8.5 Clean Up Terraform State Files

Clean up Terraform state and temporary files.

```
# Listing: Cleaning up Terraform files
# Removing state and temporary files

cd "$PROJECT_DIR/terraform"

# List Terraform state files
ls -la terraform.tfstate*

# Optional: Back up state files before removal
```

```
mkdir -p ../terraform-backups
cp terraform.tfstate* ../terraform-backups/ 2>/dev/null || echo "No state files to backup"

# Remove Terraform state files (only after infrastructure is destroyed)
rm -f terraform.tfstate
rm -f terraform.tfstate.backup
rm -f tfplan*
rm -f terraform.tfvars

# Clean up Terraform cache
rm -rf .terraform

# Verify cleanup
ls -la
```

Result: Terraform state files and temporary files are removed to complete the cleanup process.

8.6 Verify Complete Cleanup

Perform final verification that all resources have been removed.

```
# Listing: Final cleanup verification
# Confirming all resources are removed

# Check for any remaining EC2 instances
aws ec2 describe-instances \
  --filters "Name=tag:project,Values=ragflow" \
  --query
↪  'Reservations[*].Instances[*].[InstanceId, State.Name, Tags[?Key==`Name`]|[0].Value]' \
  --output table

# Check for remaining security groups
aws ec2 describe-security-groups \
  --filters "Name=group-name,Values=RagFlowSecurityGroup" \
  --query 'SecurityGroups[*].[GroupId, GroupName]' \
  --output table

# Check for remaining key pairs
aws ec2 describe-key-pairs \
  --filters "Name=key-name,Values=ragflow-key" \
  --query 'KeyPairs[*].[KeyName, KeyFingerprint]' \
  --output table

# Summary
echo "============================================="
echo "Cleanup Verification Complete"
echo "============================================="
echo "If any resources are still listed above,"
echo "they may need manual cleanup via AWS Console"
echo "============================================="
```

Result: Final verification confirms all AWS resources have been properly cleaned up and removed.

8.7 Cost Optimization Notes

Some important considerations for cost management:

- **EC2 Instance**: Charges accrue while the instance is running. Use ‘terraform destroy‘ when not actively using the system.
- **EBS Volumes**: Storage costs continue even when instances are stopped. Destroy volumes when not needed.
- **Elastic IPs**: Unused Elastic IPs incur charges. Ensure they’re released during infrastructure destruction.
- **Data Transfer**: Outbound data transfer from AWS to the internet incurs costs.

For occasional use, consider stopping the instance rather than destroying it:

```
# Stop instance without destroying (preserves data but stops compute charges)
aws ec2 stop-instances --instance-ids "$instance_id"

# Start instance when needed again
aws ec2 start-instances --instance-ids "$instance_id"
```

8.8 Summary

Infrastructure cleanup procedures remove all deployed resources. Terraform destroy commands remove AWS infrastructure, SSH configurations clean from local systems, and verification steps confirm proper resource removal. This prevents unexpected costs after project completion.

This project demonstrates that privacy-focused RAG systems are both technically feasible and practically deployable. Self-hosted solutions provide alternatives to cloud-only AI services.

—

This completes the RagFlow EC2 deployment using live-scripting methodology. Each code block is executable with F4 in Emacs using ‘send-line-to-vterm‘, or can be copied and pasted into any terminal for the same results.

9 Appendix: GitHub Self-Hosted Runner Setup

9.1 GitHub Runner for CI/CD Testing on AWS

This section establishes a GitHub Self-Hosted Runner on the AWS EC2 instance to enable continuous integration and testing workflows. The runner executes GitHub Actions directly on your infrastructure, providing complete control over the CI/CD environment.

9.1.1 Prerequisites and Security Group Configuration

Before installing the GitHub Runner, ensure the necessary ports and dependencies are available on the EC2 instance.

```
# Listing: Update AWS security group for GitHub Runner
# Adding required ports for GitHub Actions communication
exit
pwd
cd "$PROJECT_DIR/terraform"

# GitHub Runner requires outbound HTTPS (443) and SSH (22) access
```

```
# These are typically already open, but verify in your security group
# For immediate access, update via AWS CLI if needed:
aws ec2 authorize-security-group-ingress \
  --group-name RagFlowSecurityGroup \
  --protocol tcp \
  --port 443 \
  --cidr 0.0.0.0/0

  --description "GitHub Actions HTTPS"

# Verify security group rules
aws ec2 describe-security-groups \
  --filters "Name=group-name,Values=RagFlowSecurityGroup" \
  --query 'SecurityGroups[*].IpPermissions[*].[FromPort,ToPort,IpProtocol]' \
  --output table
```

Result: Security group is configured to allow GitHub Runner communication with GitHub servers.

