Private RAG AWS GPU Deployment - Live-Scripting Workflow

Andreas Wittmann

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

2025-11-18

Contents

1 Summary

This document outlines the deployment of RagFlow, an open-source RAG platform, on Ubuntu 24.04 with GPU acceleration. Using the live-scripting methodology, it contains executable code blocks that ensure reproducible infrastructure setup. It covers Terraform infrastructure provisioning with pre-configured Deep Learning AMI, Docker containerization, and RagFlow deployment with GPU support.

- **HTML Version:** private-rag-aws-gpu.html
- **PDF Version:** private-rag-aws-gpu.pdf
- **GitHub:** `https://github.com/andreaswittmann/private-rag/tree/main`

2 Building Privacy-Focused RAG Platform with GPU Acceleration

2.1 Project Overview

This project uses the live-scripting methodology to create an example configuration of the open-source project RagFlow on Ubuntu 24.04 with GPU support. 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 with GPU-enabled EC2 instance running Ubuntu 24.04 is established as a platform for RagFlow installation. The Deep Learning AMI comes pre-configured with NVIDIA drivers and CUDA toolkit, eliminating manual GPU setup. The example demonstrates all necessary configuration and software tool installations required for the project. To test the installation a sample use case shows the how an document is ingested and queried using RagFlow UI. A section in the end outlines the steps for removing the infrastructure completely.

2.2 Live-Scripting Methodology

The deployment uses live-scripting methodology to transform static documentation into executable workflows. Code blocks can be executed step by step directly in Emacs (F4 key) or be copied 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 Orgmode File to HTML and PDF
- Reproducible deployments with consistent infrastructure setup
- Support of version control through plain-text format

2.2.1 Implementation Options

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

```
# Listing: Emacs Lisp function for sending current line or region to vterm
;; Put this to your  ~/.emacs.d/personal/my_custom.el or similar file and load it.

(defun my-select-current-line ()
  "Selects the current line, including the NEXT-LINE char at the end"
  (interactive)
  (move-beginning-of-line nil)
  (set-mark-command nil)
  (move-end-of-line 2)
  (move-beginning-of-line nil)
  (setq deactivate-mark nil))

(defun send-line-to-vterm ()
  "Send region if active, or current line to vterm buffer. Then move to next line."
  (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))
```

Non-Emacs Users Copy code blocks to any terminal,

2.3 Expected Deployment Outcome

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

3 Foundation & Prerequisites

3.1 Environment Setup

These steps establish the development environment and validate prerequisites.

3.1.1 Bootstrapping with Cloning private-rag Repository

This creates a Lab directory and clones the private-rag repository for subsequent live-scripting consumption with Emacs.

```
# Listing: Cloning private-rag repository

export BASE_DIR="~/$(whoami)/Labs"
mkdir -p $BASE_DIR
cd $BASE_DIR

git clone https://github.com/andreaswittmann/private-rag.git

# open in Emacs: ~/$(whoami)/Labs/private-rag/private-rag-aws-gpu.org
```

To fowllow along usnig the live-scripting methodology, open the file ~/$(whoami)/Labs/private-rag/private-rag-aws-gpu.org in Emacs.

3.1.2 Setting the Project Foundation

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

```
# Listing: Project environment setup

bash
PROJECT_DIR="~/$(whoami)/Labs/private-rag"
cd "$PROJECT_DIR"
pwd
ls -la

# Set AWS profile for this session
export AWS_PROFILE=lab-a-north
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

# 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
```

Result: Project environment is configured and AWS default VPC, AWS credentials and required tools are validated for infrastructure deployment.

3.2 Summary

The foundation phase establishes the development environment by cloning the private-rag repository and setting the project foundation. AWS credentials are verified, default VPC created if needed, S3 access tested, STS identity confirmed, SSH keys validated, and required tools like Terraform checked. This groundwork enables reproducible infrastructure deployment through live-scripting methodology.

