Финансирано от Европейския съюз

СледващоПоколениеЕС

План за възстановяване и устойчивост

Република България

OLLAMA

Техническо описание

Версия 1 / 27.04.2025 г.

Съдържание

1. Въведение

Ollama е софтуер с отворен код, разработен от компанията Мета (Meta) и позволяващ локалното използване на големи езикови и многомодални модели. Ollama предлага опростен интерфейс на командния ред (command line interface – CLI), чрез който може лесно да се стартира модел за инференция (генериране и обработка на текст, създаване на изображения, преобразуване на реч в текст и др.), както и да бъде извършена фина настройка на модели с цел по-тясна специализация. Ollama разполага с богата библиотека от свободно достъпни модели, сред които последни версии на Llama, Mistral, Phi и др. Софтуерът може да работи и с модели извън библиотеката на Ollama.

Ollama е алтернатива на използването на модели, достъпни чрез комерсиален приложно-програмен интерфейс (application programming interface – API). Софтуерът използва наличните ресурси на машината, на която е инсталиран, като позволява работа както локално, така и в облачни инсталации, без необходимост от заплащане. Инсталирането и подготовката на Ollama са направени по начин, който позволява бързо използване. Локалната инсталация и контролът на средата, в която работи Ollama, дава възможност на потребителите да управляват сигурността и поверителността на използваните модели и данни. Софтуерът може да бъде инсталиран на всички популярни операционни системи – Windows, macOS, много от дистрибуциите на Linux – както и в Docker контейнер със съществуващия Docker имидж. В същото време това не ограничава използването на инструмента спрямо системата, на която е инсталиран, тъй като Ollama предлага и API, с който може да бъдат достъпени функциите на софтуера от друга среда, хост или контейнер. Това позволява разделението на средата, в която работи Ollama, и различните приложения, които могат да ползват обща инсталация и модели.

2. Инсталация

Възможни са различни подходи за инсталиране на системата Ollama според вида на операционната система и хардуерната конфигурация.

2.1. Линукс (Linux)

Системни изисквания и препоръки за инсталиране и използване на Ollama в Линукс:

1. 64-битова система;
2. RAM памет според размера на моделите, които ще бъдат използвани: препоръчва се 8GB за модели до 3B, 16GB за модели до 7B, 32GB за модели с до 13B параметъра;
3. 4 GB свободно пространство за инсталираното приложение;
4. Допълнително свободно пространство за изтегляне на модели – в зависимост от големината на моделите.
5. Може да се инсталира на повечето дистрибуции на Линукс, например Debian, Ubuntu и др.

6. Функциите на Ollama могат да бъдат ускорени с използването на графични карти Nvidia или AMD:
    6.1. Nvidia карти и драйвери, съвместими с CUDA toolkit, за системи с графични карти Nvidia (https://developer.nvidia.com/cuda-gpus);
    6.2. AMD Radeon драйвери за системи с графични карти AMD (https://ollama.com/blog/amd-preview);

Има два основни начина за инсталиране на Ollama на системи с Линукс – автоматично и ръчно.

2.1.1. Автоматична инсталация

За автоматична инсталация в операционната система Линукс обикновено се използва командата:

```
curl -fsSL https://ollama.com/install.sh | sh
```

Командата curl изтегля скрипта install.sh, в случая със следните опции:

- -f – прекратява изпълнението, ако сървърът отговори с код за грешка;
- -s – потиска показването на подробна информация (тих режим);
- -S – запазва възможността за показване на информация за възникнали грешки;
- -L – позволява следване на пренасочванията от сървъра към друг URL.

След изтегляне на файла install.sh, той се предава чрез оператора | (конвейер, pipe) към интерпретатора sh за изпълнение. Инсталационният скрипт извършва следните действия:

- Проверява системните изисквания и хардуерната конфигурация:
    - Процесорна архитектура – AMD64 или Arm64;
    - Налични графични карти – Nvidia (Cuda) или AMD Radeon (ROCm);
- Изтегля подходяща версия на Ollama;
- Инсталира Ollama в системна директория (обикновено /usr/local/bin);
- Конфигурира системни услуги като добавяне на Ollama като системна услуга и нейните настройки;
- Инсталира допълнителни софтуерни компоненти за GPU изпълнение, когато са налични графични карти Nvidia или AMD.

2.1.2. Ръчна инсталация

Като алтернатива на автоматичната инсталация може да се изтегли архив с Ollama, специфичен за системната конфигурация:

```
curl -L https://ollama.com/download/ollama-linux-amd64.tgz -o ollama-linux-amd64.tgz
```

В дадения пример се изтегля версия на Ollama за AMD64 процесорна архитектура. След като се изтегли, се разархивира в системна директория:

```
sudo tar -C /usr -xzf ollama-linux-amd64.tgz
```

За по-стабилна работа обикновено се препоръчва добавяне на Ollama като системна услуга. За целта се създава потребител и група ollama:

```
sudo useradd -r -s /bin/false -U -m -d /usr/share/ollama ollama
```

Тук се използват опциите -r за създаване на системен потребител, -s /bin/false за предотвратяване на стартиране на системния интерпретатор с този потребител, -U за създаване на група със същото име като потребителя, -m за създаване на потребителска директория с име, зададено с опцията -d. След това е необходимо да се добави текущият потребител към групата ollama:

```
sudo usermod -a -G ollama $(whoami)
```

Тези команди са важни за сигурността и правилното функциониране на Ollama като системна услуга, като се осигурява изолация на процесите и управление на правата за достъп.
Създава се файл /etc/systemd/system/ollama.service за конфигурация на systemd услуга със следното съдържание

```
[Unit]
Description=Ollama Service
After=network-online.target

[Service]
ExecStart=/usr/bin/ollama serve
User=ollama
Group=ollama
Restart=always
RestartSec=3

[Install]
WantedBy=default.target
```

Тази конфигурация определя стартиране на услугата след мрежовите услуги, изпълнение на командата ollama serve за стартиране на ollama сървъра, работа като потребител и група ollama, автоматично рестартиране при неуспех с 3 секунди пауза между опитите, инсталиране като част от default.target услуга, за да се стартира автоматично при зареждане на системата. За активиране на услугата default.target се използва програмата systemctl:

```
sudo systemctl daemon-reload
```

```
sudo systemctl enable ollama
```

2.2. Уиндоус (Windows)