9.1.2 GitHub Runner Installation and Configuration

Download and configure the GitHub Self-Hosted Runner on the EC2 instance.

```
# Listing: GitHub Runner installation and setup
# Installing and configuring self-hosted runner for CI/CD

ssh EC2-RagFlow

# Create runner directory
mkdir -p ~/actions-runner
cd ~/actions-runner
pwd
# Download the latest runner package
# Check https://github.com/actions/runner/releases for the latest version
curl -o actions-runner-linux-x64-2.329.0.tar.gz -L
↪  https://github.com/actions/runner/releases/download/v2.329.0/actions-runner-linux-x64-2.329.0.tar
ls -ltr
# Verify the package integrity (optional but recommended)
echo
↪  "194f1e1e4bd02f80b7e9633fc546084d8d4e19f3928a324d512ea53430102e1d  actions-runner-linux-x64-2.329
↪  | shasum -a 256 -c

# Extract the runner
tar xzf ./actions-runner-linux-x64-2.329.0.tar.gz

# List extracted files
ls -la

exit
```

Result: GitHub Runner package is downloaded and extracted on the EC2 instance.

9.1.3 Runner Registration with GitHub

Register the runner with your GitHub repository to enable CI/CD workflows.

```
# Listing: GitHub Runner registration
# Configuring runner to connect with GitHub repository

ssh EC2-RagFlow

cd ~/actions-runner

# Generate a registration token from GitHub
# Visit: https://github.com/YOUR_USERNAME/YOUR_REPO/settings/actions/runners/new
# Copy the registration token and use it below

# Configure the runner with your repository details
# Replace the token, URL, and labels with your actual values
./config.sh \
    --url https://github.com/andreaswittmann/ragflow \
    --token AFO65PTWT2JONPGSBNUANR3JBOEXO \
    --name ec2-ragflow-runner \
    --labels ragflow-test,self-hosted \
    --runnergroup Default \
    --work _workcd ~/actions-runner


# Verify configuration
./config.sh list

exit
```

Result: GitHub Runner is configured and registered with your GitHub repository.

9.1.4 Running the Runner as a Service

Install and start the GitHub Runner as a systemd service for automatic startup and management.

```
# Listing: Installing GitHub Runner as systemd service
# Setting up automatic runner startup and management

ssh EC2-RagFlow

cd ~/actions-runner

# Install the runner as a service
sudo ./svc.sh install

# Start the runner service
sudo ./svc.sh start

# Check service status
sudo systemctl --no-pager status actions.runner*

# View runner logs
```

```
sudo journalctl -u actions.runner* -f

exit
```

Result: GitHub Runner is installed as a systemd service and automatically starts on EC2 instance boot.

9.1.5 Verifying Runner Status

Confirm the runner is connected and ready to execute workflows.

```
# Listing: Verifying GitHub Runner status
# Confirming runner connectivity and readiness

ssh EC2-RagFlow

cd ~/actions-runner

# Check runner status
./config.sh list

# View service status
sudo systemctl status actions.runner*

# Check if runner is listening for jobs
ps aux | grep -i "actions-runner" | grep -v grep

exit

# Verify in GitHub web interface:
# 1. Navigate to: https://github.com/YOUR_USERNAME/YOUR_REPO/settings/actions/runners
# 2. Your runner should appear with status "Idle" (ready for jobs)
```

Result: GitHub Runner is verified as connected and ready to execute CI/CD workflows.

9.1.6 Testing the Runner with a Sample Workflow

Create a simple GitHub Actions workflow to test the runner functionality.

```
# Listing: Creating test workflow for GitHub Runner
# Setting up a sample CI/CD workflow

# On your local machine, in your repository:
mkdir -p .github/workflows

# Create a simple test workflow
cat > .github/workflows/test-runner.yml << 'EOF'
name: Test Self-Hosted Runner

on:
  push:
    branches: [ main, develop ]
```

```
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: [self-hosted, ec2, ragflow-ci]
    steps:
      - uses: actions/checkout@v3

      - name: Display system information
        run: |
          echo "Runner: ${{ runner.name }}"
          echo "OS: $(uname -s)"
          uname -a

      - name: Check Docker
        run: |
          docker --version
          docker ps

      - name: Run basic tests
        run: |
          echo "Running tests on self-hosted runner"
          echo "Current directory: $(pwd)"
          ls -la
EOF

# Commit and push the workflow
git add .github/workflows/test-runner.yml
git commit -m "Add test workflow for self-hosted runner"
git push origin main
```

Result: Sample workflow is created and pushed to trigger runner execution.

