

LibreChat AWS Deployment - Live-Scripting Workflow

Andreas Wittmann

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

2025-01-31

Contents

1	Building a Privacy-Focused AI Assistant	1
1.1	Project Overview and Privacy Philosophy	1
1.2	The Live-Scripting Methodology	1
1.2.1	Implementation Options	1
	Emacs Users	1
	Non-Emacs Users	2
1.3	Project Components	2
1.4	Expected Deployment Outcome	2
2	Part 1: Foundation & Prerequisites	2
2.1	Environment Setup	2
2.1.1	Setting the Project Foundation	2
2.1.2	Validating AWS Prerequisites	3
2.2	Part 1 Summary	3
3	Part 2: Infrastructure as Code	3
3.1	Infrastructure Automation Benefits	3
3.1.1	SSH Key Generation for Secure Access	3
3.1.2	Terraform Configuration and Planning	4
3.1.3	Infrastructure Deployment Planning	4
3.1.4	Infrastructure Deployment Execution	5
3.1.5	SSH Configuration for Easy Access	5
3.1.6	VS Code Remote Development Integration	6
3.2	Use Case: Query Bedrock Model via AWS CLI	6
3.3	Part 2 Summary	8
4	Part 3: Core LibreChat Deployment	8
4.1	The Heart of the System: LibreChat Setup	8
4.1.1	Initial Environment Preparation	8
4.1.2	LibreChat Repository and Docker Setup	9
4.2	Use Case: Connect via SSH-Tunnel, Create User, Chat with user provided api key for antropic.	10
4.3	Part 3 Summary	10

5	Part 4: Security & Production Readiness	10
5.1	Security Configuration Philosophy	10
5.1.1	Local Configuration Preparation	10
5.1.2	SSL Certificate Generation and Deployment	11
5.1.3	Production Deployment with SSL	12
5.1.4	AWS Bedrock Integration Configuration	12
5.2	Use Case: Chat with AWS Bedrock Models, Agents for Calculation and Internet Access .	14
5.3	Part 4 Summary	14
6	Part 5: Advanced Features - RAG Integration	14
6.1	Why RAG Matters for Privacy	14
6.1.1	Ollama Container Deploymentssh EC2-LibreChat	14
6.1.2	RAG Environment Configuration	15
6.1.3	Demonstration of RAG	16
6.1.4	Installing Ollama Models for Inference	16
6.2	Part 5 Summary	17
7	Part 6: Using Ollama for inference.	17
7.0.1	Change to a more powerful instance type	18
7.0.2	Installing NVIDIA Drivers and CUDA for GPU Support	19
7.0.3	Updating Docker Configuration for Ollama with GPU Support	20
7.0.4	Testing Ollama with GPU Support	21
7.0.5	Monitoring GPU usage	21
7.0.6	Using Ollama for inference	22
7.0.7	Upgrade to a more powerful instance type	24
7.0.8	Destroy the AWS resources and clean up	25
7.1	Use Case: Chat with ollama models. RAG with ollama models.	25
7.2	Part 6 Summary	25
8	Reflection & Lessons Learned	26
8.1	Technical Insights	26
8.2	Alternative Approaches	26
8.3	Assessment	26
	• 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:

```
1 (defun send-line-to-vterm ()
2   "Send region if active, or current line to vterm buffer."
3   (interactive)
4   (if (region-active-p)
5       (send-region "*vterm*" (region-beginning) (region-end))
6       (my-select-current-line)
7       (send-region "*vterm*" (region-beginning) (region-end)))
8   (deactivate-mark))
9
10 (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.

```
1  # Listing: Project environment setup
2  # Establishing the base directory for our privacy-focused AI project
3
4  PROJECT_DIR="/Users/$(whoami)/LocalProjects/ai-bootcamp/private-ai"
5  cd "$PROJECT_DIR"
6  pwd
7  ls -la
8
9  # Verify we're in the right location
10 echo "Working in: $PROJECT_DIR"
11
12 # Set AWS profile for this session
13 export AWS_PROFILE=lab-a-north
14 echo $AWS_PROFILE
15
16 ## create default VPC if it does not exist
17 aws ec2 create-default-vpc
18 ## check if the default vpc is created
19 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.

```
1  # Listing: AWS environment validation
2  # Ensuring AWS access and Bedrock availability
3
4  # Test basic AWS connectivity
5  aws s3 ls
6
7  # Verify AWS identity and permissions
8  aws sts get-caller-identity
9
10 # Verify SSH key is available for EC2 access
11 ls -la ~/.ssh/
12
13 # Validate required tools are installed
14 terraform --version
15 git --version
16 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.