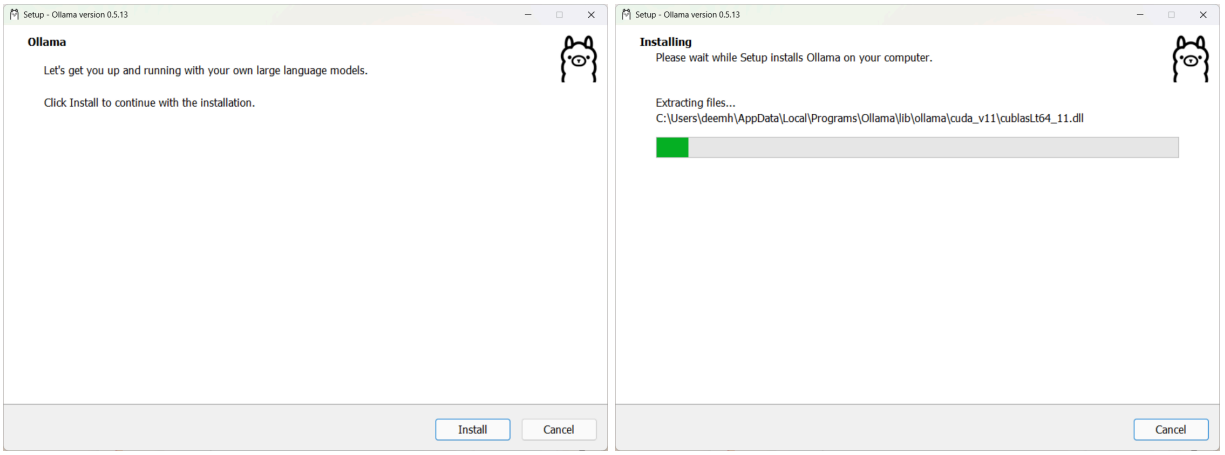
2.2.1. Инсталация

Системните изисквания за инсталиране и използване на Ollama в Уиндоус са:

- Windows 10 22H2 или Windows 11 (Home или Pro);
- Nvidia 452.39 или по-нови драйвери за системи с графични карти Nvidia;
- AMD Radeon драйвери за системи с графични карти AMD;
- 4 GB свободно пространство за инсталираното приложение;
- Допълнително свободно пространство за изтегляне на модели, в зависимост от големината на моделите.

Актуалните изисквания могат да бъдат намерени на следния адрес в GitHub: https://github.com/ollama/ollama/blob/main/docs/windows.md

Инсталацията на Ollama за Уиндоус е лесна – чрез стартиране на инсталационния файл OllamaSetup.exe, който може да бъде изтеглен от уебсайта на приложението. Файлът е опростен графичен инсталатор, в който не се задават допълнителни настройки, с два екрана – начален, с който да се стартира инсталацията, и екран за проследяване на процеса. След успешно инсталиране на Ollama, инсталаторът се затваря автоматично.



По подразбиране Ollama се инсталира в директорията на потребителя, като за това не са необходими администраторски права. Ollama може да бъде инсталирана и в друга директория чрез стартиране на инсталатора от команден ред с параметър /DIR за желаната директория:

```
OllamaSetup.exe /DIR="c:/path/to/dir"
```

2.2.2. Деинсталация

Тъй като инсталаторът на Ollama регистрира приложение за деинсталация, Ollama може да бъде деинсталирана от списъка с инсталирани приложения в Настройки (Settings), секция Приложения (Apps), бутон Инсталирани приложения (Installed apps). Деинсталацията на Ollama ще премахне и изтеглените модели.

2.3. макОС (macOS)

2.3.1. Инсталация

Подобно на инсталацията на Уиндоус, инсталацията на Ollama на макОС е сравнително лесна. От страницата за сваляне на Ollama (https://ollama.com/download/mac) се сваля архивът Ollama-Darwin.zip. С двойно кликване на архива той се разархивира и създава Ollama.app, който се мести в системната папка Applications. Оттам отново с двойно кликване се стартира инсталаторът на Ollama. Три последователни екрана на инсталатора представят:

1. начален екран с бутон Next
2. екран с бутон за започване на инсталацията
3. финален екран с инструкции за базов тест за Ollama от командния ред

След инсталация Ollama може да бъде използвана от командния ред без допълнителни стъпки.

2.3.2. Използване

След инсталиране Ollama може да бъде използвана директно от командния ред без необходимост от стартиране на Ollama сървъра чрез ollama serve.

2.4. Докър (Docker) (за Линукс)

Докър представлява платформа за организиране на процеса на изграждане, внедряване и управление на софтуерни приложения. Платформата позволява пакетирането на дадено приложение и неговите зависимости в контейнер, който след това може да се изпълнява на различни системи.

За използване на Ollama в Докър контейнери съществува Докър имидж ollama/ollama (https://hub.docker.com/r/ollama/ollama). Стартирането на контейнера и наличния на хост системата софтуер зависи от използваните ресурси.

2.4.1. CPU

За използване на Ollama само със наличния процесор (CPU) контейнерът се стартира със следната команда без допълнителни изисквания и настройки:

```
docker run -d -v ollama:/root/.ollama -p 11434:11434 --name ollama ollama/ollama
```

2.4.2. GPU – NVIDIA

За да може Ollama да използва една или повече NVIDIA графични карти, ако такива са налични и поддържат Cuda интерфейс (https://developer.nvidia.com/cuda-gpus), е необходимо да бъде инсталиран предварително софтуерът на NVIDIA за поддръжка на контейнери NVIDIA Container Toolkit. NVIDIA Container Toolkit може да бъде инсталиран от пакетните мениджъри Apt, Yum или Dnf в зависимост от версията на Линукс, инсталирана на хост системата.

2.4.2.1. Инсталиране на NVIDIA Container Toolkit с Apt

За инсталиране на NVIDIA Container Toolkit чрез Apt е необходимо първо да се добави хранилището за софтуерния пакет чрез изпълнение на следните команди от командния ред:

```
curl -fsSL https://nvidia.github.io/libnvidia-container/gpgkey | sudo gpg --dearmor -o /usr/share/keyrings/nvidia-container-toolkit-keyring.gpg
```

```
curl -s -L https://nvidia.github.io/libnvidia-container/stable/deb/nvidia-container-toolkit.list | sed 's#deb https://#deb [signed-by=/usr/share/keyrings/nvidia-container-toolkit-keyring.gpg] https://#g' | sudo tee /etc/apt/sources.list.d/nvidia-container-toolkit.list
```

```
sudo apt-get update
```

След това се инсталира софтуерът NVIDIA Container Toolkit:

```
sudo apt-get install -y nvidia-container-toolkit
```

2.4.2.2. Инсталиране на NVIDIA Container Toolkit с Yum/Dnf

За инсталиране на NVIDIA Container Toolkit чрез Yum или Dnf е необходимо първо да се добави хранилището за софтуерния пакет чрез изпълнение на следната команда от командния ред:

```
curl -s -L https://nvidia.github.io/libnvidia-container/stable/rpm/nvidia-container-toolkit.repo | sudo tee /etc/yum.repos.d/nvidia-container-toolkit.repo
```

След това се инсталира софтуерът NVIDIA Container Toolkit:

```
sudo yum install -y nvidia-container-toolkit
```

За инсталиране с пакетния мениджър Dnf се заменя в командата “yum” с “dnf”.