9.1.7 Monitoring and Troubleshooting

Monitor runner activity and resolve common issues.

```
# Listing: Monitoring and troubleshooting GitHub Runner
# Checking logs and resolving common problems

ssh EC2-RagFlow

# View runner service logs
sudo journalctl -u actions.runner* -n 50 --no-pager

# Check runner process
ps aux | grep -i "actions-runner"

# View runner configuration
cat ~/actions-runner/.runner

# Check disk space (runners need adequate space for job artifacts)
df -h
```

```
# Check memory usage
free -h

# If runner is stuck, restart the service
sudo systemctl restart actions.runner*

# View real-time logs
sudo journalctl -u actions.runner* -f
q

exit
```

Result: Runner logs are monitored and common issues are diagnosed.

9.1.8 Removing the Runner (if needed)

Deregister and remove the runner from your GitHub repository.

```
# Listing: Removing GitHub Runner
# Deregistering runner from GitHub and cleaning up

ssh EC2-RagFlow

cd ~/actions-runner

# Stop the runner service
sudo ./svc.sh stop

# Uninstall the service
sudo ./svc.sh uninstall

# Remove the runner from GitHub (requires token)
./config.sh remove --token YOUR_REMOVAL_TOKEN_HERE

# Clean up the runner directory
cd ~
rm -rf ~/actions-runner

# Verify removal
ls ~/actions-runner 2>/dev/null || echo "Runner directory successfully removed"

exit
```

Result: GitHub Runner is cleanly removed from the EC2 instance and deregistered from GitHub.

9.2 Summary

GitHub Self-Hosted Runner installation enables continuous integration and testing on AWS infrastructure. Docker Engine provides containerization for workflow jobs, and the runner executes GitHub Actions with full control over the environment. This integration supports automated testing, building, and deployment of RagFlow without GitHub-hosted runners.

10 Appendix: Remote Desktop Access with RustDesk

10.1 RustDesk Installation and Configuration

RustDesk provides secure remote desktop access to the EC2 instance without requiring complex port forwarding or VPN setup. This section installs and configures RustDesk for remote management and monitoring.

10.1.1 Prerequisites and Security Group Updates

Before installing RustDesk, ensure the necessary ports are open in the AWS security group. RustDesk requires TCP ports 21116 (control) and 21117 (data transfer).

```
# Listing: Update AWS security group for RustDesk
# Adding required ports for remote desktop access

cd "$PROJECT_DIR/terraform"

# Add RustDesk ports to security group (requires Terraform update)
# Note: This would need to be added to main.tf security group rules:
# ingress {
#   from_port   = 21116
#   to_port     = 21116
#   protocol    = "tcp"
#   cidr_blocks = [var.allowed_ip]
#   description = "RustDesk control port"
# }
# ingress {
#   from_port   = 21117
#   to_port     = 21117
#   protocol    = "tcp"
#   cidr_blocks = [var.allowed_ip]
#   description = "RustDesk data transfer port"
# }
# ingress {
#   from_port   = 21117
#   to_port     = 21117
#   protocol    = "udp"
#   cidr_blocks = [var.allowed_ip]
#   description = "RustDesk data transfer port (UDP)"
# }

# For immediate access, update via AWS CLI:
aws ec2 authorize-security-group-ingress \
  --group-name RagFlowSecurityGroup \
  --protocol tcp \
  --port 21116 \
  --cidr 0.0.0.0/0 \
  --description "RustDesk control"

aws ec2 authorize-security-group-ingress \
  --group-name RagFlowSecurityGroup \
  --protocol tcp \
  --port 21117 \
```

```
  --cidr 0.0.0.0/0 \
  --description "RustDesk data transfer TCP"

aws ec2 authorize-security-group-ingress \
  --group-name RagFlowSecurityGroup \
  --protocol udp \
  --port 21117 \
  --cidr 0.0.0.0/0 \
  --description "RustDesk data transfer UDP"
```

Result: Security group is updated to allow RustDesk remote desktop connections.