```

1 # Listing: SSH key generation for EC2 access
2 # Creating dedicated SSH keys for secure instance access
3
4 cd "$PROJECT_DIR"
5
6 # Generate a new SSH key pair specifically for LibreChat
7 ssh-keygen -t rsa -b 4096 -f ~/.ssh/librechat_key -N ""
8 y
9 # Display the public key for use in Terraform variables
10 cat ~/.ssh/librechat_key.pub
11
12 # Store public key in environment variable
13 export TF_VAR_SSH_PUBLIC_KEY="$(cat ~/.ssh/librechat_key.pub)"
14 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.

```

1 # Listing: Terraform environment preparation
2 # Setting up infrastructure configuration
3
4 cd "$PROJECT_DIR/terraform"
5
6 # Create terraform variables file with our configuration
7 cat > terraform.tfvars << EOF
8 # Generated tfvars file - $(date)
9 aws_region      = "eu-north-1"
10 instance_type   = "t3.medium"
11 root_volume_size = 100
12 root_volume_type = "gp3"
13 allowed_ip      = "0.0.0.0/0"
14 ssh_public_key  = "${TF_VAR_SSH_PUBLIC_KEY}"
15 ec2_name        = "EC2-LibreChat"
16 environment     = "development"
17 project         = "private-ai"
18 ssh_cidr_blocks = ["0.0.0.0/0"]
19 EOF
20
21 # Review our configuration
22 cat terraform.tfvars
23
24 # Initialize Terraform
25 terraform init
26

```

```

27 # Format and validate our configuration
28 terraform fmt
29 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.

```

1 # Listing: Terraform deployment planning
2 # Creating and reviewing the deployment plan
3
4 # Generate deployment plan
5 terraform plan -out=tfplan -var-file=terraform.tfvars
6
7 # Create human-readable plan
8 terraform show -json tfplan > tfplan.json
9 cat tfplan.json | jq > tfplan.pretty.json
10
11 # Review what resources will be created
12 echo "Resources to be created:"
13 cat tfplan.json | jq '.resource_changes[] | {address: .address, action: .change.actions[0]}'
14
15 # Summary of planned changes
16 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.

```

1 # Listing: AWS infrastructure deployment
2 # Bringing our infrastructure to life
3
4 # Execute the deployment plan
5 terraform apply tfplan
6 terraform apply
7 yes
8
9 # Capture and display outputs
10 terraform output
11
12 # List our EC2 instances
13 aws ec2 describe-instances \
14   --query 'Reservations[*].Instances[*].[InstanceId, InstanceType, Tags[?Key==`Name`][0].Value,
  ↳ State.Name]' \
15   --output text
16
17 # Get our instance details - get only the first/latest instance ID
18 instance_id=$(aws ec2 describe-instances \
19   --filters "Name=tag:Name,Values=EC2-LibreChat"
  ↳ "Name=instance-state-name,Values=running,pending,stopped" \
20   --query 'Reservations[*].Instances[*].[InstanceId]' \
21   --output text | head -n1)
22 echo "Instance ID: $instance_id"

```

```

23
24
25 public_ip=$(aws ec2 describe-instances \
26     --instance-ids "$instance_id" \
27     --query 'Reservations[*].Instances[*].[PublicIpAddress]' \
28     --output text)
29 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.

```

1  # Listing: SSH configuration setup
2  # Establishing convenient SSH access
3
4  # Test initial SSH connection
5  ssh -i ~/.ssh/librechat_key ec2-user@"$public_ip" 'whoami && pwd'
6  yes
7  # Create SSH config entry for easy access
8  bbedit ~/.ssh/config # we may delete an old entry first, manually
9  cat >> ~/.ssh/config << EOF
10 # SSH over EC2 - LibreChat Privacy AI Project
11 Host EC2-LibreChat
12     HostName $public_ip
13     User ec2-user
14     IdentityFile ~/.ssh/librechat_key
15 EOF
16 ## use a perl one-liner to grep these lines from the ssh-config file.
17 perl -nE 'print if /Host EC2-LibreChat/ .. /yesEOF/' ~/.ssh/config
18 #bbedit ~/.ssh/config
19
20 # Test SSH with hostname
21 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.

```

1  # Listing: VS Code Remote SSH setup
2  # Enabling seamless remote development experience
3
4  # Install VS Code Remote-SSH extension (if not already installed)
5  code --install-extension ms-vscode-remote.remote-ssh
6
7  # Verify the extension is installed
8  code --list-extensions | grep ms-vscode-remote.remote-ssh
9
10 # Connect to the remote instance using VS Code command line
11 code --folder-uri vscode-remote://ssh-remote+EC2-LibreChat/home/ec2-user
12 code --folder-uri vscode-remote://ssh-remote+EC2-LibreChat/home/ec2-user
13
14 # Now that VS Code is connected to the remote instance, install Docker extension from Microsoft via the
   ↪ UI

```

```

15 # In the VS Code window that just opened (connected to EC2-LibreChat):
16 # Press Cmd+Shift+P → Type "Extensions: Install Extension" → Search "Docker" → Install
17
18 # Test direct SSH access
19 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.