2.4.2.3. Конфигуриране на Докър и стартиране на контейнер

Необходимо е да се конфигурира Докър, за да използва драйверите на NVIDIA.

```
sudo nvidia-ctk runtime configure --runtime=docker
```

```
sudo systemctl restart docker
```

След това контейнерът може да бъде стартиран.

```
docker run -d --gpus=all -v ollama:/root/.ollama -p 11434:11434 --name ollama ollama/ollama
```

2.4.3. GPU – AMD

За стартиране на контейнер с имиджа ollama/ollama, който да може да използва една или повече AMD Radeon графични карти, ако такива са налични, не е необходимо инсталиране на допълнителен софтуер на хост системата. Контейнерът се стартира със следната команда:

```
docker run -d --device /dev/kfd --device /dev/dri -v ollama:/root/.ollama -p 11434:11434 --name ollama ollama/ollama:rocm
```

2.4.4. Използване на Ollama в Докър

Ollama, работеща в Докър контейнер, може да бъде използвана чрез изпращане на команда към контейнера чрез docker exec. По-долу е даден пример за стартиране на модела Llama 3.2 чрез изпълнение на командата ollama run llama3.2 в Докър контейнера ollama:

```
docker exec -it ollama ollama run llama3.2
```

В този пример:

- docker exec – изпълнява зададена команда в нов процес в контейнер
- -it – комбинация на опция -i за интерактивно изпълнение (проддържа отворен канала за вход stdin) и опция -t за отваряне на виртуален терминал
- ollama – името на контейнера, в който да бъде изпълнена командата
- ollama run llama3.2 – командата, която да бъде изпълнена

3. Работа с модели

3.1. Интерфейс на командния ред (CLI) на Ollama

3.1.1. Стартиране на сървъра на Ollama

```
ollama serve
```

```
ollama start
```

Двете команди са еквивалентни. Софтуерът Ollama работи, като създава сървър на локалната машина (localhost), достъпен по подразбиране чрез IP адреса 127.0.0.1 на порт 11434.

Командата стартира Ollama сървъра, който е необходим за изпълнението на всички други команди. След стартиране изпълнението на командата не приключва, а терминалът се използва за изписване на логовете от Ollama сървъра.

При операционната система Уиндоус сървърът може да бъде стартиран и от десктоп приложението на Ollama. Приложението няма графичен прозорец, който да се отваря, а вместо това се създава системна икона в таскбара. Менюто на системната икона позволява

спирането на сървъра, както и лесен достъп до папката, в която са записани лог файловете на Ollama.

Пример:

```
ollama serve
```

```
2025/03/22 16:50:35 routes.go:1225: INFO server config env="map[CUDA_VISIBLE_DEVICES:
GPU_DEVICE_ORDINAL: HIP_VISIBLE_DEVICES: HSA_OVERRIDE_GFX_VERSION: HTTPS_PROXY:
HTTP_PROXY: NO_PROXY: OLLAMA_CONTEXT_LENGTH:2048 OLLAMA_DEBUG:false
OLLAMA_FLASH_ATTENTION:false OLLAMA_GPU_OVERHEAD:0 OLLAMA_HOST:http://127.0.0.1:11434
OLLAMA_INTEL_GPU:false OLLAMA_KEEP_ALIVE:5m0s OLLAMA_KV_CACHE_TYPE:
OLLAMA_LLM_LIBRARY: OLLAMA_LOAD_TIMEOUT:5m0s OLLAMA_MAX_LOADED_MODELS:0
OLLAMA_MAX_QUEUE:512 OLLAMA_MODELS:/home/deemhristov/.ollama/models
OLLAMA_MULTIUSER_CACHE:false OLLAMA_NEW_ENGINE:false OLLAMA_NOHISTORY:false
OLLAMA_NOPRUNE:false OLLAMA_NUM_PARALLEL:0 OLLAMA_ORIGINS:[http://localhost
https://localhost http://localhost:* https://localhost:* http://127.0.0.1 https://127.0.0.1 http://127.0.0.1:*
https://127.0.0.1:* http://0.0.0.0 https://0.0.0.0 http://0.0.0.0:* https://0.0.0.0:* app://* file://* tauri://*
vscode-webview://* vscode-file://*] OLLAMA_SCHED_SPREAD:false ROCR_VISIBLE_DEVICES: http_proxy:
https_proxy: no_proxy:]"
time=2025-03-22T16:50:35.217+02:00 level=INFO source=images.go:432 msg="total blobs: 20"
time=2025-03-22T16:50:35.218+02:00 level=INFO source=images.go:439 msg="total unused blobs removed: 0"
time=2025-03-22T16:50:35.220+02:00 level=INFO source=routes.go:1292 msg="Listening on 127.0.0.1:11434
(version 0.6.0)"
time=2025-03-22T16:50:35.220+02:00 level=INFO source=gpu.go:217 msg="looking for compatible GPUs"
time=2025-03-22T16:50:38.014+02:00 level=INFO source=gpu.go:377 msg="no compatible GPUs were
discovered"
time=2025-03-22T16:50:38.014+02:00 level=INFO source=types.go:130 msg="inference compute" id=0
library=cpu variant="" compute="" driver=0.0 name="" total="7.4 GiB" available="6.9 GiB"
...
```

3.1.2. Стартиране на модел

```
ollama run <модел> [<промпт>] [<опции>]
```

Параметри:

- <модел> – името на модел от библиотеката на Ollama, локален модел (създаден чрез ollama create) или път до GGUF модел (директория или URL)
- <промпт> – директно подаване на промпт към модела

Опции:

- --format <параметър> – формат на отговора, напр. “json” или дефиниция на JSON схема (JSON Schema)
- --insecure – използване на локация (регистър, хранилище) с некриптирана връзка за изтегляне на модела
- --keepalive <параметър> – задава за колко време да бъде зареден моделът в паметта, стойност по подразбиране 5m (5 минути)

- --nowordwrap – позволява пренасяне на нов ред в резултата между всеки два символа вместо само между отделни думи
- --verbose – принтира статистически данни след края на генерирания отговор

Командата изпълнява следните стъпки:

1. Изтегля модела от библиотеката на ollama или зададената локация (например HuggingFace).
2. Зарежда модела в паметта (RAM и/или VRAM).
3. Стартира модела.
    a. Ако не е зададен промпт, моделът се стартира в интерактивен "чат" формат, при който потребителят може да "разговаря" с модела чрез поредица от промптове и отговори, които се пазят като контекст за всеки следващ промпт.
    b. Ако е зададен промпт, се генерира директно отговор и изпълнението на командата приключва.
4. След приключване на командата (прекъсване на чат сесията или генериране на отговор към предварително зададен промпт), моделът остава зареден в паметта за определено време (5 минути по подразбиране, може да бъде променено чрез опция --keepalive).