10.1.2 RustDesk Installation on EC2 Instance

This installs RustDesk server components and configures the service for remote access.

```
# Listing: RustDesk installation on EC2
# Installing remote desktop server for management access

ssh EC2-RagFlow

#
echo "ubuntu:ham2burg" | sudo chpasswd

# Update package list and install dependencies
sudo apt update
sudo apt install -y curl wget apt-transport-https

# Download and install RustDesk (latest stable version)
# Check latest version at: https://github.com/rustdesk/rustdesk/releases
wget https://github.com/rustdesk/rustdesk/releases/download/1.2.3/rustdesk-1.2.3-x86_64.deb
sudo dpkg -i rustdesk-1.2.3-x86_64.deb

# Fix any missing dependencies
sudo apt install -f

# Enable and start the RustDesk service
sudo systemctl enable rustdesk
sudo systemctl start rustdesk

# Verify service is running
sudo systemctl status rustdesk

# Check RustDesk configuration
cat ~/.config/rustdesk/RustDesk.toml

exit
```

Result: RustDesk is installed and running on the EC2 instance, ready for remote connections.

10.1.3 RustDesk Client Setup on Local Machine

Configure RustDesk on your MacBook Pro for connecting to the remote instance.

```
# Listing: Local RustDesk client setup
# Preparing local machine for remote desktop connection

# Download and install RustDesk from https://rustdesk.com/
# For macOS:
curl -L https://github.com/rustdesk/rustdesk/releases/download/1.2.3/rustdesk-1.2.3.dmg -o
↪  rustdesk.dmg
hdiutil attach rustdesk.dmg
cp -r /Volumes/RustDesk/RustDesk.app /Applications/
hdiutil detach /Volumes/RustDesk

# Launch RustDesk
open /Applications/RustDesk.app
```

Result: RustDesk client is installed and ready for connection to the remote EC2 instance.

10.1.4 Establishing Remote Desktop Connection

Connect to the EC2 instance using RustDesk for graphical remote access.

```
# Listing: Connecting to EC2 via RustDesk
# Establishing remote desktop session

# On the EC2 instance, get the RustDesk ID:
ssh EC2-RagFlow 'sudo rustdesk --get-id'

# Set a password for security (optional but recommended):
ssh EC2-RagFlow 'sudo rustdesk --password'

# In RustDesk client on Mac:
# 1. Enter the ID shown above (e.g., 55689929)
# 2. Enter the password (if set)
# 3. Enter the public IP: 63.180.133.202
# 4. Click Connect

# For direct connection (bypass relay servers):
ssh EC2-RagFlow
↪  'echo "[options]\ndirect-server = \"true\"" >> ~/.config/rustdesk/RustDesk.toml'

# Note: If you get "Unsupported display server type" error:
# 1. Install a desktop environment: sudo apt install -y xfce4 xfce4-goodies
# 2. Start the display manager: sudo systemctl start lightdm
# 3. Add user to nopasswdlogin group: sudo usermod -aG nopasswdlogin ubuntu
# 4. Set default target to graphical: sudo systemctl set-default graphical.target
# 5. Restart lightdm: sudo systemctl restart lightdm
# 6. Set user password if needed: echo "ubuntu:yourpassword" | sudo chpasswd
# 7. Configure XFCE session: sudo mkdir -p /etc/lightdm/lightdm.conf.d && sudo bash -c "echo
↪  -e \"[Seat:*]\nuser-session=xfce\n\" > /etc/lightdm/lightdm.conf.d/50-xfce-session.conf"
# 8. Configure German keyboard: sudo dpkg-reconfigure keyboard-configuration (select German)
↪  or sudo sed -i "s/XKBLAYOUT=.*/XKBLAYOUT=\"de\"/g" /etc/default/keyboard
# 9. Increase resolution: Create /etc/X11/xorg.conf.d/10-monitor.conf with 1920x1080
↪  configuration and restart lightdm
# 10. Alternative VNC access: Install x11vnc for higher performance VNC connections on port
↪  5900
```

Result: Remote desktop connection is established, providing full graphical access to the EC2 instance.

10.2 Summary

RustDesk installation enables secure remote desktop access to the EC2 instance. The configuration provides graphical management capabilities with security through password protection and direct connection options. This complements SSH access with visual interface capabilities for system administration and monitoring.

