# LibreChat AWS Deployment - Live-Scripting Workflow

Andreas Wittmann

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

*2025-06-03*

## Contents

- **HTML Version:** librechat-aws-deployment.html
- **PDF Version:** librechat-aws-deployment.pdf

# 1 Building a Privacy-Focused AI Assistant

## 1.1 Project Overview and Privacy Philosophy

In an era where AI capabilities are rapidly expanding but privacy concerns remain paramount, a fundamental challenge exists: How can powerful language models be leveraged while maintaining complete control over data and interactions?

This project demonstrates a solution—a fully self-hosted LibreChat deployment with AWS Bedrock integration that achieves enterprise-grade AI capabilities without sacrificing privacy. The implementation combines privacy-preserving technology with practical AI assistance requirements.

## 1.2 The Live-Scripting Methodology

This deployment uses live-scripting methodology to transform static documentation into executable workflows. Code blocks can be executed directly in Emacs (F4 key) or copied to any terminal, creating a seamless bridge between documentation and execution.

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

- **Executable Documentation**: Code blocks remain current and functional
- **Multi-Format Publishing**: Single source generates Org, HTML, and PDF outputs
- **Reproducible Deployments**: Consistent infrastructure deployment across environments
- **Version Control Integration**: Plain-text format enables proper diff tracking

### 1.2.1 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 regardless of tools.

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

## 1.3 Project Components

- **Complete privacy control**: Self-hosted LibreChat with AWS Bedrock models (Claude, Nova, DeepSeek)
- **Production-ready foundation**: SSL/HTTPS, Docker containerization, infrastructure automation
- **Advanced AI capabilities**: RAG functionality, multi-model support, document processing
- **Reproducible methodology**: Live-scripting workflow for consistent deployment

## 1.4 Expected Deployment Outcome

This workflow produces a functioning HTTPS-enabled LibreChat instance running on AWS, integrated with multiple Bedrock models, and enhanced with RAG capabilities under complete user control.

# 2 Part 1: Foundation & Prerequisites

## 2.1 Environment Setup

The following steps establish the development environment and validate the prerequisites.

### 2.1.1 Setting the Project Foundation

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

```
# Listing: Project environment setup
# Establishing the base directory for our privacy-focused AI project

PROJECT_DIR="/Users/$(whoami)/LocalProjects/ai-bootcamp/private-ai"
cd "$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=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
```

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

### 2.1.2 Validating AWS Prerequisites

This confirms AWS environment configuration for deployment requirements.

```
# Listing: AWS environment validation
# Ensuring AWS access and Bedrock 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.

## 2.2 Part 1 Summary

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

# 3 Part 2: Infrastructure as Code

## 3.1 Infrastructure Automation Benefits

Infrastructure automation provides significant advantages over manual console interactions. The entire environment can be reproduced with code, enabling experimentation, learning, and knowledge sharing.

### 3.1.1 SSH Key Generation for Secure Access

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

```bash
# 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 LibreChat
ssh-keygen -t rsa -b 4096 -f ~/.ssh/librechat_key -N ""
y
# Display the public key for use in Terraform variables
cat ~/.ssh/librechat_key.pub

# Store public key in environment variable
export TF_VAR_SSH_PUBLIC_KEY="$(cat ~/.ssh/librechat_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.

### 3.1.2 Terraform Configuration and Planning

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

```bash
# 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 - $(date)
aws_region       = "eu-north-1"
instance_type    = "t3.medium"
root_volume_size = 100
root_volume_type = "gp3"
allowed_ip       = "0.0.0.0/0"
ssh_public_key   = "${TF_VAR_SSH_PUBLIC_KEY}"
ec2_name         = "EC2-LibreChat"
environment      = "development"
project          = "private-ai"
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.

### 3.1.3 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: length})'
```

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

### 3.1.4 Infrastructure Deployment Execution

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

```
# Listing: AWS infrastructure deployment
# Bringing our infrastructure to life

# 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-LibreChat" "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"
```

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

### 3.1.5 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

# Test initial SSH connection
ssh -i ~/.ssh/librechat_key ec2-user@"$public_ip" 'whoami && pwd'
yes
# Create SSH config entry for easy access
bbedit ~/.ssh/config # we may delete an old entry first, manually
cat >> ~/.ssh/config << EOF
# SSH over EC2 - LibreChat Privacy AI Project
Host EC2-LibreChat
    HostName $public_ip
    User ec2-user
    IdentityFile ~/.ssh/librechat_key
EOF
## use a perl one-liner to grep these lines from the ssh-config file.
perl -nE 'print if /Host EC2-LibreChat/ .. /yesEOF/' ~/.ssh/config
#bbedit ~/.ssh/config

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

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

### 3.1.6 VS Code Remote Development Integration

This enables remote development capabilities by connecting VS Code to the EC2 instance.

```
# Listing: VS Code Remote SSH setup
# Enabling seamless remote development experience

# Install VS Code Remote-SSH extension (if not already installed)
code --install-extension ms-vscode-remote.remote-ssh

# Verify the extension is installed
code --list-extensions | grep ms-vscode-remote.remote-ssh

# Connect to the remote instance using VS Code command line
code --folder-uri vscode-remote://ssh-remote+EC2-LibreChat/home/ec2-user
code --folder-uri vscode-remote://ssh-remote+EC2-LibreChat/home/ec2-user

# Now that VS Code is connected to the remote instance, install Docker extension from Microsoft via the
↪ UI
```

```
# In the VS Code window that just opened (connected to EC2-LibreChat):
# Press Cmd+Shift+P → Type "Extensions: Install Extension" → Search "Docker" → Install

# Test direct SSH access
ssh EC2-LibreChat 'whoami && pwd && uptime'
```