По време на чат сесия, когато командата е стартирана без предварително зададен промпт, могат да бъдат използвани следните команди:

- /set – задаване на променливи за сесията
    - /set parameter … – задаване на стойност на параметър
    - /set system <съобщение> – задаване на системното съобщение
    - /set history – включване на историята (моделът "помни" предишните промптове и отговори)
    - /set nohistory – изключване на историята
    - /set wordwrap – пренасяне на нов ред само на цели думи
    - /set nowordwrap – пренасяне на нов ред на символи
    - /set format json – задаване на формат JSON за отговорите
    - /set noformat – премахване на изискването за форматиране на отговорите
    - /set verbose – показване на статистика след генерираните отговори
    - /set quiet – без показване на допълнителна информация
- /show – показване на информация за модела
    - /show info – обща информация за модела
    - /show license – лиценза на модела
    - /show modelfile – принтиране на конфигурацията от modelfile файла на модела
    - /show parameters – показване на стойностите на параметрите
    - /show system – показване на системното съобщение
    - /show template – показване на шаблона за промпт
- /load <модел> – зареждане на модел или сесия
- /save <модел> – запазване на настоящата сесия
- /clear – изтриване на контекста на сесията
- /bye – прекратяване на сесията и приключване на изпълнението на ollama run

- /?, /help – помощ на наличните команди
- /? shortcuts – показване на наличните клавишни комбинации

Командата ollama run позволява и използването на многомодални модели (текст и изображение), като се използва промт във формат “<текст> [<изображения>]”, където <изображения> е един или повече пътя във файловата система към файлове на изображения. Този формат може да бъде използван както за промпта, директно зададен на командата като параметър, така и в чат сесия.

Отделен случай е използването на командата ollama run за зареждане на модел в паметта без изпълнение на инференция. Това се прави, като се зададе параметър празен промпт ("").

Примери за използване на ollama run:

```
ollama run mistral "Hello! Be quick." --verbose
```

```
Hello! Of course, here to help. What can I assist you with today?

total duration:       3.453232646s
load duration:        29.785339ms
prompt eval count:    10 token(s)
prompt eval duration: 803ms
prompt eval rate:     12.45 tokens/s
eval count:           18 token(s)
eval duration:        2.615s
eval rate:            6.88 tokens/s
```

```
ollama run mistral:7b-instruct-q3_K_M
```

```
pulling manifest
pulling 0b5c278cdb26... 100% ▕▕▕▕ 3.5 GB
pulling 43070e2d4e53... 100% ▕▕▕▕  11 KB
pulling 22e1b2e8dc2f... 100% ▕▕▕▕  43 B
pulling ed11eda7790d... 100% ▕▕▕▕  30 B
pulling 4296b21b20e8... 100% ▕▕▕▕ 485 B
verifying sha256 digest
writing manifest
success
>>> What languages do you speak?
I can communicate in multiple languages including English,
Spanish, French, German, Italian, Portuguese, and Russian.
However, my ability to understand and generate human language
is still limited compared to a human's.

>>> /show info
  Model
    architecture        llama
    parameters          7.2B
    context length      32768
```

```
  embedding length    4096
  quantization        Q3_K_M

Parameters
  stop    "[INST]"
  stop    "[/INST]"

License
  Apache License
  Version 2.0, January 2004

>>> Send a message (/? for help)
```

3.1.3. Създаване на модел

```
ollama create <модел> [<опции>]
```

Параметри:

- <модел> – зададено име за създадения модел, което може да бъде използвано впоследствие за други команди, напр. за стартиране на модела с ollama run

Опции:

- -f <modelfile> или --file <modelfile> – път до файл от тип Ollama Modelfile (вж. 3.4. Кастомизация), в който са описани основният модел, параметри и инструкции за използване на модела; по подразбиране е файл с име Modelfile в настоящата директория
- -q <ниво> или --quantize <ниво> – квантизиране на модела до зададеното ниво (напр. q4_0)

Командата създава локален модел, определен от базовия модел и параметрите, зададени в modelfile файла. Впоследствие този модел може да бъде използван за други команди.

3.1.4. Изтегляне на модел

```
ollama pull <модел> [<опции>]
```

Параметри:

- <модел> – името на модела от библиотеката на Ollama или от друга локация в мрежата, който да бъде изтеглен или обновен локално

Опции:

- --insecure – използване на локация (регистър, хранилище) с некриптирана връзка за изтегляне на модела

Командата изтегля модела, без да го зарежда в паметта и без да го стартира. Това ускорява действието на ollama run, тъй като няма да е необходимо изтеглянето на модела като първа стъпка от нейното изпълнение.

3.1.5. Изтриване на модел

```
ollama rm <модел>
```

Параметри:

- <модел> – името на локалния модел, който да бъде изтрит

След изпълнение на командата изтритият модел няма да бъде наличен локално и трябва да бъде отново създаден или изтеглен, за да може да бъде използван. Резултатът може да бъде потвърден с командата ollama list или ollama ls.

3.1.6. Копиране на модел

```
ollama cp <съществуващ-модел> <нов-модел>
```

Параметри:

- <съществуващ-модел> – името на локалния модел, който да бъде копиран
- <нов-модел> – името, с което да бъде записано копието на модела

Командата създава нов модел, идентичен на изходния с различно име. Новият модел е наличен локално.

3.1.7. Показване на информация за модел

```
ollama show <модел> [<опции>]
```

Параметри:

- <model_name> – името на локалния модел, за който да бъде показана информацията

Опции:

- --license – лицензa на модела
- --modelfile – принтиране на конфигурацията от modelfile файла на модела
- --parameters – показване на стойностите на параметрите
- --system – показване на системното съобщение
- --template – показване на шаблона за промпт

Без зададени опции ollama info показва обща информация за модела. При зададени опции се принтира определената от опцията информация. Командата е еквивалентна на командата /show в чат сесия.