11 Appendix: Remote Desktop Access with NoMachine

11.1 NoMachine Installation and Configuration

NoMachine provides high-performance remote desktop access to the EC2 instance with low latency and excellent user experience. This section installs and configures NoMachine server for secure graphical remote management.

11.1.1 Prerequisites and Security Group Updates

Before installing NoMachine, ensure the necessary ports are open in the AWS security group. NoMachine requires TCP port 4000 for the main connection and UDP ports 4011-4999 for NX technology data transfer.

```
# Listing: Update AWS security group for NoMachine
# Adding required ports for remote desktop access

cd "$PROJECT_DIR/terraform"

# Add NoMachine ports to security group (requires Terraform update)
# Note: This would need to be added to main.tf security group rules:
# ingress {
#   from_port   = 4000
#   to_port     = 4000
#   protocol    = "tcp"
#   cidr_blocks = [var.allowed_ip]
#   description = "NoMachine main connection port"
# }
# ingress {
#   from_port   = 4011
#   to_port     = 4999
#   protocol    = "udp"
#   cidr_blocks = [var.allowed_ip]
#   description = "NoMachine NX data transfer ports"
# }

# For immediate access, update via AWS CLI:
aws ec2 authorize-security-group-ingress \
  --group-name RagFlowSecurityGroup \
  --protocol tcp \
  --port 4000 \
  --cidr 0.0.0.0/0 \
```

```
  --description "NoMachine main connection"

aws ec2 authorize-security-group-ingress \
  --group-name RagFlowSecurityGroup \
  --protocol udp \
  --port 4011-4999 \
  --cidr 0.0.0.0/0 \
  --description "NoMachine NX data transfer"
```

Result: Security group is updated to allow NoMachine remote desktop connections.

11.1.2 Desktop Environment Installation

Install a lightweight desktop environment if the instance is headless (no graphical interface).

```
# Listing: Desktop environment installation for NoMachine
# Installing XFCE desktop environment for graphical access

ssh EC2-RagFlow

# Update package list
sudo apt update && sudo apt upgrade -y

# Install XFCE desktop environment
sudo apt install -y xfce4 xfce4-goodies

# Exit SSH session
exit
```

Result: XFCE desktop environment is installed, providing a graphical interface for NoMachine connections.

11.1.3 NoMachine Server Installation

Download and install the NoMachine server package on the EC2 instance.

```
# Listing: NoMachine server installation
# Installing NoMachine server for remote desktop access

ssh EC2-RagFlow

# Download NoMachine server package
wget https://download.nomachine.com/download/8.14/Linux/nomachine_8.14.2_1_amd64.deb

# Install the package
sudo apt install -y ./nomachine_8.14.2_1_amd64.deb

# The service starts automatically after installation
# Verify service status
sudo systemctl status nxserver
```

```
# Exit SSH session
exit
```

Result: NoMachine server is installed and running, ready to accept remote desktop connections.

11.1.4 Firewall Configuration on Server

Configure UFW firewall to allow NoMachine connections if UFW is active.

```
# Listing: Firewall configuration for NoMachine
# Allowing NoMachine ports through UFW firewall

ssh EC2-RagFlow

# Allow NoMachine main connection port
sudo ufw allow from 0.0.0.0/0 to any port 4000 proto tcp

# Allow NoMachine NX data transfer ports
sudo ufw allow from 0.0.0.0/0 to any port 4011:4999 proto udp

# Reload firewall rules
sudo ufw reload

# Verify firewall status
sudo ufw status

# Exit SSH session

### if system does not work correctly
ssh EC2-RagFlow 'sudo /etc/NX/nxserver --restart'
exit
```

Result: Firewall is configured to allow NoMachine connections on the required ports.

11.1.5 Testing NoMachine Connection

Connect to the EC2 instance using NoMachine client from your local machine.

```
# Listing: Testing NoMachine connection
# Establishing remote desktop connection

# In NoMachine client:
# 1. Create new connection
# 2. Host: <EC2_PUBLIC_IP>
# 3. Port: 4000
# 4. Protocol: NX
# 5. Authentication: Password (use ubuntu username and instance password)
# 6. Connect

# Display connection details
echo "Connect to NoMachine at: $public_ip:4000"
echo "Username: ubuntu"
echo "Use the SSH key or instance password for authentication"
```

Result: NoMachine connection is established, providing full graphical remote desktop access to the EC2 instance.