```

1 # Listing: Testing AWS Bedrock model access
2 # Verifying IAM permissions and model availability
3
4 # Test Bedrock model listing from EC2 instance
5 ssh EC2-LibreChat
6
7 # Check aws cli
8 aws --version
9
10 # switch of the pager and list the available models
11 export AWS_PAGER=""
12 aws bedrock list-foundation-models --region eu-central-1
13 aws bedrock list-foundation-models --region eu-central-1 | grep modelName
14 aws bedrock list-foundation-models --region eu-central-1 | grep modelId
15 aws bedrock list-foundation-models --region eu-central-1 | grep anthropic
16 aws bedrock list-foundation-models --region eu-central-1 | grep deep
17 aws bedrock list-foundation-models --region us-east-1 | grep deep # this contains deepseek llm
18 aws bedrock list-foundation-models --region us-east-1 | grep modelId
19
20
21 # List available foundation models
22 aws bedrock list-foundation-models --region us-east-1
23
24
25 # Test Claude model availability
26 aws bedrock list-foundation-models \
27   --region us-east-1 \
28   --by-provider anthropic \
29   --query 'modelSummaries[?contains(modelId, `claude`)].[modelId,modelName]' \
30   --output table
31
32 # Test Nova model availability
33 aws bedrock list-foundation-models \
34   --region us-east-1 \
35   --by-provider amazon \
36   --query 'modelSummaries[?contains(modelId, `nova`)].[modelId,modelName]' \
37   --output table
38
39 # Test a simple model invocation
40 aws bedrock-runtime invoke-model \
41   --region us-east-1 \
42   --model-id us.anthropic.claude-3-5-haiku-20241022-v1:0 \
43   --body
44   ↪ '{"anthropic_version":"bedrock-2023-05-31","max_tokens":100,"messages":[{"role":"user","content":"Hello,
45   ↪ how are you?"]}]' \
46   --cli-binary-format raw-in-base64-out \
47   /tmp/response.json

```



```

46
47 # Display the response
48 cat /tmp/response.json | jq '.content[0].text'
49
50
51
52 # First, let's check what Claude models are actually available
53 aws bedrock list-foundation-models \
54     --region us-east-1 \
55     --by-provider anthropic \
56     --query 'modelSummaries[?contains(modelId,
↪  `claude`)].{ModelId:modelId,ModelName:modelName,Status:modelLifecycleStatus}' \
57     --output table
58
59
60 # Test different claude models
61 export MODEL_ID=us.anthropic.claude-3-7-sonnet-20250219-v1:0
62 export MODEL_ID=us.anthropic.claude-3-5-haiku-20241022-v1:0
63 export MODEL_ID=us.anthropic.claude-opus-4-20250514-v1:0
64 export MODEL_ID=us.anthropic.claude-sonnet-4-20250514-v1:0
65
66
67 # Call Claude model with correct model ID (using Claude 3.5 Sonnet)
68 aws bedrock-runtime invoke-model \
69     --region us-east-1 --model-id $MODEL_ID \
70     --cli-binary-format raw-in-base64-out output.json \
71     --body
↪  '{"anthropic_version":"bedrock-2023-05-31","max_tokens":500,"messages":[{"role":"user","content":"Wie
↪  wird das Wetter morgen in Hamburg?"}]}'
72
73     --body
↪  '{"anthropic_version":"bedrock-2023-05-31","max_tokens":500,"messages":[{"role":"user","content":"Hallo,
↪  wer bist du? Was sind deine Fähigkeiten?"}]}'
74
75 # Display the response
76 cat output.json | jq -r '.content[0].text'
77
78
79 # Call Claude Modell with more detailed body message
80 aws bedrock-runtime invoke-model \
81     --model-id anthropic.claude-3-sonnet-20240229-v1:0 \
82     --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}'}
↪  \
83     --cli-binary-format raw-in-base64-out \
84     --region eu-central-1 \
85     output.json
86 cat output.json | jq -r '.content[0].text'
87 cat output.json
88
89
90 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.

```
1  # Listing: EC2 environment preparation
2  # Setting up the foundation for LibreChat
3
4  # Connect to our instance
5  ssh EC2-LibreChat
6
7  # Update the system
8  sudo yum update -y
9
10 # install some useful tools
11 sudo yum install -y htop
12 sudo yum install -y wget
13
14
15 # Create necessary directories for persistent volumes
16 sudo mkdir -p /opt/librechat/mongodb
17 sudo mkdir -p /opt/portainer/data
18 sudo mkdir -p /opt/portainer/data/certs
19
20 # Set appropriate permissions
21 sudo chmod -R 777 /opt/portainer/data
22 sudo chmod -R 777 /opt/librechat/mongodb
23
24
25 # Exit for now - we'll return with specific tasks
26 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.

```
1  # Listing: LibreChat clone and Docker installation
2  # Getting LibreChat and preparing containerization
3
4  ssh EC2-LibreChat
```

```

5
6 # Clone the LibreChat repository from this release:
↪ https://github.com/danny-avila/LibreChat/releases/tag/v0.7.8
7 git clone --branch v0.7.8 https://github.com/danny-avila/LibreChat.git
8 cd LibreChat
9
10 # Verify the correct version is checked out
11 git describe --tags
12 git branch
13
14
15 # Install docker-compose
16 sudo curl -L "https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname
↪ -s)-$(uname -m)" -o /usr/local/bin/docker-compose
17 sudo chmod +x /usr/local/bin/docker-compose
18
19 # Verify installation
20 docker-compose --version
21
22 # Initial LibreChat preparation
23 cp .env.example .env
24
25 # Test basic docker-compose functionality
26 docker-compose up -d
27 docker ps -a
28 #docker-compose down
29
30 ## Check if LibreChat is started
31 curl http://localhost:3080
32
33
34 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 anthropic.