Пример:

```
ollama show llama3.2 --parameters
```

```
stop                           "<|start_header_id|>"
stop                           "<|end_header_id|>"
stop                           "<|eot_id|>"
```

3.1.8. Изброяване на локалните модели

```
ollama list
```

```
ollama ls
```

Двете команди са еквивалентни. Изброява наличните локално модели. Резултатът е форматиран като таблица с колони име (NAME), идентификатор (ID), размер (SIZE) и време на последна промяна (MODIFIED).

Пример:

```
ollama ls
```

```
NAME                        ID              SIZE      MODIFIED
llama3.2:3b-instruct-fp16     195a8c01d91e    6.4 GB    30 hours ago
mistral:7b-instruct-q3_K_M    9fdddabfd5d    3.5 GB    2 days ago
llama3.2:latest             a80c4f17acd5    2.0 GB    2 days ago
mistral:latest              f974a74358d6    4.1 GB    4 days ago
llava:latest                8dd30f6b0cb1    4.7 GB    7 days ago
small-llama-3.2:latest        baf6a787fdff    1.3 GB    7 days ago
llama3.2:1B                 baf6a787fdff    1.3 GB    7 days ago
```

3.1.9. Изброяване на заредените модели

```
ollama ps
```

Командата изброява заредените в паметта модели. Резултатът е форматиран като таблица с колони име (NAME), идентификатор (ID), размер (SIZE), ресурс (PROCESSOR) и време до премахване на модела от паметта (UNTIL).

Пример:

```
ollama ps
```

```
NAME            ID              SIZE      PROCESSOR    UNTIL
mistral:latest    f974a74358d6    5.9 GB    100% CPU     4 minutes from now
```

3.1.10. Спиране на зареден модел

```
ollama stop <модел>
```

Параметри:

- <модел> – името на заредения (стартирания) модел

Командата премахва зададения модел от паметта. Резултатът може да бъде потвърден чрез командата ollama ps.

3.2. Кратко описание на достъпните модели по групи

В момента на писането на това техническо описание библиотеката на Ollama предлага 162 модела за изтегляне и използване в различни размери и нива на квантизация. Сред моделите са отделени и три групи, които са допълнително или специфично тренирани за определени задачи: Embedding (създаване на ембединги на думи, изречения и текст), Vision (работа с изображения в допълнение на текст), Tools (способност за работа с различни външни инструменти). Ollama поддържа и хранилища за модели, създадени от потребителите.

3.2.1. Текстови модели

Библиотеката на Ollama предлага някои от най-популярните отворени големи езикови модели с различни версии и домейни, редовно обновявани. Като се изключат многомодалните модели, сред изцяло текстовите семействата модели са:

1. Meta Llama
    - Llama 2 (llama2)
    - Llama 3 (llama3)
    - Llama 3.1 (llama3.1) – голям модел с до 405 милиарда параметъра
    - Llama 3.2 (llama3.2) – много малък модел с до 3 милиарда параметъра
    - Llama 3.3 (llama3.3)
    - Различни модели от други потребители и организации, базирани на Llama моделите
2. Microsoft Phi
    - Phi-2 (phi)
    - Phi-3 (phi3)
    - Phi-3.5-mini (phi3.5)
    - Phi-4 (phi4)
    - Phi-4-mini-instruct (phi4-mini)
    - Различни модели от други потребители и организации, базирани на Phi моделите
3. Mistral
    - Mistral 7B v0.3 (mistral)
    - Mistral NeMo (mistral-nemo) – модел с голям контекст до 128 хиляди токъна
    - Mistral Large 2 (mistral-large) – голям модел със 123 милиарда параметъра със силни възможности за логическо мислене, познание и програмен код

- Mistral Small 3 (mistral-small) – малък модел с 24 милиарда параметъра, създаден за бързина и възможност за работа с ограничени хардуерни ресурси
- Mixtral (mixtral) – Mixture-of-Experts модел
- Codestral (codestral) – първи модел на Mistral с фокус върху програмен код
- MathΣtral (mathstral) – модел с логическо мислене с фокус върху математически задачи
- Различни модели от други потребители и организации, базирани на Mistral моделите

4. DeepSeek
   - DeepSeek-R1 (deepseek-r1) – модел с логическо мислене
   - DeepSeek LLM (deepseek-llm) – оригиналният DeepSeek модел
   - DeepSeek Coder (deepseek-coder) – модел с фокус програмен код
   - DeepSeek-V2 (deepseek-v2) – Mixture-of-Experts модел
   - DeepSeek-Coder-V2 (deepseek-coder-v2) – Mixture-of-Experts модел с фокус програмен код
   - DeepSeek-V2.5 (deepseek-v2.5) – интеграция на DeepSeek-V2 и DeepSeek-Coder-V2
   - DeepSeek-V3 (deepseek-v3) – Mixture-of-Experts модел
5. Google DeepMind Gemma
   - Gemma (gemma)
   - Gemma 2 (gemma2)
   - CodeGemma (codegemma) – модел с фокус върху програмен код

3.2.2. Многомодални модели

В библиотеката на Ollama са налични и многомодални големи езикови модели с възможност за обработка на текст и изображения.

1. Meta Llama 3.2 Vision (llama3.2-vision)
2. Mistral Small 3.1 (mistral-small3.1)
3. Google DeepMind Gemma 3 (gemma3)
4. LLaVA (llava)
5. Други малки модели и модели, създадени чрез фина настройка

3.2.3. Модели за ембединги

Това са големи езикови модели, създадени специално за целите на генериране на ембединги на текст с различна дължина – думи, фрази, изречения, параграфи и големи документи. Тези ембединги могат впоследствие да се използват за различни задачи в сферата на обработката на естествен език (natural language processing – NLP). Пример за подобно приложение е описан в глава 4. Генериране, подкрепено с извличане (Retrieval-Augmented Generation – RAG)

3.3. Персонализация

Ollama позволява на потребителите да адаптират съществуващи модели за специфични нужди или да създават нови модели с уникални характеристики. При персонализацията на модел Ollama създава „допълнителни слоеве“, които не променят директно структурата и

тегловните параметри на базовия модел (дефиниран напр. чрез съответен GGUF формат). Вместо това допълнителните слоеве функционират като метаданни и инструкции, определящи как моделът да бъде използван и конфигуриран. Такъв подход дава възможност за създаване на множество версии на един и същ базов модел с минимално допълнително използване на дисково пространство.

