Modele lokalne: AI na własnym sprzęcie

Poziom: średni **Dotyczy: Ollama 0.35, Open WebUI 0.11, Rocky Linux 9, Windows 11**

Czas: ~60 min

Model językowy nie musi działać w cudzej chmurze: można go pobrać i uruchomić na własnym serwerze, a rozmowy zostają wtedy w naszej sieci. Ta strona pokazuje, kiedy to ma sens, ile sprzętu potrzeba, jak postawić Ollamę i Open WebUI oraz na co uważamy przy licencjach. Wracamy tu przy pytaniu „czy to zmieści się na naszej karcie".

Komendy wykonujemy z CLI bash jako user z uprawnieniami sudo (Linux) albo w PowerShellu (Windows). Wersje i nazwy modeli sprawdzone 3 października 2026, a obszar zmienia się co kilka tygodni, więc przed instalacją zaglądamy do stron z sekcji Źródła.

Skróty użyte na tej stronie (14)

Skrót	Co znaczy
AI	Artificial Intelligence: sztuczna inteligencja, czyli programy naśladujące ludzkie rozumowanie
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
CLI	Command Line Interface: praca przez wpisywanie komend w terminalu
FAQ	Frequently Asked Questions: zbiór najczęściej zadawanych pytań z odpowiedziami
GB	Gigabyte: miliard bajtów, jednostka pojemności pamięci i plików, tu najczęściej rozmiar modelu albo pamięci karty graficznej
GGUF	format pliku z modelem językowym, w którym leżą jego parametry i słownik tokenów, używany przez llama.cpp i Ollamę
HTTP	Hypertext Transfer Protocol: protokół stron WWW
LVM	Logical Volume Manager: warstwa w Linuksie składająca dyski w elastyczne wolumeny
RAG	Retrieval-Augmented Generation: najpierw wyszukujemy w naszych dokumentach fragmenty pasujące do pytania, potem dopisujemy je do pytania zadawanego modelowi
RAM	Random Access Memory: pamięć operacyjna
RHEL	Red Hat Enterprise Linux: komercyjna dystrybucja Linuksa
ROCm	platforma AMD do liczenia na kartach graficznych tego producenta, odpowiednik CUDA

Skrót	Co znaczy
VRAM	Video RAM: pamięć karty graficznej, w której musi zmieścić się model, żeby liczył się na karcie
WWW	World Wide Web: strony i aplikacje otwierane w przeglądarce po HTTP albo HTTPS

Fiszka

□□ TYLKO ODCZYT

Sprawdzenie nazwy i pamięci karty graficznej: nvidia-smi

```
nvidia-smi --query-gpu=name,memory.total,memory.used --format=csv
```

Pokazuje, jaką kartę widzi sterownik i ile ma pamięci (VRAM). Ta liczba decyduje, jaki model się zmieści.

Przykład:

```
name, memory.total [MiB], memory.used [MiB]
NVIDIA GeForce RTX 4090, 24564 MiB, 1 MiB
```

□□ **VRAM** = Video RAM: pamięć karty graficznej, w której musi zmieścić się model, żeby liczył się na karcie · **nvidia-smi** = NVIDIA System Management Interface: pokazuje kartę graficzną NVIDIA, jej pamięć i zajętość oraz potwierdza, że sterownik działa

□□ ZMIENIA SYSTEM

Instalacja sterownika NVIDIA na Rocky Linux 9: dnf module install

Komendy można wykonywać całościowo jako blok tekstu. Po restarcie sprawdzamy kartę komendą z poprzedniej fiszki.

```
# aktualizacja systemu i repozytoria dodatkowe
sudo dnf upgrade --refresh -y
sudo dnf install -y dnf-plugins-core epel-release

# repozytorium NVIDIA dla RHEL 9
sudo dnf config-manager --add-repo
```

```
https://developer.download.nvidia.com/compute/cuda/repos/rhel9/x86_64/cuda-rhel9.repo

# zależności potrzebne do zbudowania modułu jądra
sudo dnf install -y kernel-headers kernel-devel gcc make dkms acpid libglvnd-glx
libglvnd-opengl libglvnd-devel pkgconf-pkg-config

# sterownik z otwartymi modułami jądra (karty Turing i nowsze)
sudo dnf module install -y nvidia-driver:open-dkms

sudo reboot
```