**Result:** VS Code Remote-SSH is configured for direct development access to the EC2 instance.

## 3.2 Use Case: Query Bedrock Model via AWS CLI

This validates AWS Bedrock model access and tests various Claude models directly via CLI.

```
# Listing: Testing AWS Bedrock model access
# Verifying IAM permissions and model availability

# Test Bedrock model listing from EC2 instance
ssh EC2-LibreChat

# Check aws cli
aws --version

# switch of the pager and list the available models
export AWS_PAGER=""
aws bedrock list-foundation-models  --region eu-central-1
aws bedrock list-foundation-models  --region eu-central-1  | grep modelName
aws bedrock list-foundation-models  --region eu-central-1  | grep modelId
aws bedrock list-foundation-models  --region eu-central-1 | grep anthropic
aws bedrock list-foundation-models  --region eu-central-1 | grep deep
aws bedrock list-foundation-models --region us-east-1 | grep deep    # this contains deepseek llm
aws bedrock list-foundation-models --region us-east-1 | grep modelId


# List available foundation models
aws bedrock list-foundation-models --region us-east-1


# Test Claude model availability
aws bedrock list-foundation-models \
  --region us-east-1 \
  --by-provider anthropic \
  --query 'modelSummaries[?contains(modelId, `claude`)].[modelId,modelName]' \
  --output table

# Test Nova model availability
aws bedrock list-foundation-models \
  --region us-east-1 \
  --by-provider amazon \
  --query 'modelSummaries[?contains(modelId, `nova`)].[modelId,modelName]' \
  --output table

# Test a simple model invocation
aws bedrock-runtime invoke-model \
  --region us-east-1 \
  --model-id us.anthropic.claude-3-5-haiku-20241022-v1:0 \
  --body
↪  '{"anthropic_version":"bedrock-2023-05-31","max_tokens":100,"messages":[{"role":"user","content":"Hello,
↪  how are you?"}]}' \
  --cli-binary-format raw-in-base64-out \
  /tmp/response.json
```

```

# Display the response
cat /tmp/response.json | jq '.content[0].text'



# First, let's check what Claude models are actually available
aws bedrock list-foundation-models \
  --region us-east-1 \
  --by-provider anthropic \
  --query 'modelSummaries[?contains(modelId, `claude`)].{ModelId:modelId,ModelName:modelName,Status:modelLifecycleStatus}' \
  --output table


# Test different claude models
export MODEL_ID=us.anthropic.claude-3-7-sonnet-20250219-v1:0
export MODEL_ID=us.anthropic.claude-3-5-haiku-20241022-v1:0
export MODEL_ID=us.anthropic.claude-opus-4-20250514-v1:0
export MODEL_ID=us.anthropic.claude-sonnet-4-20250514-v1:0


# Call Claude model with correct model ID (using Claude 3.5 Sonnet)
aws bedrock-runtime invoke-model \
  --region us-east-1  --model-id $MODEL_ID  \
  --cli-binary-format raw-in-base64-out output.json \
  --body '{"anthropic_version":"bedrock-2023-05-31","max_tokens":500,"messages":[{"role":"user","content":"Wie wird das Wetter morgen in Hamburg?"}]}'

  --body '{"anthropic_version":"bedrock-2023-05-31","max_tokens":500,"messages":[{"role":"user","content":"Hallo, wer bist du? Was sind deine Fähigkeiten?"}]}'

# Display the response
cat output.json | jq -r '.content[0].text'


# Call Claude Modell with more detailed body message
aws bedrock-runtime invoke-model \
--model-id anthropic.claude-3-sonnet-20240229-v1:0 \
--body "{\"messages\":[{\"role\":\"user\",\"content\":[{\"type\":\"text\",\"text\":\"Tell me a short joke about cloud computing\"}]}],\"anthropic_version\":\"bedrock-2023-05-31\",\"max_tokens\":100,\"temperature\":0.7}" \
--cli-binary-format raw-in-base64-out \
--region eu-central-1 \
output.json
cat output.json | jq -r '.content[0].text'
cat output.json


exit
```

**Result:** AWS Bedrock models are successfully accessible and Claude models respond correctly via CLI invocation.

### 3.3 Part 2 Summary

Infrastructure automation through Terraform successfully provisions the AWS environment. The EC2 instance is operational with proper network security configuration. SSH access is established through dedicated keypairs, and VS Code remote development capabilities are configured. The infrastructure exists as version-controlled code, enabling consistent reproduction across different deployments.

## 4 Part 3: Core LibreChat Deployment

### 4.1 The Heart of the System: LibreChat Setup

This is where the magic happens - transforming a bare EC2 instance into a powerful, privacy-respecting AI assistant platform.

#### 4.1.1 Initial Environment Preparation

This prepares the EC2 instance with system updates and creates directories for persistent storage.

```
# Listing: EC2 environment preparation
# Setting up the foundation for LibreChat

# Connect to our instance
ssh EC2-LibreChat

# Update the system
sudo yum update -y

# install some useful tools
sudo yum install -y htop
sudo yum install -y wget


# Create necessary directories for persistent volumes
sudo mkdir -p /opt/librechat/mongodb
sudo mkdir -p /opt/portainer/data
sudo mkdir -p /opt/portainer/data/certs

# Set appropriate permissions
sudo chmod -R 777 /opt/portainer/data
sudo chmod -R 777 /opt/librechat/mongodb


# Exit for now - we'll return with specific tasks
exit
```

**Result:** EC2 instance is updated and persistent storage directories are created with proper permissions for Docker containers.

#### 4.1.2 LibreChat Repository and Docker Setup

This clones the LibreChat repository and installs Docker Compose for container management.

```
# Listing: LibreChat clone and Docker installation
# Getting LibreChat and preparing containerization

ssh EC2-LibreChat
```

```

# Clone the LibreChat repository from this release:
↪ https://github.com/danny-avila/LibreChat/releases/tag/v0.7.8
git clone --branch v0.7.8 https://github.com/danny-avila/LibreChat.git
cd LibreChat

# Verify the correct version is checked out
git describe --tags
git branch


# Install docker-compose
sudo curl -L "https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname
↪ -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose

# Verify installation
docker-compose --version

# Initial LibreChat preparation
cp .env.example .env

# Test basic docker-compose functionality
docker-compose up -d
docker ps -a
#docker-compose down

## Check if LibreChat is started
curl http://localhost:3080


exit
```

**Result:** LibreChat v0.7.8 is successfully cloned, Docker Compose is installed, and the application is running and accessible on port 3080.