4 Infrastructure as Code

4.1 SSH Key Generation for Secure Access

This block 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"
pwd;
# Generate a new SSH key pair specifically for RagFlow GPU
ssh-keygen -t rsa -b 4096 -f ~/.ssh/ragflow-gpu_key -N ""
y
# Display the public key for use in Terraform variables
ls -la ~/.ssh/ragflow-gpu_key*
cat ~/.ssh/ragflow-gpu_key.pub

# Store public key in environment variable
export TF_VAR_SSH_PUBLIC_KEY="$(cat ~/.ssh/ragflow-gpu_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 for GPU-enabled instance.

```
# Listing: Terraform environment preparation
# Setting up infrastructure configuration for GPU instance

cd "$PROJECT_DIR/terraform-gpu"
pwd
# Create terraform variables file with GPU configuration
```

```
cat > terraform.tfvars << EOF
# Generated tfvars file - $(date)
aws_region       = "eu-north-1"
instance_type    = "g4dn.xlarge"
root_volume_size = 100
root_volume_type = "gp3"
allowed_ip       = "0.0.0.0/0"
ssh_public_key   = "${TF_VAR_SSH_PUBLIC_KEY}"
ec2_name         = "EC2-RagFlow-GPU"
environment      = "development"
project          = "ragflow-gpu"
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 GPU-enabled infrastructure 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 GPU infrastructure 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 with GPU support.

```
# Listing: AWS GPU infrastructure deployment
# Creating the GPU-enabled 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-GPU"
↪  "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 GPU-enabled 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 GPU instance.

```
# Listing: SSH configuration setup for GPU instance

pwd
# Test initial SSH connection
ssh -i ~/.ssh/ragflow-gpu_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 GPU Project
Host EC2-RagFlow-GPU
    HostName $public_ip
    User ubuntu
    IdentityFile ~/.ssh/ragflow-gpu_key
EOF
## use a perl one-liner to grep these lines from the ssh-config file.
perl -nE 'print if /Host EC2-RagFlow-GPU/ .. /yesEOF/' ~/.ssh/config
#bbedit ~/.ssh/config

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

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

4.6 GPU Instance Selection and Upgrading

4.6.1 Recommended GPU EC2 Instance Types for RAG Workloads

Here is a list of recommended GPU-enabled EC2 instance types suitable for RAG workloads with RagFlow, focusing on instances with NVIDIA GPUs optimized for AI/ML tasks. All recommendations are for the `eu-north-1` region, with approximate on-demand pricing (as of late 2024)

Instance Type	GPU	VRAM (GB)	vCPUs	RAM (GB)	Network Speed	Approx. Price/Hour (US
g4dn.xlarge	1x T4	16	4	16	Up to 25 Gbps	$0.71
g4dn.2xlarge	1x T4	16	8	32	Up to 25 Gbps	$1.05
g5.xlarge	1x A10G	24	4	16	Up to 10 Gbps	$1.21
g6e.xlarge	1x L40S	48	4	32	Up to 20 Gbps	$1.22

Recommendations

- **For most RAG setups**: Start with `g4dn.xlarge` for a balance of GPU performance, memory, and cost.
- **For high-performance needs**: Opt for `g5.xlarge` or `g6e.xlarge` if you need more VRAM for larger models.
- **Considerations**: GPU instances are more expensive but provide significant performance improvements for embeddings and inference tasks.

4.6.2 Upgrading to a Different GPU Instance Type with Terraform

The best way to change your EC2 instance type is to update your Terraform configuration and apply the changes.

```
# Listing: Changing GPU instance type with Terraform

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

# Update the instance type in terraform.tfvars (replace 'g5.xlarge' with your desired type)
# or use an editor to change the instance type directly.
sed -i 's/instance_type    = "g4dn.xlarge"/instance_type    = "g5.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-GPU" \
  --query 'Reservations[*].Instances[*].[InstanceType, State.Name]' \
  --output text
```

Result: GPU instance type has been successfully changed to the new configuration with updated GPU capabilities.

4.7 Summary

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

5 GPU Verification

This verifies that the pre-installed GPU drivers, CUDA, and Docker GPU support are working correctly.

```
# Listing: GPU verification
ssh EC2-RagFlow-GPU

# Installing NVIDIA monitoring tool nvitop
sudo apt update && sudo apt install -y pipx
pipx install nvitop

nvitop --version

# Check NVIDIA driver and GPU status
nvidia-smi

# Check CUDA version
nvcc --version

# Test GPU in Docker
docker run --rm --gpus all nvidia/cuda:12.9.0-base-ubuntu24.04 nvidia-smi

exit
```

Result: Pre-installed GPU drivers, CUDA toolkit, and Docker GPU support are verified as fully operational.

6 Development Environment Setup

6.1 Complete Docker Engine and Tools Installation

This establishes the complete development environment on the Ubuntu 24.04 GPU instance, including Docker Engine, container management tools, GitHub CLI, and Python development utilities.

6.1.1 Docker Engine Installation and Tools

Install Docker Engine with proper repository configuration and user permissions.