> Przy włączonym Secure Boot moduł jądra musi być podpisany, inaczej `nvidia-smi` zgłosi brak sterownika. Dla starszych kart zamiast `open-dkms` wybieramy strumień `latest-dkms`. Karty AMD i Intel obsługuje Ollama przez ROCm i Vulkan, ale procedurę sterownika bierzemy wtedy z dokumentacji producenta.

ROCm = platforma AMD do liczenia na kartach graficznych tego producenta, odpowiednik CUDA · **dnf** = Dandified YUM: menedżer pakietów RHEL/Rocky · **reboot** = restart systemu: zamyka usługi i uruchamia komputer od nowa

ZMIENIA SYSTEM

Instalacja Ollamy na Rocky Linux 9: skrypt instalacyjny

```
sudo dnf install -y zstd
curl -fsSL https://ollama.com/install.sh -o /tmp/ollama-install.sh
less /tmp/ollama-install.sh
sudo sh /tmp/ollama-install.sh
```

Skrypt tworzy konto systemowe `ollama` i usługę systemd. Skrypt najpierw czytamy, a dopiero potem uruchamiamy, bo wykonujemy go jako root. Na systemach z rodziny RHEL może sam dodać repozytorium NVIDIA i doinstalować sterownik, dlatego sterownik stawiamy wcześniej i sprawdzamy `nvidia-smi`.

Weryfikujemy usługę i wersję:

```
systemctl status ollama --no-pager
ollama --version
```

Przykład:

```
● ollama.service - Ollama Service
     Loaded: loaded (/etc/systemd/system/ollama.service; enabled)
     Active: active (running) since Sat 2026-10-03 09:12:41 CEST; 4s ago
ollama version is 0.35.1
```

RHEL = Red Hat Enterprise Linux: komercyjna dystrybucja Linuksa · **dnf** = Dandified YUM: menedżer pakietów RHEL/Rocky · **curl** = client URL: pobieranie i wysyłanie przez HTTP z terminala · **less** = przeglądarka tekstu z przewijaniem (nazwa to żart: less is more) · **sh** = shell: podstawowy interpreter poleceń Linuksa, tu uruchamia pobrany skrypt instalacyjny · **systemctl** = system control: sterowanie usługami systemd · **ollama** = Ollama: pobiera modele językowe i uruchamia je lokalnie, podkomendy to pull, run, list, ps, rm, create

ZMIENIA SYSTEM

Instalacja Ollamy na Windows 11: winget install

```
winget install --id Ollama.Ollama -e
```

Ollama działa po instalacji w tle jako program w zasobniku, a API słucha pod adresem `http://localhost:11434`. Modele leżą domyślnie w `%USERPROFILE%\.ollama\models`, logi w `%LOCALAPPDATA%\Ollama`. Dokumentacja wymaga sterownika NVIDIA w wersji 551.61 lub nowszej i Windows 10 22H2 albo nowszego. Zmienne środowiskowe ustawiamy w Ustawieniach Windows dla konta użytkownika, potem restartujemy Ollamę.

> Na stacji roboczej zostawiamy adres domyślny, czyli tylko `localhost`. Nie ustawiamy tam `OLLAMA_HOST`, bo API Ollamy nie ma uwierzytelniania.

API = Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą · **winget** = Windows Package Manager: menedżer pakietów do instalacji programów w Windows

ZMIENIA SYSTEM

Przeniesienie modeli na osobny wolumen i wyłączenie funkcji chmurowych: systemctl edit

Modele zajmują dziesiątki gigabajtów, a domyślny katalog `/usr/share/ollama/.ollama/models` leży na dysku systemowym. Zgodnie z konwencją LVM tworzymy `vgModele/lvModele` zamontowany jako `/modele` (procedura na stronie Instalacja Rocky Linux z osobnymi grupami wolumenów), a potem:

```
sudo mkdir -p /modele/ollama
sudo chown -R ollama:ollama /modele/ollama
sudo systemctl edit ollama
```

W edytorze wklejamy zmianę usługi:

```
[Service]
Environment="OLLAMA_HOST=0.0.0.0:11434"
Environment="OLLAMA_MODELS=/modele/ollama"
Environment="OLLAMA_KEEP_ALIVE=30m"
Environment="OLLAMA_NO_CLOUD=1"
```

```
sudo systemctl restart ollama
```

Znaczenie zmiennych: `OLLAMA_HOST` otwiera API dla sieci (domyślnie tylko `127.0.0.1`), `OLLAMA_MODELS` wskazuje katalog z modelami, `OLLAMA_KEEP_ALIVE` trzyma model w pamięci 30 minut zamiast domyślnych 5, a `OLLAMA_NO_CLOUD=1` wyłącza modele chmurowe i wyszukiwanie w sieci.

LVM = Logical Volume Manager: warstwa w Linuksie składająca dyski w elastyczne wolumeny · **API** = Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą · **mkdir** = make directory: utwórz katalog · **chown** = change owner: zmień właściciela · **systemctl** = system control: sterowanie usługami systemd

ZMIENIA SYSTEM

Ograniczenie dostępu do API Ollamy do jednego serwera: firewall-cmd

```
sudo firewall-cmd --permanent --add-rich-rule='rule family="ipv4" source
address="AdresIPSerweraAplikacji" port port="11434" protocol="tcp" accept'
sudo firewall-cmd --reload
```

(gdzie `AdresIPSerweraAplikacji` to adres serwera z Open WebUI, u nas `10.20.10.42`)

Po zmianach należy przeładować firewallD. Port 11434 nie jest otwarty w strefie domyślnej, więc dostaje go tylko ten jeden adres.

Przykład:

```
rule family="ipv4" source address="10.20.10.42" port port="11434" protocol="tcp" accept
```

API = Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą · **firewall-cmd** = sterowanie zaporą firewalld

TYLKO ODCZYT

Pobranie modelu: ollama pull

```
ollama pull NazwaModelu:Rozmiar
```

(gdzie `NazwaModelu:Rozmiar` to nazwa z biblioteki Ollamy, na przykład `gemma4:12b`)

Przykład:

```
pulling manifest
pulling 9f3a1c0b7d42... 100% ▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕ 7.9 GB
verifying sha256 digest
success
```

ollama = Ollama: pobiera modele językowe i uruchamia je lokalnie, podkomendy to pull, run, list, ps, rm, create

TYLKO ODCZYT

Rozmowa z modelem w terminalu: ollama run

```
ollama run NazwaModelu:Rozmiar --verbose
```

Pierwsza odpowiedź trwa dłużej, bo model ładuje się do pamięci. Przełącznik `--verbose` dopisuje po odpowiedzi statystyki, z których bierzemy liczbę tokenów na sekundę. Rozmowę kończymy wpisując `/bye`.

Przykład:

```
>>> Napisz jedno zdanie o kopiach zapasowych.
Kopia zapasowa ma wartość dopiero wtedy, gdy raz na jakiś czas sprawdzamy, że da się ją
odtworzyć.

total duration:       3.4s
load duration:        2.1s
eval count:           31 token(s)
eval rate:            58.7 tokens/s
```

□□ **ollama** = Ollama: pobiera modele językowe i uruchamia je lokalnie, podkomendy to pull, run, list, ps, rm, create

□□ TYLKO ODCZYT

Sprawdzenie, co jest załadowane i gdzie liczy: ollama ps

```
ollama ps
```

Kolumna `PROCESSOR` mówi, czy model siedzi w całości na karcie.