## 4.2 Use Case: Connect via SSH-Tunnel, Create User, Chat with user provided api key for antropic.

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

```
# Listing: Testing LibreChat via SSH tunnel

ssh -L 3080:localhost:3080 EC2-LibreChat
# open LibreChat in your browser.
exit
```

When the SSH tunnel is established, LibreChat can be accessed via local browser at: `http://localhost:3080` Resister as: andreas@anwi.gmbh Password: private-ai

**Result:** LibreChat is accessible via SSH tunnel and user accounts can be created. Chat functionality works with user-provided API keys for OpenAI or Anthropic models.

## 4.3 Part 3 Summary

LibreChat deployment establishes the core AI assistant platform. The application is configured with Docker containerization, and SSH tunnel access enables secure local browser connectivity. The system demonstrates successful integration with external AI services through user-provided API keys. Basic chat functionality validates the foundation for advanced features.

# 5 Part 4: Security & Production Readiness

## 5.1 Security Configuration Philosophy

Security isn't an afterthought in this deployment - it's built in from the beginning. Self-hosting gives us complete control, but with that comes the responsibility to secure our system properly.

### 5.1.1 Local Configuration Preparation

This prepares production configuration files on the local machine and transfers them to the EC2 instance.

```
# Listing: Local configuration file preparation
# Preparing production configurations locally

cd "$PROJECT_DIR"
pwd
# Copy docker-compose override file for SSL/HTTPS deployment
scp "$PROJECT_DIR"/configs/docker-compose.override.yml ec2-user@EC2-LibreChat:~/LibreChat/docker-compose.override.yml

# Backup and copy deploy-compose.yml with SSL configuration
ssh EC2-LibreChat "cp ~/LibreChat/deploy-compose.yml ~/LibreChat/deploy-compose.yml.bak"
scp "$PROJECT_DIR"/configs/deploy-compose.yml ec2-user@EC2-LibreChat:~/LibreChat/deploy-compose.yml

# Copy NGINX configuration with SSL
ssh EC2-LibreChat "cp ~/LibreChat/client/nginx.conf ~/LibreChat/client/nginx.conf.bak"
scp "$PROJECT_DIR"/configs/nginx.conf ec2-user@EC2-LibreChat:~/LibreChat/client/nginx.conf

# Verify the differences
ssh EC2-LibreChat "diff ~/LibreChat/deploy-compose.yml ~/LibreChat/deploy-compose.yml.bak"
ssh EC2-LibreChat "diff ~/LibreChat/client/nginx.conf ~/LibreChat/client/nginx.conf.bak"
```

**Result:** Production configuration files are transferred to the EC2 instance and configuration differences are verified.

### 5.1.2 SSL Certificate Generation and Deployment

This generates SSL certificates for HTTPS and configures secure communication.

```
# Listing: SSL certificate setup for HTTPS
# Implementing secure communications

ssh EC2-LibreChat

cd ~/LibreChat

# Create SSL directory
mkdir -p client/ssl

# Generate self-signed SSL certificate for HTTPS
openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
  -keyout client/ssl/nginx.key \
  -out client/ssl/nginx.crt \
```

```
  -subj "/C=US/ST=State/L=City/O=Organization/CN=localhost"

# Generate DH parameters for enhanced security (this may take a few minutes)
openssl dhparam -out client/ssl/dhparam 2048

# Set proper permissions
chmod 644 ./client/ssl/nginx.key
chmod 644 ./client/ssl/nginx.crt
chmod 644 ./client/ssl/dhparam


# Verify SSL files
ls -la client/ssl/

# Update environment for HTTPS domains
cat .env | grep -iE 'DOMAIN_CLIENT|DOMAIN_SERVER'
sed -i 's|\(DOMAIN_.*=\)http://|\1https://|' .env

# Verify HTTPS configuration
grep -E 'DOMAIN_CLIENT|DOMAIN_SERVER' .env

# Create librechat.yaml configuration file
touch librechat.yaml

exit
```

**Result:** SSL certificates are generated and configured for secure HTTPS communication. The protocol is switched from HTTP to HTTPS in the environment configuration.

### 5.1.3 Production Deployment with SSL

This command sequence launches LibreChat in production mode with SSL configuration, which provides secure HTTPS access and proper certificate handling.

```
# Listing: Production deployment launch
# Starting LibreChat with full HTTPS and security

ssh EC2-LibreChat

cd ~/LibreChat

# Stop any running services
docker-compose -f deploy-compose.yml -f docker-compose.override.yml down

# Start production deployment with SSL. This will pull the NGNIX container.
docker-compose -f deploy-compose.yml -f docker-compose.override.yml up -d

# Verify all containers are running
docker ps -a

# Check service logs for any issues
docker logs $(docker ps -q --filter "name=LibreChat-API")

exit

# Display our public IP for browser access
echo "LibreChat is now accessible at: https://$public_ip"

```

```
## open broser to this url

open https://$public_ip
open http://$public_ip

## Check the via CLIcking

ssh EC2-LibreChat
curl https://3.69.0.104
curl http://3.69.0.104
curl http://localhost
curl https://localhost
```

**Result:** LibreChat is now running in production mode with HTTPS enabled. The service can be accessed securely via the public IP address.