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

```

1 # Listing: Testing LibreChat via SSH tunnel
2
3 ssh -L 3080:localhost:3080 EC2-LibreChat
4 # open LibreChat in your browser.
5 exit

```

When the SSH tunnel is established, LibreChat can be accessed via local browser at: <http://localhost:3080> Register 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.

```

1  # Listing: Local configuration file preparation
2  # Preparing production configurations locally
3
4  cd "$PROJECT_DIR"
5  pwd
6  # Copy docker-compose override file for SSL/HTTPS deployment
7  scp "$PROJECT_DIR"/configs/docker-compose.override.yml
   ↪ ec2-user@EC2-LibreChat:~/LibreChat/docker-compose.override.yml
8
9  # Backup and copy deploy-compose.yml with SSL configuration
10 ssh EC2-LibreChat "cp ~/LibreChat/deploy-compose.yml ~/LibreChat/deploy-compose.yml.bak"
11 scp "$PROJECT_DIR"/configs/deploy-compose.yml ec2-user@EC2-LibreChat:~/LibreChat/deploy-compose.yml
12
13 # Copy NGINX configuration with SSL
14 ssh EC2-LibreChat "cp ~/LibreChat/client/nginx.conf ~/LibreChat/client/nginx.conf.bak"
15 scp "$PROJECT_DIR"/configs/nginx.conf ec2-user@EC2-LibreChat:~/LibreChat/client/nginx.conf
16
17 # Verify the differences
18 ssh EC2-LibreChat "diff ~/LibreChat/deploy-compose.yml ~/LibreChat/deploy-compose.yml.bak"
19 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.

```

1  # Listing: SSL certificate setup for HTTPS
2  # Implementing secure communications
3
4  ssh EC2-LibreChat
5
6  cd ~/LibreChat
7
8  # Create SSL directory
9  mkdir -p client/ssl
10
11 # Generate self-signed SSL certificate for HTTPS
12 openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
13     -keyout client/ssl/nginx.key \
14     -out client/ssl/nginx.crt \

```

```

15  -subj "/C=US/ST=State/L=City/O=Organization/CN=localhost"
16
17  # Generate DH parameters for enhanced security (this may take a few minutes)
18  openssl dhparam -out client/ssl/dhparam 2048
19
20  # Set proper permissions
21  chmod 644 ./client/ssl/nginx.key
22  chmod 644 ./client/ssl/nginx.crt
23  chmod 644 ./client/ssl/dhparam
24
25
26  # Verify SSL files
27  ls -la client/ssl/
28
29  # Update environment for HTTPS domains
30  cat .env | grep -iE 'DOMAIN_CLIENT|DOMAIN_SERVER'
31  sed -i 's|\\(DOMAIN_.*=\\)http://|\\1https://|' .env
32
33  # Verify HTTPS configuration
34  grep -E 'DOMAIN_CLIENT|DOMAIN_SERVER' .env
35
36  # Create librechat.yaml configuration file
37  touch librechat.yaml
38
39  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.

```

1  # Listing: Production deployment launch
2  # Starting LibreChat with full HTTPS and security
3
4  ssh EC2-LibreChat
5
6  cd ~/LibreChat
7
8  # Stop any running services
9  docker-compose -f deploy-compose.yml -f docker-compose.override.yml down
10
11  # Start production deployment with SSL. This will pull the NGINX container.
12  docker-compose -f deploy-compose.yml -f docker-compose.override.yml up -d
13
14  # Verify all containers are running
15  docker ps -a
16
17  # Check service logs for any issues
18  docker logs $(docker ps -q --filter "name=LibreChat-API")
19
20  exit
21
22  # Display our public IP for browser access
23  echo "LibreChat is now accessible at: https://$public_ip"
24

```

```

25  ## open broser to this url
26
27  open https://$public_ip
28  open http://$public_ip
29
30  ## Check the via CLicking
31
32  ssh EC2-LibreChat
33  curl https://3.69.0.104
34  curl http://3.69.0.104
35  curl http://localhost
36  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.