Przykład:

```
NAME          ID            SIZE     PROCESSOR          UNTIL
gemma4:12b    a1b2c3d4e5f6  9.1 GB   100% GPU           29 minutes from now
qwen3:32b     b7c8d9e0f1a2  24 GB    38%/62% CPU/GPU    4 minutes from now
```

Drugi wiersz to model za duży na kartę: część warstw liczy procesor i odpowiedzi są wyraźnie wolniejsze.

□□ **ollama** = Ollama: pobiera modele językowe i uruchamia je lokalnie, podkomendy to pull, run, list, ps, rm, create

□□ TYLKO ODCZYT

Sprawdzenie pobranych modeli: ollama list

```
ollama list
```

□□ **ollama** = Ollama: pobiera modele językowe i uruchamia je lokalnie, podkomendy to pull, run, list, ps, rm, create

□□ OSTROŻNIE · NIEODWRACALNE

Usunięcie modelu: ollama rm

```
ollama rm NazwaModelu:Rozmiar
```

□□ **ollama** = Ollama: pobiera modele językowe i uruchamia je lokalnie, podkomendy to pull, run, list, ps, rm, create

TYLKO ODCZYT

Zapytanie przez API HTTP: /api/chat

```
curl http://localhost:11434/api/chat -d '{
  "model": "NazwaModelu:Rozmiar",
  "messages": [{"role": "user", "content": "TrescPytania"}],
  "stream": false,
  "options": {"num_ctx": 8192}
}'
```

Odpowiedź czytamy z pola `message.content`. Parametr `num_ctx` ustawia okno kontekstowe w tokenach: domyślnie to 4096, więc dłuższy wklejony tekst bez tego parametru zostaje po cichu przycięty. Lista modeli dostępnych w API: `curl http://AdresSerweraOllamy:11434/api/tags`. Ta sama usługa odpowiada też w formacie zgodnym z OpenAI pod ścieżką `/v1`, więc wiele gotowych narzędzi podłączamy zmianą jednego adresu.

Przykład:

```
{"model":"gemma4:12b","message":{"role":"assistant","content":"Dzień dobry. W czym możemy pomóc?"},"done":true}
```

API = Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą · **HTTP** = Hypertext Transfer Protocol: protokół stron WWW · **curl** = client URL: pobieranie i wysyłanie przez HTTP z terminala

ZMIENIA SYSTEM

Import własnego pliku GGUF do Ollamy: ollama create

```
cat > Modelfile <<'EOF'
FROM /modele/gguf/NazwaPliku.gguf
EOF
ollama create NazwaWlasnegoModelu -f Modelfile
```

Ollama nie kwantyzuje plików przy imporcie, więc plik przygotowujemy wcześniej narzędziem `llama-quantize` z llama.cpp. Dla modeli rozmownych przydaje się też w `Modelfile` szablon rozmowy zgodny z modelem, bo bez niego odpowiedzi bywają bezładne.

GGUF = format pliku z modelem językowym, w którym leżą jego parametry i słownik tokenów, używany przez llama.cpp i Ollamę · **cat** = concatenate: sklej i wypisz pliki · **ollama** = Ollama: pobiera modele językowe i uruchamia je lokalnie, podkomendy to pull, run, list, ps, rm, create

TYLKO ODCZYT

Uruchomienie serwera llama.cpp z plikiem GGUF: llama-server

```
llama-server -m /modele/gguf/NazwaPliku.gguf --host 127.0.0.1 --port 8080 -c 8192 -ngl 99
```

(gdzie `-c` to okno kontekstowe w tokenach, a `-ngl` to liczba warstw liczonych na karcie graficznej, wartość 99 oznacza wszystkie)

Serwer wystawia API zgodne z OpenAI pod `http://127.0.0.1:8080/v1`. Przełączniki sprawdzamy w `llama-server --help` dla zainstalowanej wersji, bo projekt wydaje nowe wersje kilka razy w tygodniu.