### 5.1.4 AWS Bedrock Integration Configuration

This creates AWS access credentials and configures LibreChat to use AWS Bedrock models.

```
# Listing: AWS Bedrock Model Configuration
# Prerequisites: Access to the AWS CLI with IAM user

cd $PROJECT_DIR
pwd

## List all aws access keys for the current user als complete json
aws iam list-access-keys --user-name user-lab-a --profile lab-a --output json

## create a new AWS access key and store it in Environment Variables
aws iam create-access-key --user-name user-lab-a --profile lab-a-north --output json >>
↪  ./aws-credentials.json
cat ./aws-credentials.json
export AWS_ACCESS_KEY_ID=$(jq -r '.AccessKey.AccessKeyId' ./aws-credentials.json)
export AWS_SECRET_ACCESS_KEY=$(jq -r '.AccessKey.SecretAccessKey' ./aws-credentials.json)
echo $AWS_ACCESS_KEY_ID
echo $AWS_SECRET_ACCESS_KEY  # should not be printed out, but in this case I delete it in the clean-up
↪  section.

## Check if the status of the AWS access key is active
aws iam list-access-keys --user-name user-lab-a --profile lab-a-north --output json | jq -r
↪  '.AccessKeyMetadata[] | select(.Status == "Active") | .AccessKeyId'

## Copy .env file from the ec2 instance to the local machine
mkdir -p $PROJECT_DIR/LibreChat
ssh EC2-LibreChat "whoami; ls -la ~/LibreChat/.env.example"
scp ec2-user@EC2-LibreChat:~/LibreChat/.env.example $PROJECT_DIR/LibreChat/.env.example
ls -la LibreChat
cp LibreChat/.env.example LibreChat/.env

## Update the Bedrock Config using a here document
cat >> $PROJECT_DIR/LibreChat/.env << EOF

#=================#
#   AWS Bedrock   #
#=================#

BEDROCK_AWS_DEFAULT_REGION=us-east-1
```

```
BEDROCK_AWS_ACCESS_KEY_ID=$AWS_ACCESS_KEY_ID
BEDROCK_AWS_SECRET_ACCESS_KEY=$AWS_SECRET_ACCESS_KEY

BEDROCK_AWS_MODELS="
--us-inferance-profiles,
us.anthropic.claude-opus-4-20250514-v1:0,
us.anthropic.claude-sonnet-4-20250514-v1:0,
us.anthropic.claude-3-7-sonnet-20250219-v1:0,
us.deepseek.r1-v1:0,
us.anthropic.claude-3-5-sonnet-20241022-v2:0,
us.anthropic.claude-3-5-haiku-20241022-v1:0,
us.meta.llama3-3-70b-instruct-v1:0,
--us-east-1--,
amazon.titan-text-lite-v1,
amazon.nova-micro-v1:0,
amazon.nova-lite-v1:0,
amazon.nova-pro-v1:0,
mistral.mistral-large-2402-v1:0,
mistral.mistral-small-2402-v1:0,
"
EOF

# check if the env file is updated
tail -n 50 $PROJECT_DIR/LibreChat/.env

## scp the env file to the instance
scp $PROJECT_DIR/LibreChat/.env ec2-user@EC2-LibreChat:~/LibreChat/.env

## Login to the instance and restart the docker-compose
ssh EC2-LibreChat
whoami; pwd
cd ~/LibreChat
docker-compose -f deploy-compose.yml -f docker-compose.override.yml down
docker-compose -f deploy-compose.yml -f docker-compose.override.yml up -d
exit

## open via public ip
echo $public_ip
open https://$public_ip
```

**Result:** AWS Bedrock models are configured and accessible via the LibreChat instance. I can now interact with various AI models directly from the chat interface.

## 5.2 Use Case: Chat with AWS Bedrock Models, Agents for Calculation and Internet Access

**Chat with different Bedrock Model:** Asking: Who are you? What are your capabilities?

**Nova Agnet:** Create this Agent using the Nova Pro model from AWS Bedrock. Name: Nova Agent Instruction: You are a helpful AI assistant that can use tools.

Test-Case: Calculate the harmonic series for n=5 to a precison of 10 digits. Test-Case: Find the square root of 999999937

## 5.3 Part 4 Summary

Production security implementation transforms the development system into enterprise-ready deployment. SSL certificate generation enables HTTPS encryption, and AWS Bedrock integration provides access to advanced AI models without external API dependencies. The system operates securely over public internet while maintaining data privacy through self-hosted architecture.

# 6 Part 5: Advanced Features - RAG Integration

## 6.1 Why RAG Matters for Privacy

Retrieval Augmented Generation (RAG) represents the pinnacle of this privacy-focused approach. Instead of sending sensitive documents to external AI services, I can process them locally while still leveraging powerful cloud models for reasoning.

### 6.1.1 Ollama Container Deploymentssh EC2-LibreChat

This deploys Ollama container for local RAG processing and installs the embedding model.

```
# Listing: Ollama installation for local RAG processing
# Setting up local model inference for embeddings

# Prepare Ollama-specific Docker configuration
cd "$PROJECT_DIR"; pwd
scp "$PROJECT_DIR"/configs/docker-compose.override.yml.ollama
↪  ec2-user@EC2-LibreChat:~/LibreChat/docker-compose.override.yml

# Deploy with Ollama integration
ssh EC2-LibreChat
cd ~/LibreChat
docker-compose -f deploy-compose.yml -f docker-compose.override.yml down
docker-compose -f deploy-compose.yml -f docker-compose.override.yml up -d
docker ps -a


# Install embedding model
docker exec -it $(docker ps -q --filter "name=ollama") bash
# verify that we are inside the container
whoami; hostname; pwd; uname -a
ollama --version
ollama pull nomic-embed-text
ollama list
exit # docke-container
exit # ec2-instance

exit
```

**Result:** The installation now includes Ollama for local RAG processing. The container is deployed and the nomic-embed-text model is pulled for embeddings.