```

1  # Listing: AWS Bedrock Model Configuration
2  # Prerequisites: Access to the AWS CLI with IAM user
3
4  cd $PROJECT_DIR
5  pwd
6
7  ## List all aws access keys for the current user als complete json
8  aws iam list-access-keys --user-name user-lab-a --profile lab-a --output json
9
10 ## create a new AWS access key and store it in Environment Variables
11 aws iam create-access-key --user-name user-lab-a --profile lab-a-north --output json >>
12 ↪ ./aws-credentials.json
13 cat ./aws-credentials.json
14 export AWS_ACCESS_KEY_ID=$(jq -r '.AccessKey.AccessKeyId' ./aws-credentials.json)
15 export AWS_SECRET_ACCESS_KEY=$(jq -r '.AccessKey.SecretAccessKey' ./aws-credentials.json)
16 echo $AWS_ACCESS_KEY_ID
17 echo $AWS_SECRET_ACCESS_KEY # should not be printed out, but in this case I delete it in the clean-up
18 ↪ section.
19
20 ## Check if the status of the AWS access key is active
21 aws iam list-access-keys --user-name user-lab-a --profile lab-a-north --output json | jq -r
22 ↪ '.AccessKeyMetadata[] | select(.Status == "Active") | .AccessKeyId'
23
24 ## Copy .env file from the ec2 instance to the local machine
25 mkdir -p $PROJECT_DIR/LibreChat
26 ssh EC2-LibreChat "whoami; ls -la ~/LibreChat/.env.example"
27 scp ec2-user@EC2-LibreChat:~/LibreChat/.env.example $PROJECT_DIR/LibreChat/.env.example
28 ls -la LibreChat
29 cp LibreChat/.env.example LibreChat/.env
30
31 ## Update the Bedrock Config using a here document
32 cat >> $PROJECT_DIR/LibreChat/.env << EOF
33
34 #####
35 #   AWS Bedrock   #
36 #####
37
38 BEDROCK_AWS_DEFAULT_REGION=us-east-1

```

```

36 BEDROCK_AWS_ACCESS_KEY_ID=$AWS_ACCESS_KEY_ID
37 BEDROCK_AWS_SECRET_ACCESS_KEY=$AWS_SECRET_ACCESS_KEY
38
39 BEDROCK_AWS_MODELS="
40 --us-inference-profiles,
41 us.anthropic.claude-opus-4-20250514-v1:0,
42 us.anthropic.claude-sonnet-4-20250514-v1:0,
43 us.anthropic.claude-3-7-sonnet-20250219-v1:0,
44 us.deepseek.r1-v1:0,
45 us.anthropic.claude-3-5-sonnet-20241022-v2:0,
46 us.anthropic.claude-3-5-haiku-20241022-v1:0,
47 us.meta.llama3-3-70b-instruct-v1:0,
48 --us-east-1--,
49 amazon.titan-text-lite-v1,
50 amazon.nova-micro-v1:0,
51 amazon.nova-lite-v1:0,
52 amazon.nova-pro-v1:0,
53 mistral.mistral-large-2402-v1:0,
54 mistral.mistral-small-2402-v1:0,
55 "
56 EOF
57
58 # check if the env file is updated
59 tail -n 50 $PROJECT_DIR/LibreChat/.env
60
61 ## scp the env file to the instance
62 scp $PROJECT_DIR/LibreChat/.env ec2-user@EC2-LibreChat:~/LibreChat/.env
63
64 ## Login to the instance and restart the docker-compose
65 ssh EC2-LibreChat
66 whoami; pwd
67 cd ~/LibreChat
68 docker-compose -f deploy-compose.yml -f docker-compose.override.yml down
69 docker-compose -f deploy-compose.yml -f docker-compose.override.yml up -d
70 exit
71
72 ## open via public ip
73 echo $public_ip
74 open https://$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 precision 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.

```

1  # Listing: Ollama installation for local RAG processing
2  # Setting up local model inference for embeddings
3
4  # Prepare Ollama-specific Docker configuration
5  cd "$PROJECT_DIR"; pwd
6  scp "$PROJECT_DIR"/configs/docker-compose.override.yml.ollama
   ↪ ec2-user@EC2-LibreChat:~/LibreChat/docker-compose.override.yml
7
8  # Deploy with Ollama integration
9  ssh EC2-LibreChat
10 cd ~/LibreChat
11 docker-compose -f deploy-compose.yml -f docker-compose.override.yml down
12 docker-compose -f deploy-compose.yml -f docker-compose.override.yml up -d
13 docker ps -a
14
15
16 # Install embedding model
17 docker exec -it $(docker ps -q --filter "name=ollama") bash
18 # verify that we are inside the container
19 whoami; hostname; pwd; uname -a
20 ollama --version
21 ollama pull nomic-embed-text
22 ollama list
23 exit # docke-container
24 exit # ec2-instance
25
26 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.

```

1  # Listing: RAG environment setup
2  # Configuring LibreChat for local RAG processing
3
4  ssh EC2-LibreChat
5  cd ~/LibreChat

```



```

6
7 # Add Ollama RAG configuration to environment
8 cat >> .env << EOF
9
10 #####
11 #   Ollama RAG   #
12 #####
13 # Use Ollama for embeddings
14 RAG_API_URL=http://host.docker.internal:8000
15 EMBEDDINGS_PROVIDER=ollama
16 OLLAMA_BASE_URL=http://host.docker.internal:11434
17 EMBEDDINGS_MODEL=nomic-embed-text
18
19 EOF
20
21 # Verify configuration
22 tail -n 10 .env
23
24 # Test Ollama API connectivity
25 docker exec -it $(docker ps -q --filter "name=rag_api") sh
26 whoami; hostname; pwd; uname -a
27 curl http://ollama:11434/api/version
28
29 ## Check the embeddings api
30 curl http://ollama:11434/api/embeddings -d '{
31     "model": "nomic-embed-text",
32     "prompt": "The sky is blue because of Rayleigh scattering"
33 }'
34
35 exit # container
36 exit # ec2-instance
37
38 # copy the librechat.yaml to the ec2 instance
39 scp $PROJECT_DIR/configs/librechat_ollama.yaml ec2-user@EC2-LibreChat:~/LibreChat/librechat.yaml
40
41 ssh EC2-LibreChat
42 cd ~/LibreChat
43
44 # Restart services with RAG configuration
45 docker-compose -f deploy-compose.yml -f docker-compose.override.yml down
46 docker-compose -f deploy-compose.yml -f docker-compose.override.yml up -d
47
48 ## It may be necessary to check the firewall settings on the EC2 instance to allow traffic on port
49 ↪ 11434.
50 # Check if port 11434 is allowed
51 sudo iptables -L -n | grep 11434
52
53 # Allow traffic if needed
54 sudo iptables -A INPUT -p tcp --dport 11434 -j ACCEPT
55
56 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 tested with a sample text.