Допълнителните слоеве включват: шаблон за форматиране на входния текст, системни инструкции за контекст и поведение, параметри за настройка на изпълнението, обща конфигурация и информация за лицензи. Основният метод за персонализация на модел в Ollama е чрез създаване на конфигурационен файл (т. нар. Modelfile), който съдържа тези инструкции за конфигуриране.

3.3.1. Създаване на Modelfile

Файловият тип Modelfile представлява текстов файл с инструкции на всеки ред. Възможните инструкции са:

- FROM <име-или-път-на-модел> – задължителна първа инструкция, чрез която се определя базовият модел, върху който ще се изгради профилираният модел
- PARAMETER <параметър> <стойност> – задава глобални параметри на модела
    - temperature – определя пропорционално „креативността“ на езиковия модел със стойности между 0 и 1, като влияе върху разпределението на вероятностите при генериране на изходните данни
    - num_ctx – размер на контекстовия прозорец
    - num_predict – максималният брой токъни, които да бъдат предвидени (по подразбиране -2 – неограничен)
    - repeat_last_n – размер на прозореца на повтарящ се текст
    - repeat_penalty – контролира наказанието за повторения
    - top_k – ограничава броя на вариантите за избор на следваща лексема
    - top_p – ограничава вариантите за следваща лексема по вероятност
    - min_p – ограничава вариантите за следваща лексема по вероятност, дефинирана относително към лексемата с най-голяма вероятност
    - seed – задава сийда за генериране на случайни числа – определянето на този параметър позволява консистентното генериране на един и същи резултат за един и същи промпт, като се осигурява сигурно репродуциране на резултатите
    - stop – добавя „стоп“ низ, до който ще приключи генерирането на резултат; могат да бъдат добавени множество стоп низове с отделни команди
    - mirostat – включване на алгоритъма Mirostat за определяне на следващата лексема (0 – изключено, 1 – Mirostat, 2 – Mirostat 2.0)
    - mirostat_eta – настройка на скоростта на „обучение“ – колко бързо се учи от обратната връзка от генерираните текстове
    - mirostat_tau – настройка на несигурността (perplexity) за алгоритъма Mirostat, по-високо число за по-разнообразен резултат, по-малко – за по-сигурен
- TEMPLATE """<шаблон>""" – дефинира шаблона, чрез който да бъде форматиран промптът преди подаване към модела; шаблонът следва синтаксиса на езика за шаблони Go (https://pkg.go.dev/text/template) и може да включва:

    - системните инструкции {{ .System }} – дефинирани чрез команда SYSTEM
    - промпта {{ .Prompt }} – задължително
    - мястото на резултата {{ .Response }} – последващ текст е игнориран при генериране
- SYSTEM """<съобщение>""" – дефинира системните инструкции, които ще определят поведението на модела
- ADAPTER <път-към-адаптер> – задаване на LoRA адаптер, създаден чрез фина настройка към базовия модел, определен чрез команда FROM; може да бъде Safetensor или GGUF формат
- LICENSE """<лиценз>"""– позволява определяне на лицензa за използване и споделяне на настоящия Modelfile; няма влияние върху изпълнението на модела и другите параметри
- MESSAGE <роля> <текст> – позволява задаването на история на интеракция между потербител и модела чрез серия от команди
    - system – алтернативен начин за дефиниране на системното съобщение
    - user – въпрос на потребител
    - assistant – отговор на модел

Пример за файл от тип Makefile с име custom_assist (съкратен):

```
FROM phi4
PARAMETER temperature 0.7
SYSTEM """
Вие сте специализиран асистент за обучение по информатика. При взаимодействие с потребителите:
1. Предоставяйте точни и актуални обяснения ...
...
Вашата цел е да предизвиквате интерес към информатиката ...
"""
```

3.3.2. Използване на профилиран модел

За да бъде използван модел, профилиран чрез Modelfile, той трябва да бъде създаден локално чрез командата ollama create (вж. 3.1.3. Създаване на модел). За създаване на модел с примерния Modelfile custom_assist:

```
ollama create phi4-informatics-assistant -f ./custom_assist
```

```
gathering model components
using existing layer sha256:fd7b6731c33c57f61767612f56517460ec2d1e2e5a3f0163e0eb3d8d8cb5df20
using existing layer sha256:32695b892af87ef8fca6e13a1a31c67c1441d7398be037e366e2fc763857c06a
using existing layer sha256:fa8235e5b48faca34e3ca98cf4f694ef08bd216d28b58071a1f85b1d50cb814d
creating new layer sha256:af30b989a0ffa7268ddaa223e1bf4e5edc24837bdddf882efb3d9bb24bf20bd2
creating new layer sha256:7313aaee19e08d6e42936e44e6a5eea3746566ba2084206e2251581cfbf5e516
writing manifest
success
```

След изпълнението на ollama create, моделът може да се види в списъка на локалните модели с ollama list или ollama ls:

```
ollama ls
```

```
NAME                                ID              SIZE      MODIFIED
phi4-informatics-assistant:latest    2d30936e677f    9.1 GB    About a minute ago
...
```

След създаването на модела той може да се използва както всеки друг модел чрез ollama run, като бъде дадено само неговото име без път към файл.

4. RAG

Съществуват няколко начина да бъде добавена нова информация, която моделът да използва като контекст за генерирането на отговора си. Най-известният метод е фината настройка, при която предварително трениран голям езиков модел допълнително се тренира на по-малко множество от данни, най-често с информация от специфична сфера или със задачи от конкретен характер или вид, така че да се разширят знанията и способностите на модела за конкретна задача или цел.

Друг популярен начин на допълване на информацията на модела за генериране на релевантен и фактически верен отговор е генерирането, подпомогнато с извличане (retrieval-augmented generation – RAG). RAG използва допълване на заявката на потребителя с релевантна за задачата информация, поради което се отнася към групата методи, наречени инженерство на промптове (prompt engineering). Чрез RAG даден модел може да използва допълнително знания, които са зададени в сбор от документи, като контекст за генериране на отговор.

4.1. Какво е RAG?

Като метод за инженерство на промптове, вместо допълнителната информация за сферата или задачата, RAG допълва заявката на потребителя с информация от свързани документи. Действията като част от RAG се разделят на две части – конфигуриране и операция.

Конфигуриране на RAG системата:

1. Подготовка на модела, алгоритъма, интерфейса и хранилището за вграждания (embeddings)
2. Текстова подготовка на документите
3. Генериране и съхранение на вграждания (embeddings)

Допълнителни документи могат да бъдат добавени и на по-късен етап, като в тези случаи се повтарят стъпки 2 и 3.