```
# Listing: Docker Engine installation on GPU instance
# Setting up containerization platform for GPU-accelerated applications

ssh EC2-RagFlow-GPU

# Test Docker installation
docker run hello-world

# Verify Docker Compose
docker compose version

# Enable and Check Docker service to start on boot
sudo systemctl enable docker
sudo systemctl start docker
sudo systemctl status docker  --no-pager

# Install lazydocker for terminal-based Docker management
curl
↪  https://raw.githubusercontent.com/jesseduffield/lazydocker/master/scripts/install_update_linux.sh
↪  | bash
echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.bashrc
source ~/.bashrc

lazydocker --version

# Install htop for system monitoring
sudo apt install -y htop
htop --version


exit
```

Result: Docker insallation is varified. Docker service is configured to start automatically on EC2 instance boot. Lazydocker is installed and available in the PATH for terminal-based Docker management. The tool htop is also installed.

6.1.2 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-GPU

# 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 update
sudo apt 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.

6.1.3 Development Environment Summary

Verify the complete development environment is ready for deployment. [Editors Note: Remove the echo statements in the this code block.]

```
# Listing: Development environment verification

ssh EC2-RagFlow-GPU

# Display installed versions
docker --version
docker compose version
gh --version
uv --version
gcc --version | head -1
/usr/local/cuda/bin/nvcc --version | head -1
nvidia-smi --query-gpu=name --format=csv,noheader
uname -a
df -h /
nvidia-smi --query-gpu=memory.total --format=csv,noheader

exit
```

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

6.2 Summary

The development environment setup establishes necessary tools for containerized application deployment and CI/CD workflows on GPU-enabled infrastructure. Docker Engine provides the containerization platform, Lazydocker offers terminal management, and Python development tools support application requirements. CUDA and NVIDIA drivers enable GPU acceleration for enhanced performance.

7 Core RagFlow Deployment

7.1 Basic Ragflow Setup

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

```
# Listing: RagFlow deployment on GPU instance
# Installing and starting RagFlow RAG platform with GPU support

# SSH into the GPU EC2 instance
ssh EC2-RagFlow-GPU

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

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

# Update .env file to use GPU profile
sed -i 's/DEVICE=cpu/DEVICE=gpu/' .env
cat .env | grep DEVICE

# Upate docker-compose.yml to uncomment the executor service.
perl -i -pe 's/^  # /  / if /^  # /' docker/docker-compose.yml


# 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

exit
```

Result: RagFlow is successfully deployed and running on the GPU-enabled EC2 instance.

7.2 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 on the GPU instance.

Prerequisites

- Docker and Docker Compose installed on your GPU 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

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

Generate a self-signed certificate for your server's IP address.

```
bash
pwd
ssh EC2-RagFlow-GPU
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 localhost or 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.

7.2.2 Step 2: Modify Nginx Configuration**

Update the existing ragflow.https.conf file to use your IP address.

```
# 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
# For GPU version, use ragflow-gpu service
sed -i 's|http://ragflow:|http://ragflow-gpu:|' docker/nginx/ragflow.https.ip.conf

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

7.2.3 Step 3: Update Docker Compose Configuration

Modify the docker-compose.yml file to mount the self-signed certificates.

```
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.conf


# 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
```

7.2.4 Step 4: Restart RAGFlow Services

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

```
ssh EC2-RagFlow-GPU

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

# Set the profile in .env and start (CPU or GPU)
docker compose  up -d

# Check that services are running
docker compose ps
docker compose exec ragflow-gpu date; pwd; ls -la
```

7.2.5 Step 5: Test HTTPS Access

Verify that the HTTPS setup is working correctly.