6.1.3 Demonstration of RAG

Now we select the model nova pro and start a new chat. First I ask who Andreas Wittmann it at it usually 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 Installing Ollama Models for Inference

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

```

1  # Listing: Advanced model deployment
2  # Installing additional models for various use cases
3
4  ssh EC2-LibreChat
5
6  # Access Ollama container for model management
7  docker exec -it $(docker ps -q --filter "name=ollama") bash
8
9  # Install additional models for various use cases
10 ollama pull deepseek-r1:8b
11 ollama pull allenporter/xlam:7b # Tool-capable model for RAG
12 ollama pull mistral-nemo
13
14 # List all available models
15 ollama list
16
17 exit
18 # Test model functionality
19 curl -X POST http://localhost:11434/api/generate \
20     -H "Content-Type: application/json" \
21     -d '{
22         "model": "deepseek-r1:8b",
23         "prompt": "What is the capital of Spain?",
24         "max_tokens": 50
25     }'
26
27
28
29 # Test model functionality
30 curl -X POST http://localhost:11434/api/generate \
31     -H "Content-Type: application/json" \
32     -d '{
33         "model": "mistral-nemo",
34         "prompt": "What is the capital of Spain?",
35         "max_tokens": 50
36     }'
37
38
39 # Restart services to integrate new models
40 cd ~/LibreChat
41 docker-compose -f deploy-compose.yml -f docker-compose.override.yml down
42 docker-compose -f deploy-compose.yml -f docker-compose.override.yml up -d
43
44 exit

```

Result: Three models 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: Using Ollama for inference.

- 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 too long or provoke an error. Inference 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.

```

1  # Listing: Using terraform to change the instance type
2  # Prerequisites: We start from the local machine
3  exit
4  whoami; pwd;
5  cd $PROJECT_DIR
6  cd terraform
7  pwd; ls -la
8
9  # Use a perl oneliner to change the instance type in the terraform.tfvars file to g4dn.xlarge, only
   ↪ print to stdout
10 perl -pe 's/instance_type = ".*"/instance_type = "t3.medium"/' terraform.tfvars
11 perl -pe 's/instance_type = ".*"/instance_type = "g4dn.xlarge"/' terraform.tfvars
12 perl -pe 's/instance_type = ".*"/instance_type = "g6e.xlarge"/' terraform.tfvars
13
14 ## And change the file
15 perl -pi -e 's/instance_type = ".*"/instance_type = "t3.medium"/' terraform.tfvars
16 perl -pi -e 's/instance_type = ".*"/instance_type = "g4dn.xlarge"/' terraform.tfvars # GPU-Mem
   ↪ 16 GiB
17 perl -pi -e 's/instance_type = ".*"/instance_type = "g6e.xlarge"/' terraform.tfvars # GPU-Mem
   ↪ 4x24 GiB
18

```

```

19 cat terraform.tfvars
20
21 # Create a new plan
22 terraform plan
23 terraform plan -out=tfplan -var-file=terraform.tfvars
24 terraform show -json tfplan > tfplan.json
25 cat tfplan.json | jq > tfplan.pretty.json
26 cat tfplan.json | jq '.resource_changes[] | {address: .address, action: .change.actions[0]}'
27
28 # Apply the new plan
29 terraform apply
30 yes

```

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

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.

```

1  # Listing: Installing NVIDIA Drivers and CUDA for GPU Support
2
3  ssh EC2-LibreChat
4
5  # Show machine details, like CPU, GPU, RAM
6  lscpu
7  lspci | grep -i nvidia
8
9
10 ## Extra Configuration for GPU usage
11 sudo su
12
13 # Install NVIDIA drivers and CUDA
14 # Update system packages
15 dnf update -y
16
17
18 ##### this works!!!! [2025-06-01 Sun 12:31]
19 # Install required tools
20 sudo dnf install -y gcc kernel-devel-$(uname -r) make
21
22 # Install NVIDIA drivers through AWS package manager
23 sudo dnf config-manager --add-repo
24 ↪ https://developer.download.nvidia.com/compute/cuda/repos/rhel9/x86_64/cuda-rhel9.repo
25
26 sudo dnf clean all
27 sudo dnf -y module install nvidia-driver:latest-dkms
28
29 # reboot to activate the NVIDIA drivers
30 shutdown -r now
31 ssh EC2-LibreChat
32
33 # Check driver installation
34 nvidia-smi # NVIDIA-SMI has failed because it couldn't communicate with the NVIDIA driver