Операция:

1. Въвеждане на промпт от потребителя
2. Създаване на вграждания за промпта и откриване на най-близките вграждания от документите
3. Извличане на информацията (текста) от документите с откритите вграждания
4. Допълване на промпта на потребителя с извлечената от документите информация, която да послужи за контекст при генериране на отговора
5. Задаване на допълнения промпт като вход на модела
6. Генериране на отговор от модела

Следващите точки ще опишат горните стъпки чрез примери.

4.2. Имплементация чрез Python, Ollama и LangChain

Един сравнително лесен начин за създаване на RAG приложение е чрез езика Python и библиотеката LangChain в допълнение на Ollama.

Библиотеката с отворен код LangChain цели лесна и бърза разработка на приложения, използващи големи езикови модели, като предоставя стандартен програмен интерфейс за създаване на вериги (последователност от действия, т.нар. на английски chains), множество интеграции с други инструменти (като Ollama, Chroma DB за съхранение на вграждания, и др.), както и пълни вериги за по-често срещани приложения. Библиотеката LangChain е налична и за JavaScript.

4.2.1. Подготовка на средата

Този пример за създаване на приложение предполага, че на работната машина е инсталиран Python 3.7 и pip за управление на инсталирани Python пакети.

Подготовка на Ollama с модел Llama 3.1:

1. Инсталация (вж. 2. Инсталация)
2. Стартиране на сървъра с командата ollama serve (вж. 3.1.1. Стартиране на сървъра на Ollama)
3. Изтегляне на моделите за вграждания и инференция с командите ollama pull nomic-embed-text и ollama pull llama3.1 (вж. 3.1.4. Изтегляне на модел)

Инсталиране на пакетите на LangChain за примера с използване на Ollama за работа с модели:

```
pip install langchain langchain_community scikit-learn langchain-ollama
```

4.2.2. Подготовка на данни

LangChain поддържа множество интеграции с външни инструменти за зареждане на различни формати документи. Библиотеката поддържа зареждане и извличане на текстова и визуална (изображения) информация от текстови файлове, онлайн съдържание, PDF, Microsoft Open

Office XML и множество други. В този пример се използва WebBasedLoader за зареждане на документи, зададени чрез уеб адреси (URL):

```
from langchain_community.document_loaders import WebBaseLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Списък с уеб адреси, от които да бъдат заредени документите
urls = [
    "https://ifgpt.dcl.bas.bg/",
    "https://ifgpt.dcl.bas.bg/objectives/",
    "https://ifgpt.dcl.bas.bg/results/",
]

# Зареждане на документи от адресите
docs = [WebBaseLoader(url).load() for url in urls]
docs_list = [item for sublist in docs for item in sublist]
```

След като бъдат свалени, документите трябва да бъдат разделени на по-малки части. Това ще направи процеса на търсене и извличане по-ефикасен. Разделянето ще бъде извършено чрез сплитера RecursiveCharacterTextSplitter, който да направи парчета от по 250 символа:

```
# Инициализация на сплитера с определения размер (250) и застъпване (0) на парчетата
text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=250, chunk_overlap=0
)

# Разделяна на документите на парчета
doc_splits = text_splitter.split_documents(docs_list)
```

Документите вече са подготвени за следващата стъпка.

4.2.3. Вграждания и тяхното съхранение

За да може търсенето и извличането да са бързи, ефективни и базирани на семантичната информация в текста, те се извършват чрез създаване и сравняване на т.нар. вграждания (embeddings). Вгражданията са числова репрезентация на информацията във вид на вектори от предварително определена размерност, като по-близките семантично текстове имат по-близки вграждания във векторното пространство и обратното. След създаването на вгражданията за документите, те се пазят в хранилище за вектори (vector store), откъдето впоследствие се извличат.

В този пример е използван моделът за вграждане nomic-embed-text от библиотеката на Ollama. За съхранение на вгражданията е използван SKLearnVectorStore от scikit-learn. Инструментът за извличане е настроен да извлича 4-те най-релевантни (близки) документа за зададената заявка.

```
from langchain_community.vectorstores import SKLearnVectorStore
from langchain_ollama import OllamaEmbeddings

# Създаване и съхранение на вгражданията за документите
vectorstore = SKLearnVectorStore.from_documents(
    documents=doc_splits,
    embedding=OllamaEmbeddings(model="nomic-embed-text"),
)
retriever = vectorstore.as_retriever(k=4)
```

4.2.4. Подготовка на модела

След подготовката на документите и техните вграждания е ред на модела, който ще изпълнява заявките. Първа стъпка е създаването на шаблон за заявката към модела, в който ще бъдат допълнени началната заявка на потребителя и информацията от извлечените документи, за да бъде извършено генерирането, подпомогнато с извличане.

```
from langchain_ollama import ChatOllama
from langchain.prompts import PromptTemplate
from langchain_core.output_parsers import StrOutputParser

# Дефиниране на шаблона за заявка към големия езиков модел
prompt = PromptTemplate(
    template="""Ти си асистент, който отговаря на въпроси.
    Използвай зададените документи за генериране на отговор.
    Ако не знаеш отговора, просто кажи, че не знаеш.
    Отговаряй кратко, като използваш не повече от 3 изречения.
    Въпрос: {question}
    Документи: {documents}
    Отговор:
    """,
    input_variables=["question", "documents"],
)
```

След това се конфигурира връзката към модела през Ollama чрез класа ChatOllama. Тук се задава и самият модел, който да бъде използван – в този случай Llama 3.1. На параметъра temperature е зададена стойност 0, за да се генерират консистентни отговори.

```
# Инициализация на Llama 3.1 като използвания в приложението модел
llm = ChatOllama(
    model="llama3.1",
    temperature=0,
)
```

Накрая се създава веригата от LangChain инструменти, чрез които да се извърши създаването на допълнената заявка, генерирането и окончателната обработка на резултата чрез StrOutputParser, за да бъде подходящ за представяне на потребителя. Създаването на верига

става чрез свързване на отделните компоненти с предефинирания от LangChain оператор | (конвейер или pipe).

```
# Създаване на верига, която комбинира шаблона за заявка и големия езиков модел
rag_chain = prompt | llm | StrOutputParser()
```

4.2.5. Използване в приложение за RAG

Дотук бяха подготвени документите и техните вграждания в хранилище, както и беше дефинирана LangChain веригата за обработка на допълнената заявка. За да бъдат комбинирани за целта на генерацията, подпомогната с извличане, е необходимо да бъде създадено приложение, което да ги използва.