11.1.6 Troubleshooting Common Issues

Address common NoMachine connection problems.

```
# Listing: Troubleshooting NoMachine issues
# Resolving common connection problems

ssh EC2-RagFlow

# If connection times out:
# Restart NoMachine service
sudo /etc/NX/nxserver --restart

# If black screen appears:
# Switch from Wayland to X11
sudo nano /etc/gdm3/custom.conf
# Add or uncomment: WaylandEnable=false
# Save and exit, then reboot
sudo reboot

# If connection hangs:
# In client, press Ctrl+Alt+0 to disable hardware decoding

# Exit SSH session
exit
```

Result: Common NoMachine issues are resolved, ensuring stable remote desktop connections.

11.1.7 High Resolution Configuration with XDummy Driver

For optimal NoMachine performance with higher resolutions, configure the XDummy driver to support resolutions up to 1920x1080 and beyond. This creates a virtual display that NoMachine can connect to with full resolution control.

```
# Listing: XDummy driver installation for high resolution
# Installing virtual display driver for NoMachine resolution support

ssh EC2-RagFlow

# Install the XDummy driver
sudo apt install -y xserver-xorg-video-dummy

# Exit SSH session
exit
```

Result: XDummy driver is installed, providing virtual display capabilities for higher resolutions.

```
# Listing: Xorg configuration for high resolution display
# Creating Xorg configuration with multiple resolution support

ssh EC2-RagFlow

# Remove any existing monitor configuration
sudo rm -f /etc/X11/xorg.conf.d/10-monitor.conf

# Create comprehensive Xorg configuration
sudo tee /etc/X11/xorg.conf > /dev/null << 'EOF'
Section "Device"
    Identifier "DummyDevice"
    Driver "dummy"
    Option "UseEDID" "false"
    VideoRam 512000
EndSection

Section "Monitor"
    Identifier "DummyMonitor"
    HorizSync 5.0 - 1000.0
    VertRefresh 5.0 - 200.0
    Option "ReducedBlanking"
    ModeLine "1920x1080" 148.5 1920 2008 2052 2200 1080 1084 1089 1125 +hsync +vsync
    ModeLine "1680x1050" 119.0 1680 1728 1760 1840 1050 1053 1059 1080 +hsync -vsync
    ModeLine "1600x900" 97.5 1600 1648 1680 1760 900 903 908 926 +hsync -vsync
    ModeLine "1440x900" 88.8 1440 1488 1520 1600 900 903 909 926 +hsync -vsync
    ModeLine "1366x768" 72.3 1366 1414 1446 1494 768 771 777 803 +hsync +vsync
    ModeLine "1280x1024" 109.0 1280 1368 1496 1712 1024 1027 1034 1063 -hsync +vsync
    ModeLine "1280x800" 71.0 1280 1328 1360 1440 800 803 809 823 +hsync -vsync
    ModeLine "1024x768" 65.0 1024 1048 1184 1344 768 771 777 806 -hsync -vsync
EndSection

Section "Screen"
    Identifier "DummyScreen"
    Device "DummyDevice"
    Monitor "DummyMonitor"
    DefaultDepth 24
    SubSection "Display"
        Viewport 0 0
        Depth 24
        Modes "1920x1080" "1680x1050" "1600x900" "1440x900" "1366x768" "1280x1024" "1280x800" "1024x7
        Virtual 1920 1080
    EndSubSection
EndSection
EOF

# Exit SSH session
exit
```

Result: Xorg configuration is created with comprehensive resolution support including 1920x1080.

```
# Listing: Restarting display services for resolution changes
# Applying the new display configuration
```

```
ssh EC2-RagFlow

# Restart LightDM to apply Xorg changes
sudo systemctl restart lightdm

# Wait for services to start
sleep 10

# Restart NoMachine server
sudo /etc/NX/nxserver --restart

# Exit SSH session
exit
```

Result: Display services are restarted with the new high-resolution configuration applied.

```
# Listing: Verifying high resolution configuration
# Checking that the display supports higher resolutions

ssh EC2-RagFlow

# Check Xorg log for dummy driver confirmation
sudo grep -i "dummy\|1920x1080\|modes" /var/log/Xorg.0.log | tail -10

# Exit SSH session
exit

# Test connection with higher resolution
echo "Reconnect with NoMachine - you should now see resolutions up to 1920x1080 available"
echo "In NoMachine client: Connection Settings → Display → Select desired resolution"
```

Result: High resolution configuration is verified and ready for NoMachine connections with full resolution support.