### 6.1.2 RAG Environment Configuration

This configures LibreChat environment variables for local RAG processing and tests Ollama API connectivity.

```
# Listing: RAG environment setup
# Configuring LibreChat for local RAG processing

ssh EC2-LibreChat
cd ~/LibreChat
```

```

# Add Ollama RAG configuration to environment
cat >> .env << EOF

#=================#
#   Ollama RAG    #
#=================#
# Use Ollama for embeddings
RAG_API_URL=http://host.docker.internal:8000
EMBEDDINGS_PROVIDER=ollama
OLLAMA_BASE_URL=http://host.docker.internal:11434
EMBEDDINGS_MODEL=nomic-embed-text

EOF

# Verify configuration
tail -n 10 .env

# Test Ollama API connectivity
docker exec -it $(docker ps -q --filter "name=rag_api") sh
whoami; hostname; pwd; uname -a
curl http://ollama:11434/api/version

## Check the embeddings api
curl http://ollama:11434/api/embeddings -d '{
  "model": "nomic-embed-text",
  "prompt": "The sky is blue because of Rayleigh scattering"
}'

exit # container
exit # ec2-instance

# copy the librechat.yaml to the ec2 instance
scp $PROJECT_DIR/configs/librechat_ollama.yaml ec2-user@EC2-LibreChat:~/LibreChat/librechat.yaml

ssh EC2-LibreChat
cd ~/LibreChat

# Restart services with RAG configuration
docker-compose -f deploy-compose.yml -f docker-compose.override.yml down
docker-compose -f deploy-compose.yml -f docker-compose.override.yml up -d

## It may be necessary to check the firewall settings on the EC2 instance to allow traffic on port 11434.
# Check if port 11434 is allowed
sudo iptables -L -n | grep 11434

# Allow traffic if needed
sudo iptables -A INPUT -p tcp --dport 11434 -j ACCEPT

exit
```

**Result:** The .env file is updated to include Ollama RAG configuration, enabling local embeddings processing. The access from RAG container to the Ollama API is verified. The embedding was testet with a sample text.

### 6.1.3 Demonstration of RAG

Now we select the model nova pro and start a new chat. Frist I ask who Andreas Wittmann it at it usally finds a musician with the same name.

I drag&drop a pdf of the CV of Andreas Wittmann. It has a text layer. The upload takes some time. I check the cpu usage with top. ollama command consumes 100% CPU. It is busy on the embedding activity for about 1 minute. After this finishes, we can query about Andreas Wittmann again.

### 6.1.4 Installating Ollama Models for Inference

This installs additional Ollama models for inference and tests their functionality via API calls.

```
# Listing: Advanced model deployment
# Installing additional models for various use cases

ssh EC2-LibreChat

# Access Ollama container for model management
docker exec -it $(docker ps -q --filter "name=ollama") bash

# Install additional models for various use cases
ollama pull deepseek-r1:8b
ollama pull allenporter/xlam:7b  # Tool-capable model for RAG
ollama pull mistral-nemo

# List all available models
ollama list

exit
# Test model functionality
curl -X POST http://localhost:11434/api/generate \
     -H "Content-Type: application/json" \
     -d '{
        "model": "deepseek-r1:8b",
        "prompt": "What is the capital of Spain?",
        "max_tokens": 50
     }'



# Test model functionality
curl -X POST http://localhost:11434/api/generate \
     -H "Content-Type: application/json" \
     -d '{
        "model": "mistral-nemo",
        "prompt": "What is the capital of Spain?",
        "max_tokens": 50
     }'


# Restart services to integrate new models
cd ~/LibreChat
docker-compose -f deploy-compose.yml -f docker-compose.override.yml down
docker-compose -f deploy-compose.yml -f docker-compose.override.yml up -d

exit
```

**Result:** Three modells for inference are installed in the Ollama container. However the test using the api with curl reveals that the machine does not have enough memory to run the models. We need to upgrade the instance type to a more powerful instance type.

### 6.2 Part 5 Summary

RAG integration enables advanced document processing while preserving privacy. Ollama container deployment provides local embedding generation, eliminating the need to transmit sensitive documents to external services. The hybrid architecture combines local document processing with cloud-based reasoning, creating a comprehensive AI assistant that maintains complete data control.

## 7 Part 6: Usinig Ollama for inferance.

- Upgrade to g5.xlarge
- Install ollama inference models.
- install gpu monitoring tool
- Demonstrate inference.

#### 7.0.1 Change to a more powerful instance type

The t3.medium instance type is not powerful enough to run the Ollama container. It is working with small files, however bigger files take just to long or provocate an error. Inerence doesn't work as all.

**Recommendations:** GPU-accelerated instances (best for embeddings):

- g4dn.xlarge: 4 vCPUs, 16GB RAM, 1 NVIDIA T4 GPU, 16 GiB VRAM - good balance of performance/cost
- g6e.xlarge: 4 vCPUs, 16GB RAM, 1x AMD Radeon Pro V620 GPU, 32 GiB VRAM - AI workloads with bigger VRAM

CPU-only alternatives (if cost is a concern):

- c6i.2xlarge: 8 vCPUs, 16GB RAM - compute-optimized without GPU
- r6i.xlarge: 4 vCPUs, 32GB RAM - memory-optimized for larger models

This uses Terraform to upgrade the EC2 instance type to a GPU-enabled instance for better performance.

```
# Listing: Using terraform to change the instance type
# Prerequisites: We start from the local machine
exit
whoami; pwd;
cd $PROJECT_DIR
cd terraform
pwd; ls -la

# Use a perl oneliner to change the instance type in the terraform.tfvars file to g4dn.xlarge, only print to stdout
perl -pe 's/instance_type    = ".*"/instance_type    = "t3.medium"/' terraform.tfvars
perl -pe 's/instance_type    = ".*"/instance_type    = "g4dn.xlarge"/' terraform.tfvars
perl -pe 's/instance_type    = ".*"/instance_type    = "g6e.large"/' terraform.tfvars

## And chnage the file
perl -pi -e  's/instance_type    = ".*"/instance_type    = "t3.medium"/' terraform.tfvars
perl -pi -e  's/instance_type    = ".*"/instance_type    = "g4dn.xlarge"/' terraform.tfvars  # GPU-Mem 16 GiB
perl -pi -e  's/instance_type    = ".*"/instance_type    = "g6e.xlarge"/' terraform.tfvars   # GPU-Mem 4x24 GiB