```

```

35
36 ### Troubleshooting
37 #Fixing NVIDIA Driver Symbol Error on Amazon Linux 2023
38 # 1. Remove existing NVIDIA packages
39 sudo dnf remove -y '*nvidia*' '*cuda*'
40
41 # 2. Install kernel headers that match exactly
42 sudo dnf install -y kernel-devel-$(uname -r) kernel-headers-$(uname -r)
43
44 # 3. Install required packages
45 sudo dnf install -y gcc make dkms
46
47 # 4. Install DRM kernels (missing dependency)
48 sudo dnf install -y kernel-modules-extra
49
50 # 5. Reinstall NVIDIA drivers using Amazon-specific method
51 sudo dnf config-manager --add-repo
52 ↪ https://developer.download.nvidia.com/compute/cuda/repos/rhel9/x86_64/cuda-rhel9.repo
53 sudo dnf clean all
54 sudo dnf -y module install nvidia-driver:latest-dkms
55
56 # 6. Reboot to load the driver
57 sudo reboot
58 ssh EC2-LibreChat
59
60
61
62 #-----
63
64 # Load modules
65 sudo modprobe nvidia
66 sudo modprobe nvidia_uvm
67
68
69 # Download the CUDA installer for Amazon Linux 2023
70 cd /tmp
71 wget
72 ↪ https://developer.download.nvidia.com/compute/cuda/12.2.0/local_installers/cuda_12.2.0_535.54.03_linux.run
73
74 # Add execute permissions
75 chmod +x cuda_12.2.0_535.54.03_linux.run
76
77 # Create a new temporary directory and Use this directory for the installation
78 mkdir ~/cuda_tmp
79 # Run the installer (silent mode with custom options)
80 sudo TMPDIR=~/cuda_tmp sh cuda_12.2.0_535.54.03_linux.run --silent --override --toolkit --samples
81 ↪ --toolkitpath=/usr/local/cuda-12.2 --samplespath=/usr/local/cuda --no-opengl-libs
82
83 # Set as default CUDA version
84 sudo ln -s /usr/local/cuda-12.2 /usr/local/cuda
85
86 ## Check CUDA installation
87 /usr/local/cuda/bin/nvcc --version
88
89 # Check if libraries exist
90 ls -l /usr/local/cuda/lib64
91

```

```

92 # Install NVIDIA Container Toolkit if not already installed
93 sudo dnf install -y nvidia-container-toolkit
94 sudo nvidia-ctk runtime configure --runtime=docker
95 sudo systemctl restart docker
96
97 # Test GPU support in Docker
98 docker run --rm --gpus all nvidia/cuda:12.2.0-base-ubuntu22.04 nvidia-smi
99
100 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.

```

1 # Listing: Updating Docker Configuration for Ollama with GPU Support
2
3
4 # Update Ollama-specific Docker configuration
5 cd "$PROJECT_DIR"; pwd
6 scp "$PROJECT_DIR"/configs/docker-compose.override.yml ollama_gpu
  ↪ ec2-user@EC2-LibreChat:~/LibreChat/docker-compose.override.yml
7
8
9 ## Login to container and restart docker-compose
10 ssh EC2-LibreChat
11 whoami; pwd
12 cd ~/LibreChat
13 docker ps -a
14 docker-compose -f deploy-compose.yml -f docker-compose.override.yml down
15 docker-compose -f deploy-compose.yml -f docker-compose.override.yml up -d
16
17 # Verify Ollama is Using GPU
18 # Check running containers
19 docker ps
20
21 # Check GPU usage by running nvidia-smi
22 nvidia-smi
23
24 # Check logs from Ollama container
25 docker logs $(docker ps -q --filter name=ollama)
26
27 exit
28 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 answers are ok.

7.0.5 Monitoring GPU usage

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

```

1  # Listing: Install monitoring tools for GPU usage
2
3  ssh EC2-LibreChat
4  whoami; pwd
5
6  ## Simple command to check GPU usage
7  nvidia-smi -l 1 # This will refresh every second and run indefinitely until stopped with Ctrl+C
8
9
10 ##### Install pip if not already installed
11 sudo dnf install -y python3-pip
12
13 ### Install gpustat
14 pip3 install gpustat
15
16 # Run gpustat
17 gpustat
18
19
20 ##### Install nvitop
21 pip install nvitop --user
22 nvitop
23 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 Ollama for inference

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 agentic 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:Q5KM](#) This is an instruct model that is quantized from the llama 3 model.

`ollama run eramax/salesforce-iterative-llama3-8b-dpo-r:Q5KM`

```

1  # Listing: Configuring Ollama for inference
2  # Prerequisites: EC2-LibreChat is started and ollama container is running.
3
4  PROJECT_DIR=~/.LocalProjects/ai-bootcamp/private-ai/

```