Приложението е дефинирано като клас RAGApplication, който се инициализира с вече дефинираните компоненти – инструмент за извличане на релевантни документи чрез вграждания и верига за допълване и изпълнение на потребителската заявка чрез избрания голям езиков модел. Класът има метод run, приемащ като параметър потребителската заявка и връщащ генерирания след пълния RAG процес резултат.

```
# Дефиниране на приложението за RAG като клас
class RAGApplication:
    def __init__(self, retriever, rag_chain):
        self.retriever = retriever
        self.rag_chain = rag_chain

    def run(self, question):
        # Извличане на релевантните документи
        documents = self.retriever.invoke(question)

        # Комбиниране на текста от документите
        doc_texts = "\\n".join([doc.page_content for doc in documents])

        # Генериране на отговора на модела с параметри потребителската заявка и
        # извлечената от документите информация
        answer = self.rag_chain.invoke({"question": question,
                                        "documents": doc_texts})

        return answer

# Инициализиране на приложението
rag_application = RAGApplication(retriever, rag_chain)
```

Следва да бъде получена заявката от потребителя, за да може да бъде използвана в приложението.

```
# Получаване на заявката от потребителя
question = input("Въпрос: ")
```

```
# Генериране и показване на отговор
answer = rag_application.run(question)
print("Отговор:", answer)
```

```
Въпрос: Какво е IfGPT?
Отговор: IfGPT е проект за подобряване на свободно достъпните големи езикови модели и чатмодели по отношение отразяване на българския език и култура.
```

4.2.6. Разширения на приложението

Така описаното приложение е много статично – изисква промяна в кода за добавяне на документи, обработва документите при всяко изпълнение и приема само една заявка, без да може да се зададе история на предишни „разговори“ с модела. За да може да се използва за задача, по-близка до честите приложения на RAG, е необходимо:

1. Позволяване на добавяне, премахване и обновяване на документи и постоянното съхранение на вгражданията:

Използваният в примера клас SKLearnVectorStore позволява запазването на базата данни с вграждания в постоянната памет (на диск), като при инациализация бъде зададен параметърът persist_path (път към файла, от който да бъде извлечена и в който да бъде запазена информацията). По подразбиране данните ще бъдат запазени във JSON формат, но това може да бъде променено с параметъра serializer, който поддържа стойностите json, bson и parquet. Записването на настоящите данни във файла се извършва чрез метода persist.

Класът също така има и методи за добавяне, премахване и обновяване на документи чрез методите add_document, add_texts и delete, както и техните асинхронни варианти aadd_document, aadd_texts и adelete.

2. Използване на приложението като чат със запазена история

Често са необходими допълнителни въпроси към вече зададения. Това може да се случи по няколко начина. В случая на описаното примерно приложение, историята (въпроси и отговори) може да бъде включена като част от потребителската заявка. При RAG приложение това крие риск от извличане на същите документи заради близостта на вгражданията на заявката към предишните изтеглени документи.

Алтернативно се използва цикъл за получаване на заявка, задаване на историята като контекст за модела, генериране на отговор, допълване на историята със заявката и отговора и показване на генерирания отговор на потребителя. За класа ChatOllama тази история може да бъде подадена, като се създаде списък с предишните съобщения и към него се добави допълнената нова заявка на потребителя. Например:

```
messages = [
    ("system", """Ти си асистент, който отговаря на въпроси.
    Използвай зададените документи за генериране на отговор.
    Ако не знаеш отговора, кажи, че не знаеш.
    Отговаряй кратко, като използваш не повече от 3 изречения.""")
    ("human", "Какво си ти?"),
    ("assistant", "Асистент за отговаряне на въпроси с помощта на документи"),
    ("human", """Въпрос: Какво можеш да ми кажеш за проекта IfGPT?"
    Документи: ...
    Отговор:""")
]
```

5. Списък на използваните източници

Tabatabaei, H. (13 февруари 2025 г.) Build a RAG-Powered LLM Service with Ollama & Open WebUI : A Step-by-Step Guide. *Medium*. https://medium.com/@hassan.tbt1989/build-a-rag-powered-llm-service-with-ollama-open-webui-a-step-by-step-guide-a688ec58ac97

Maurya, A. (14 май 2024 г.) Ollama: A Deep Dive into Running Large Language Models Locally(PART-1):. *Medium*. https://medium.com/@mauryaanoop3/ollama-a-deep-dive-into-running-large-language-models-locally-part-1-0a4b70b30982

(λx.x)eranga. (25 март 2024 г.) Build RAG Application Using a LLM Running on Local Computer with Ollama Llama2 and LlamaIndex. *Medium*. https://medium.com/rahasak/build-rag-application-using-a-llm-running-on-local-computer-with-ollama-and-llamaindex-97703153db20

Rodewald, G. (8 март 2024 г.) Running models with Ollama step-by-step. *Medium*. https://medium.com/@gabrielrodewald/running-models-with-ollama-step-by-step-60b6f6125807

Huynh, D. (5 декември 2023 г.) Build your own RAG and run it locally: Langchain + Ollama + Streamlit. *Medium*. https://medium.com/@vndee.huynh/build-your-own-rag-and-run-it-locally-langchain-ollama-streamlit-181d42805895

Ong, R. (5 септември 2024 г.) RAG With Llama 3.1 8B, Ollama, and Langchain: Tutorial. *DataCamp*. https://www.datacamp.com/tutorial/llama-3-1-rag

GeeksforGeeks. (6 февруари 2025 г.) Introduction to LangChain. *GeeksforGeeks*. https://www.geeksforgeeks.org/introduction-to-langchain/

Metric Coders. (29 август 2024 г.) How to Install and Run Ollama on macOS. *Metric Coders*. https://www.metriccoders.com/post/how-to-install-and-run-ollama-on-macos

ollama. (21 април 2025 г.) ollama/README.md. *GitHub*.
https://github.com/ollama/ollama/blob/main/README.md

ollama. (7 март 2025 г.) ollama/docs/linux.md. *GitHub*.
https://github.com/ollama/ollama/blob/main/docs/linux.md

LangChain, Inc. (5 януари 2025 г.) Document Loaders. *LangChain*.
https://python.langchain.com/docs/integrations/document_loaders/

LangChain, Inc. SKLearnVectorStore. *LangChain documentation*.
https://python.langchain.com/api_reference/community/vectorstores/langchain_community.vectorstores.sklearn.SKLearnVectorStore.html