```

```
cat terraform.tfvars

# Create a new plan
terraform plan
terraform plan -out=tfplan -var-file=terraform.tfvars
terraform show -json tfplan > tfplan.json
cat tfplan.json | jq > tfplan.pretty.json
cat tfplan.json | jq '.resource_changes[] | {address: .address, action: .change.actions[0]}'

# Apply the new plan
terraform apply
yes
```

```
Note: The instance failed to start. I had to request a quota increase for the Running On-Dema
```

**Result:** The instance type is changed to g4dn.xlarge, which has a NVIDIA T4 GPU with 16 GiB VRAM.

### 7.0.2 Installing NVIDIA Drivers and CUDA for GPU Support

This installs NVIDIA drivers and CUDA toolkit to enable GPU acceleration for AI workloads.

```
# Listing: Installing NVIDIA Drivers and CUDA for GPU Support

ssh EC2-LibreChat

# Show machine details, like CPU, GPU, RAM
lscpu
lspci | grep -i nvidia


## Extra Configuration for GPU usgae
sudo su

# Install NVIDIA drivers and CUDA
# Update system packages
dnf update -y


##########  this works!!!! [2025-06-01 Sun 12:31]
# Install required tools
sudo dnf install -y gcc kernel-devel-$(uname -r) make

# Install NVIDIA drivers through AWS package manager
sudo dnf config-manager --add-repo
↪  https://developer.download.nvidia.com/compute/cuda/repos/rhel9/x86_64/cuda-rhel9.repo

sudo dnf clean all
sudo dnf -y module install nvidia-driver:latest-dkms

# reboot to activate the NVIDIA drivers
shutdown -r now
ssh EC2-LibreChat

# Check driver installation
nvidia-smi  # NVIDIA-SMI has failed because it couldn't communicate with the NVIDIA driver

```

```

### Troubleshooting
#Fixing NVIDIA Driver Symbol Error on Amazon Linux 2023
# 1. Remove existing NVIDIA packages
sudo dnf remove -y '*nvidia*' '*cuda*'

# 2. Install kernel headers that match exactly
sudo dnf install -y kernel-devel-$(uname -r) kernel-headers-$(uname -r)

# 3. Install required packages
sudo dnf install -y gcc make dkms

# 4. Install DRM kernels (missing dependency)
sudo dnf install -y kernel-modules-extra

# 5. Reinstall NVIDIA drivers using Amazon-specific method
sudo dnf config-manager --add-repo
↪ https://developer.download.nvidia.com/compute/cuda/repos/rhel9/x86_64/cuda-rhel9.repo
sudo dnf clean all
sudo dnf -y module install nvidia-driver:latest-dkms

# 6. Reboot to load the driver
sudo reboot
ssh EC2-LibreChat




#---------

# Load modules
sudo modprobe nvidia
sudo modprobe nvidia_uvm


# Download the CUDA installer for Amazon Linux 2023
cd /tmp
wget
↪ https://developer.download.nvidia.com/compute/cuda/12.2.0/local_installers/cuda_12.2.0_535.54.03_linux.run

# Add execute permissions
chmod +x cuda_12.2.0_535.54.03_linux.run


# Create a new temporary directory and Use this directory for the installation
mkdir ~/cuda_tmp
# Run the installer (silent mode with custom options)
sudo TMPDIR=~/cuda_tmp sh cuda_12.2.0_535.54.03_linux.run --silent --override --toolkit --samples
↪ --toolkitpath=/usr/local/cuda-12.2 --samplespath=/usr/local/cuda --no-opengl-libs

# Set as default CUDA version
sudo ln -s /usr/local/cuda-12.2 /usr/local/cuda

## Check CUDA installation
/usr/local/cuda/bin/nvcc --version

# Check if libraries exist
ls -l /usr/local/cuda/lib64


```

```
# Install NVIDIA Container Toolkit if not already installed
sudo dnf install -y nvidia-container-toolkit
sudo nvidia-ctk runtime configure --runtime=docker
sudo systemctl restart docker

# Test GPU support in Docker
docker run --rm --gpus all nvidia/cuda:12.2.0-base-ubuntu22.04 nvidia-smi

exit
```

**Result:** NVIDIA drivers and CUDA toolkit are successfully installed on the EC2 instance. The ‘nvidia-smi‘ command shows the GPU is recognized and ready for use.

### 7.0.3 Updating Docker Configuration for Ollama with GPU Support

This updates the Docker configuration to enable GPU access for Ollama containers.

```
# Listing: Updating Docker Configuration for Ollama with GPU Support



# Update Ollama-specific Docker configuration
cd "$PROJECT_DIR"; pwd
scp "$PROJECT_DIR"/configs/docker-compose.override.yml.ollama_gpu
↪  ec2-user@EC2-LibreChat:~/LibreChat/docker-compose.override.yml


## Login to container and restart docker-compose
ssh EC2-LibreChat
whoami; pwd
cd ~/LibreChat
docker ps -a
docker-compose -f deploy-compose.yml -f docker-compose.override.yml down
docker-compose -f deploy-compose.yml -f docker-compose.override.yml up -d

# Verify Ollama is Using GPU
# Check running containers
docker ps

# Check GPU usage by running nvidia-smi
nvidia-smi

# Check logs from Ollama container
docker logs $(docker ps -q --filter name=ollama)

exit
exit
```

**Result:** The Docker configuration is updated to allow Ollama to utilize the GPU. The Ollama container is restarted with GPU support, and the ‘nvidia-smi‘ command confirms that the GPU is being used.

### 7.0.4 Testing Ollama with GPU Support

In the LibreChat GUI start a chat with ollama model: mistral-nemo:latests. I provide the CV and ask about Andreas Wittmann. The answer is ok. I provide the prompting study. It is uploaded fast. I switch to deepseek-r1:8b I ask about the prompting study: What are the key take-aways of this prompting study? The response uses full GPU power (90%+), and the response is generated in more than 2 minutes. The answer is somewhat confused. I switch to xlam:7b and ask the same question. It takes about 1 minute to load the model into memory, but the response is generated in less than 10 seconds. The anwsers are ok.