```

5  cd $PROJECT_DIR
6  pwd; whoami
7  ssh EC2-LibreChat
8  whoami; pwd
9  cd ~/LibreChat
10
11  ## Check if the ollama container is running
12  docker ps -a
13  docker ps -q --filter "name=ollama"
14  docker exec -it $(docker ps -q --filter "name=ollama") bash
15  ollama --version
16  ollama list
17  # pull other models
18
19  ollama pull deepseek-r1:8b
20  ollama pull deepseek-r1:14b
21  ollama pull allenporter/xlam:7b
22  ollama pull mistral-nemo # 7GB
23  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 knowledge base. the answers are mostly nonsens.
24  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.
25  ollama pull open-orca-platypus2 # 26 GB Answers are a bit confused but very fast on the g6.12xlarge
    ↪ instance type.
26  ollama rm open-orca-platypus2 # 26 GB Answers are a bit confused but very fast on the g6.12xlarge
    ↪ instance type.
27  ollama pull qwq:32b-q8_0 # 35GB
28  #ollama pull qwq:32b # 20GB
29  #ollama pull qwq:32b-preview-q4_K_M # 20GB
30  #ollama pull qwq:32b-preview-q8_0 # 35GB
31
32  ollama rm mistral-small3.1:24b # 15GB tool use. Fast on single requests, but produces nonsens in
    ↪ RAG
33  ollama rm qwq:32b # 20GB Quite fast and impressive in chat but fails in RAG
34  ollama rm qwq:32b-q8_0 # 35GB, inference take ca. 4 minutes. Probably to load it into memory.
    ↪ Second call is fast.
35
36  ollama rm llama3:70b-instruct-q4_K_M
37
38  exit # container
39
40  # Check the models by querying the API from the commandline
41  curl -X POST http://localhost:11434/api/generate \
42    -H "Content-Type: application/json" \
43    -d '{
44      "model": "deepseek-r1:8b",
45      "prompt": "What is your name?",
46      "max_tokens": 50
47    }'
48    "prompt": "What is the capital of France?",
49
50  # Check the models by querying the API from the commandline
51  curl -X POST http://localhost:11434/api/generate \
52    -H "Content-Type: application/json" \
53    -d '{
54      "model": "allenporter/xlam:7b",
55      "prompt": "What is the capital of France?",
56      "max_tokens": 50
57    }'
58

```



```

59 # Check the models by querying the API from the commandline
60 curl -X POST http://localhost:11434/api/generate \
61     -H "Content-Type: application/json" \
62     -d '{
63         "model": "eramax/salesforce-iterative-llama3-8b-dpo-r:Q5_K_M",
64         "prompt": "What is the capital of France?",
65         "max_tokens": 50
66     }'
67
68 # Check the models by querying the API from the commandline
69 curl -X POST http://localhost:11434/api/generate \
70     -H "Content-Type: application/json" \
71     -d '{
72         "model": "qwq:32b-q8_0",
73         "prompt": "What is the capital of Germany?",
74         "max_tokens": 50
75     }'
76
77 # Check the models by querying the API from the commandline
78 curl -X POST http://localhost:11434/api/generate \
79     -H "Content-Type: application/json" \
80     -d '{
81         "model": "qwq:32b",
82         "prompt": "What is the capital of Sweden?",
83         "max_tokens": 50
84     }'
85
86 # Check the models by querying the API from the commandline
87 curl -X POST http://localhost:11434/api/generate \
88     -H "Content-Type: application/json" \
89     -d '{
90         "model": "mistral-small3.1:24b",
91         "prompt": "What is the capital of Spain?",
92         "max_tokens": 50
93     }'
94
95
96 # Check the models by querying the API from the commandline # requires 13,1 GiB GPU memory
97 curl -X POST http://localhost:11434/api/generate \
98     -H "Content-Type: application/json" \
99     -d '{
100         "model": "llama3.3:70b-instruct-q2_K",
101         "prompt": "What is the capital of Spain?",
102         "max_tokens": 50
103     }'
104
105 # Check the models by querying the API from the commandline
106 curl -X POST http://localhost:11434/api/generate \
107     -H "Content-Type: application/json" \
108     -d '{
109         "model": "open-orca-platypus2",
110         "prompt": "What is the capital of Spain?",
111         "max_tokens": 50
112     }'
113
114 exit
115 ##
116
117 ## Configure these models in librechat via VSCode.
118

```

```

119  ## restart the docker-compose
120  cd ~/LibreChat
121  ls -la
122
123  docker-compose -f deploy-compose.yml -f docker-compose.override.yml down
124  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_KM could be loaded but failed to provide an answer.

Result: The g6.12xlarge instance type could be started without changing the driver or CUDA installation. The overall performance was excellent. 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.

```

1  # Listing: Destroy the AWS resources and clean update
2
3  whoami; pwd;
4  cd $PROJECT_DIR
5  cd terraform
6  pwd; ls -la
7
8  ## destroy the instance
9  terraform destroy
10 yes
11
12
13
14 ## delete the aws keys and clean up
15 echo $AWS_ACCESS_KEY_ID
16 aws iam delete-access-key --user-name user-lab-a --access-key-id $AWS_ACCESS_KEY_ID --profile lab-a
17 # clean up
18 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.