```
# Listing: Testing HTTPS access to RagFlow

# 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

# Test GPU functionality in running containers
docker compose exec ragflow-gpu nvidia-smi --query-gpu=name,memory.used,memory.total
↪  --format=csv
```

Ragflow URL is accessible at:

- http://13.62.214.0/
- https://13.62.214.0/

7.3 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-GPU
# 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.

7.4 Summary

RagFlow deployment establishes the core RAG platform with full GPU acceleration. The application uses GPU profile with optimized Docker Compose configuration, GPU resource limits, and CUDA environment variables for maximum performance. HTTPS setup with self-signed certificates and SSH tunnel access provide secure local browser connectivity. The GPU-enabled infrastructure delivers significant performance improvements for embeddings generation, vector similarity searches, and document processing operations.

8 Advanced Features - Document Processing

8.1 Document Ingestion and Processing

This demonstrates uploading and processing documents for RAG functionality with GPU acceleration.

8.1.1 Upload Documents

Access RagFlow via browser and upload sample documents for processing.

8.1.2 Configure Knowledge Base

Create a knowledge base and configure document parsing settings.

8.1.3 Test RAG Queries

Test retrieval-augmented generation with uploaded documents, leveraging GPU acceleration for faster processing.

8.2 Summary

Document processing enables advanced knowledge management while preserving privacy. GPU acceleration provides enhanced performance for embeddings generation and vector similarity searches. Local processing handles sensitive documents without transmitting data to external services.

9 Infrastructure Cleanup

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 GPU resources.

9.1 Stop RagFlow Services (Optional)

Before destroying infrastructure, gracefully stop all running services.

```
# Listing: Stopping RagFlow services

ssh EC2-RagFlow-GPU

# Stop RagFlow Docker containers
cd ~/ragflow
docker compose --profile gpu 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 GPU EC2 instance.

9.2 Terminate GPU EC2 Instance with Terraform

Use Terraform to cleanly destroy all AWS infrastructure resources including the GPU instance.

```
# Listing: Destroying AWS GPU infrastructure

cd "$PROJECT_DIR/terraform-gpu"
pwd
# Review what will be destroyed
terraform init
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-GPU tag)
aws ec2 describe-instances \
  --filters "Name=tag:Name,Values=EC2-RagFlow-GPU" \
  --query 'Reservations[*].Instances[*].[InstanceId, State.Name]' \
  --output text
```

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

9.3 Clean Up Local SSH Configuration

Remove SSH configurations and keys associated with the destroyed GPU instance.

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

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

# Verify removal
grep -A 5 "EC2-RagFlow-GPU" ~/.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-gpu_key
rm -f ~/.ssh/ragflow-gpu_key.pub

# List remaining SSH keys
ls -la ~/.ssh/ragflow-gpu_key*
```

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

9.4 Clean Up Terraform State Files

Clean up Terraform state and temporary files.

```
# Listing: Cleaning up Terraform files
# Removing state and temporary files
pwd
cd "$PROJECT_DIR/terraform-gpu"

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

# Optional: Back up state files before removal
mkdir -p ../terraform-gpu-backups
cp terraform.tfstate* ../terraform-gpu-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
pwd
ls -la
```

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

9.5 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-gpu" \
  --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=RagFlowGPUSecurityGroup" \
  --query 'SecurityGroups[*].[GroupId, GroupName]' \
  --output table

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

#  "Cleanup Verification Complete"
```

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

9.6 Summary

Infrastructure cleanup procedures remove all deployed GPU 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 with optimized GPU acceleration using Ubuntu 24.04 and pre-configured Deep Learning AMIs. The GPU profile configuration, CUDA environment variables, and Docker resource limits ensure maximum performance for embeddings generation and vector operations. Self-hosted solutions provide alternatives to cloud-only AI services while delivering enterprise-grade GPU performance for enhanced RAG capabilities.

—

This completes the RagFlow AWS GPU 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.

10 Appendix: Live-Cycle Operations

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

```
exit

bash
pwd

# Initialize project environment
PROJECT_DIR="~/$(whoami)/Labs/private-rag"
cd "$PROJECT_DIR"
export AWS_PROFILE=lab-a-north

# Get instance ID and filter out terminated instances if needed
instance_id=$(aws ec2 describe-instances --filters "Name=tag:Name,Values=EC2-RagFlow-GPU"
↪  "Name=instance-state-name,Values=running,pending,stopping,stopped" --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 with timeout of 10 seconds
ssh EC2-RagFlow-GPU -v -o ConnectTimeout=10 'whoami && date && uptime'
ssh EC2-RagFlow-GPU '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: GPU EC2 instance started and ready for use. Stop commands are provided as well.

11 End