### 7.0.5 Monitoring GPU usage

This installs GPU monitoring tools to track GPU utilization and performance.

```
# Listing: Install monitoring tools for GPU usage

ssh EC2-LibreChat
whoami; pwd

## Simple command to check GPU usage
nvidia-smi -l 1  # This will refresh every second and run indefinitely until stopped with Ctrl+C


#### Install pip if not already installed
sudo dnf install -y python3-pip

### Install gpustat
pip3 install gpustat

# Run gpustat
gpustat


#### Install nvitop
pip install nvitop --user
nvitop
q
```

**Result:** The GPU usage can be monitored using 'nvidia-smi', 'gpustat', and 'nvitop'. The 'nvitop' tool provides a continuous view of GPU utilization, memory usage, and processes using the GPU.

### 7.0.6 Using Ollmama for interference

So far we have only used the ollama container to generate embeddings for the document search. But it can also be used as a LLM for inference.

Candidate LLMS for g4dn.xlarge instance type.

**xLAM-7b-r** Salesforce/xLAM-7b-r · Hugging Face This model is a 7B parameter model that is optimized for angentic tasks.

allenporter/xlam:7b

ollama pull allenporter/xlam

**deepseek-r1** deepseek-r1 This model is a 7B parameter model that is optimized for reasoning.

ollama run deepseek-r1:7b

ollama run deepseek-r1:8b

**eramax/salesforce-iterative-llama3-8b-dpo-r** eramax/salesforce-iterative-llama3-8b-dpo-r:Q5$_{\mathrm{KM}}$ This is an instruct model that is quantized from the llama 3 model.

ollama run eramax/salesforce-iterative-llama3-8b-dpo-r:Q5$_{\mathrm{KM}}$

```
# Listing: Configuring Ollmama for interference
# Prerequisites: EC2-LibreChat is startet and ollama container is running.

PROJECT_DIR=~/LocalProjects/ai-bootcamp/private-ai/
```

```
cd $PROJECT_DIR
pwd; whoami
ssh EC2-LibreChat
whoami; pwd
cd ~/LibreChat

## Check if the ollama container is running
docker ps -a
docker ps -q --filter "name=ollama"
docker exec -it $(docker ps -q --filter "name=ollama") bash
ollama --version
ollama list
# pull ohter models

ollama pull deepseek-r1:8b
ollama pull deepseek-r1:14b
ollama pull allenporter/xlam:7b
ollama pull mistral-nemo              # 7GB
ollama pull llama3.3:70b-instruct-q2_K # 26gb   # it is fast, but not usable for RAG. I have to persuade it to use the knowledg base. the answers are mostly nonsens.
ollama pull llama3:70b-instruct-q4_K_M # ~38-42 GB Excellent for general chat and instruction-following, the model loads but hangs and does not give an answer.
ollama pull open-orca-platypus2       # 26 GB Ansers are a bit confused but very fast on the g6.12xlarge instance type.
ollama rm open-orca-platypus2       # 26 GB Ansers are a bit confused but very fast on the g6.12xlarge instance type.
ollama pull qwq:32b-q8_0            # 35GB
#ollama pull qwq:32b                 # 20GB
#ollama pull qwq:32b-preview-q4_K_M  # 20GB
#ollama pull qwq:32b-preview-q8_0    # 35GB

ollama rm mistral-small3.1:24b     # 15GB tool use. Fast on single reqeusts, but produces nonsens in RAG
ollama rm qwq:32b                 # 20GB Quite fast and impressive in chat but fails in RAG
ollama rm qwq:32b-q8_0            # 35GB, inference take ca. 4 minutes. Probably to load it into memory. Second call is fast.

ollama rm llama3:70b-instruct-q4_K_M

exit # container

# Check the models by quering the API from the commandline
curl -X POST http://localhost:11434/api/generate \
     -H "Content-Type: application/json" \
     -d '{
        "model": "deepseek-r1:8b",
        "prompt": "What is your name?",
        "max_tokens": 50
     }'
        "prompt": "What is the capital of France?",

# Check the models by quering the API from the commandline
curl -X POST http://localhost:11434/api/generate \
     -H "Content-Type: application/json" \
     -d '{
        "model": "allenporter/xlam:7b",
        "prompt": "What is the capital of France?",
        "max_tokens": 50
     }'

```

```
# Check the models by quering the API from the commandline
curl -X POST http://localhost:11434/api/generate \
     -H "Content-Type: application/json" \
     -d '{
        "model": "eramax/salesforce-iterative-llama3-8b-dpo-r:Q5_K_M",
        "prompt": "What is the capital of France?",
        "max_tokens": 50
     }'

# Check the models by quering the API from the commandline
curl -X POST http://localhost:11434/api/generate \
     -H "Content-Type: application/json" \
     -d '{
        "model": "qwq:32b-q8_0",
        "prompt": "What is the capital of Germany?",
        "max_tokens": 50
     }'

# Check the models by quering the API from the commandline
curl -X POST http://localhost:11434/api/generate \
     -H "Content-Type: application/json" \
     -d '{
        "model": "qwq:32b",
        "prompt": "What is the capital of Sweden?",
        "max_tokens": 50
     }'

# Check the models by quering the API from the commandline
curl -X POST http://localhost:11434/api/generate \
     -H "Content-Type: application/json" \
     -d '{
        "model": "mistral-small3.1:24b",
        "prompt": "What is the capital of Spain?",
        "max_tokens": 50
     }'


# Check the models by quering the API from the commandline # requiers 13,1 GiB GPU memory
curl -X POST http://localhost:11434/api/generate \
     -H "Content-Type: application/json" \
     -d '{
        "model": "llama3.3:70b-instruct-q2_K",
        "prompt": "What is the capital of Spain?",
        "max_tokens": 50
     }'

# Check the models by quering the API from the commandline
curl -X POST http://localhost:11434/api/generate \
     -H "Content-Type: application/json" \
     -d '{
        "model": "open-orca-platypus2",
        "prompt": "What is the capital of Spain?",
        "max_tokens": 50
     }'

exit
##

## Configure these models in librechat via VSCode.