11.1.8 German Keyboard Configuration Fix

For users with German keyboards, the NoMachine GUI may default to English keyboard layout even when SSH connections work correctly. This occurs because XFCE desktop settings use an incorrect keyboard model that prevents proper key mapping.

Problem Diagnosis The issue manifests as:

- SSH terminal connections recognize German keyboard correctly
- NoMachine GUI session only offers English keyboard layouts
- Keys produce wrong characters (e.g., 'z' and 'y' swapped, special characters missing)

Root cause: XFCE keyboard model set to "genius" instead of "pc105" (standard PC keyboard).

```
# Listing: Diagnosing keyboard configuration issue
# Checking current keyboard settings in NoMachine session

ssh EC2-RagFlow
```

```
# Check system-level keyboard configuration
localectl status

# Check XFCE keyboard configuration
cat ~/.config/xfce4/xfconf/xfce-perchannel-xml/keyboard-layout.xml

# Display current keyboard configuration in X11
# Note: This command only works from within an active X session
# Run this in a terminal inside NoMachine: setxkbmap -query

exit
```

Result: Diagnosis reveals incorrect keyboard model ("genius" instead of "pc105") in XFCE configuration.

Solution: Update XFCE Keyboard Configuration This corrects the keyboard model to enable proper German keyboard mapping.

```
# Listing: Fixing XFCE keyboard model configuration
# Updating keyboard model from "genius" to "pc105"

ssh EC2-RagFlow

# Backup existing configuration
cp ~/.config/xfce4/xfconf/xfce-perchannel-xml/keyboard-layout.xml \
   ~/.config/xfce4/xfconf/xfce-perchannel-xml/keyboard-layout.xml.backup

# Update XFCE keyboard configuration with correct model
cat > ~/.config/xfce4/xfconf/xfce-perchannel-xml/keyboard-layout.xml << 'EOF'
<?xml version="1.0" encoding="UTF-8"?>

<channel name="keyboard-layout" version="1.0">
  <property name="Default" type="empty">
    <property name="XkbDisable" type="bool" value="false"/>
    <property name="XkbLayout" type="string" value="de,us"/>
    <property name="XkbVariant" type="string" value=","/>
    <property name="XkbModel" type="string" value="pc105"/>
  </property>
</channel>
EOF

# Verify the configuration was updated
cat ~/.config/xfce4/xfconf/xfce-perchannel-xml/keyboard-layout.xml

exit
```

Result: XFCE keyboard configuration is updated with the correct pc105 model supporting both German and US layouts.

Solution: Add Session Startup Script This ensures keyboard settings are applied automatically on each login.

```
# Listing: Creating session startup script for keyboard
# Ensuring keyboard layout persists across sessions

ssh EC2-RagFlow

# Create .xprofile for automatic keyboard configuration
echo "#!/bin/bash" > ~/.xprofile
echo "# Set keyboard layout for X session" >> ~/.xprofile
echo "setxkbmap -model pc105 -layout de,us -variant , -option grp:alt_shift_toggle" >>
↪  ~/.xprofile

# Make the script executable
chmod +x ~/.xprofile

# Verify the script was created
cat ~/.xprofile

exit
```

Result: Session startup script is created to automatically apply German keyboard layout with Alt+Shift toggle between German and US layouts.

Applying the Fix Log out of the NoMachine session and reconnect to apply the changes.

```
# Listing: Verifying keyboard fix
# Testing the corrected configuration

# After reconnecting to NoMachine, open a terminal and verify:
# setxkbmap -query
# Expected output should show:
# model: pc105 (not "genius")
# layout: de,us

# Test German keyboard:
# - Try typing German special characters: ä ö ü ß
# - Use Alt+Shift to toggle between de and us layouts
# - Verify 'z' and 'y' keys produce correct characters

echo "German keyboard should now work correctly in NoMachine GUI"
echo "Use Alt+Shift to toggle between German (de) and US (us) keyboard layouts"
```

Result: German keyboard functions correctly in NoMachine GUI with Alt+Shift toggle for layout switching.

11.2 Summary

NoMachine installation provides high-performance remote desktop access to the EC2 instance. The configuration includes security group updates, desktop environment setup, and firewall configuration. This complements SSH and RustDesk access with optimized graphical remote management capabilities.

12 End