```

```
## restart the docker-compose
cd ~/LibreChat
ls -la

docker-compose -f deploy-compose.yml -f docker-compose.override.yml down
docker-compose -f deploy-compose.yml -f docker-compose.override.yml up -d
```

**Result:** The ollama models could be loaded into the container. The models can be used for inference via the API. The models can be used in the LibreChat application, after configuring them in the librechat.yaml file.

Different models work für inference. But they are very slow. I have to wait about 20s for the response.

### 7.0.7 Upgrade to a more powerful instance type

The g4dn.xlarge instance type is not powerful enough to run the ollama models for inference. The inference takes too long.

I configure the instance type to g6.12xlarge, which has 48 vCPUs g6.12xlarge

I follow the instruction in: Change to a more powerful instance type

The switch take about 6 minutes

I report the runtime of the g6.12xlarge instance type, to keep track of the costs. *[2025-06-02 Mon 22:35]* g6.12xlarge instance type is started. *[2025-06-02 Mon 23:21]* g6.12xlarge instance type is stopped.

I loaded different models, the inference is very fast. The accuracy of the medium sized models is better. The llama3:70b-instruct-q4$_{\mathrm{KM}}$ could be loaded but failed to provide an anwser.

**Result:** The g6.12xlarge instance type could be started without changing the driver or CUDA installation. The overall performance was execllent. Medium-sized models run performant. RAG functionality can be used and performs well. This could also serve a multi-user environment with 1-20 user. More analysis is needed to choose the optimal model for this instance type and also to tune the system.

### 7.0.8 Destroy the AWS resources and clean update

This safely removes all AWS infrastructure and cleans up credentials to avoid ongoing costs.

```
# Listing: Destroy the AWS resources and clean update

whoami; pwd;
cd $PROJECT_DIR
cd terraform
pwd; ls -la

## destroy the instance
terraform destroy
yes



## delete the aws keys and clean up
echo $AWS_ACCESS_KEY_ID
aws iam delete-access-key --user-name user-lab-a --access-key-id $AWS_ACCESS_KEY_ID --profile lab-a
# clean up
rm ./aws-credentials.json
```

**Result:** All AWS infrastructure is destroyed and credentials are safely removed to prevent ongoing charges.

### 7.1 Use Case: Chat with ollama models. RAG with ollama models.

### 7.2 Part 6 Summary

Part 6 accomplishes the transformation of the LibreChat deployment from a basic inference system to a high-performance GPU-accelerated AI platform capable of running local large language models. The section establishes GPU infrastructure through instance type upgrades to g4dn.xlarge and g6.12xlarge configurations, enabling NVIDIA driver and CUDA toolkit installation for hardware acceleration support.

The implementation demonstrates successful integration of multiple Ollama models including DeepSeek-R1, xLAM-7b, and Mistral variants, each optimized for different computational requirements and use cases. The deployment includes comprehensive monitoring capabilities through nvidia-smi, gpustat, and nvitop tools, providing real-time visibility into GPU utilization and performance metrics.

Testing reveals significant performance variations across different model sizes and instance types. The g6.12xlarge configuration enables practical multi-user deployment scenarios while maintaining responsive inference times for medium-sized models. The section concludes with proper resource cleanup procedures, demonstrating cost-conscious cloud resource management practices.

This phase establishes a fully functional local AI inference platform that maintains data privacy while delivering enterprise-grade performance capabilities.

## 8 Reflection & Lessons Learned

### 8.1 Technical Insights

The implementation reveals several key considerations. Docker Compose provides an effective balance between simplicity and functionality for multi-container orchestration. Security-first design through initial HTTPS implementation creates more robust foundations than retrofitting SSL later.

The hybrid architecture combining local Ollama models for embeddings with cloud Bedrock models for reasoning demonstrates practical privacy-performance balance. Local processing handles sensitive document embedding while cloud capabilities provide complex reasoning. This separation enables fine-grained data exposure control.

Infrastructure automation through Terraform transforms manual processes into reproducible, version-controlled workflows. The live-scripting methodology bridges documentation and execution, making complex deployments more accessible.

Resource requirements for RAG functionality with Ollama require careful instance sizing consideration. The computational overhead of embedding generation and vector similarity searches significantly impacts performance on undersized instances. Docker networking for multi-container applications needs thoughtful planning, especially when access patterns vary between development and production.

### 8.2 Alternative Approaches

Infrastructure alternatives include Kubernetes for production scaling, AWS ECS/Fargate for serverless containers, and local Docker Desktop for development. Security enhancements could involve VPN access restriction, WAF integration, or comprehensive monitoring through CloudWatch or Prometheus.

Cost optimization options include spot instances for development environments, ARM instances for better price-performance ratios, and multi-region deployments for disaster recovery and performance optimization.

Future enhancements might include multi-modal support for images and audio, custom model training capabilities, agent framework integration, high availability deployment, comprehensive backup strategies, and enhanced monitoring capabilities.

## 8.3 Assessment

This project demonstrates that privacy-focused AI systems are both technically feasible and practically deployable. The combination of open-source tools, cloud infrastructure, and security design creates a platform that maintains user privacy while delivering enterprise-grade AI capabilities.

The live-scripting methodology provides value for both development and knowledge transfer. Each executable code block serves dual purposes as implementation step and educational content.

This approach establishes user control over AI infrastructure where data sovereignty concerns continue to grow. Self-hosted solutions provide alternatives to cloud-only AI services, though they require accepting operational responsibility.

—

**This completes the LibreChat AWS 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.**