GGUF = format pliku z modelem językowym, w którym leżą jego parametry i słownik tokenów, używany przez llama.cpp i Ollamę · **API** = Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą · **llama-server** = serwer z projektu llama.cpp: uruchamia plik GGUF i udostępnia go przez API HTTP zgodne z OpenAI

ZMIENIA SYSTEM

Instalacja Open WebUI na docker01, katalogi i klucz: mkdir

> W naszym środowisku dane aplikacji trzymamy w `/docker/apps/open-webui`, a plik Compose w `/docker/compose/open-webui/compose.yaml`. To konwencja, nie wymóg Dockera: można ją zmienić, byle w całej firmie była jedna. Katalog `/docker` to osobny wolumen `vgDocker/lvDocker`, żeby pełny dysk aplikacji nie zatrzymał systemu.

```
sudo mkdir -p /docker/apps/open-webui/data /docker/compose/open-webui
echo "WEBUI_SECRET_KEY=$(openssl rand -hex 32)" | sudo tee /docker/compose/open-webui/.env
sudo chmod 600 /docker/compose/open-webui/.env
```

Klucz w pliku `.env` jest stały, więc restart kontenera nie wylogowuje wszystkich użytkowników.

mkdir = make directory: utwórz katalog · **echo** = od angielskiego echo: wypisuje podany tekst na standardowe wyjście, często łączone z przekierowaniem do pliku · **chmod** = change mode: zmień prawa dostępu

TYLKO ODCZYT

Instalacja Open WebUI na docker01, plik Compose: compose.yaml

```
sudo vim /docker/compose/open-webui/compose.yaml
```

```
services:
  open-webui:
    image: ghcr.io/open-webui/open-webui:v0.11.4
    container_name: open-webui
    restart: unless-stopped
    env_file: .env
    environment:
      OLLAMA_BASE_URL: http://10.20.10.45:11434
      WEBUI_URL: https://ai.zielona.example
      DEFAULT_USER_ROLE: pending
    volumes:
      - /docker/apps/open-webui/data:/app/backend/data
    networks:
      - proxy
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.open-webui.rule=Host(`ai.zielona.example`)"
      - "traefik.http.routers.open-webui.entrypoints=websecure"
      - "traefik.http.routers.open-webui.tls=true"
      - "traefik.http.services.open-webui.loadbalancer.server.port=8080"

networks:
  proxy:
    external: true
```

Obraz ma przypiętą wersję zamiast `main`, żeby aktualizacja była naszą decyzją. Wersja `v0.11.4` jest z 21 września 2026, a przed instalacją sprawdzamy listę wydań projektu. Adres `10.20.10.45` to `gpu01` z Ollamą.

vim = Vi IMproved: ulepszony edytor vi

ZMIENIA SYSTEM

Instalacja Open WebUI na docker01, start: docker compose up -d

```
cd /docker/compose/open-webui
docker compose up -d
```

Przykład:

```
[+] Running 1/1
 ✔ Container open-webui  Started
```

> Pierwsze założone konto dostaje rolę administratora. Zakładamy je od razu po starcie, tak samo jak w Portainerze, a potem w Panelu administratora wyłączamy rejestrację nowych kont. Nowe konta w pliku powyżej wpadają do roli `pending` i czekają na zatwierdzenie.

◫◫ ZRZUT EKRANU DO UZUPEŁNIENIA

Zrzut do uzupełnienia: główne okno Open WebUI po zalogowaniu, z rozwiniętą listą modeli u góry (widoczny `gemma4:12b`) i jedną krótką rozmową testową.

◫◫ **cd** = change directory: przejdź do katalogu · **docker** = narzędzie do kontenerów

◫◫ TYLKO ODCZYT

Weryfikacja połączenia Open WebUI z Ollamą: curl

```
docker exec open-webui curl -s http://10.20.10.45:11434/api/tags
```

Przykład:

```
{"models":[{"name":"gemma4:12b","size":8032114688}]}
```

◫◫ **docker** = narzędzie do kontenerów

Z życia wzięte

◫◫ Z ŻYCIA WZIĘTE

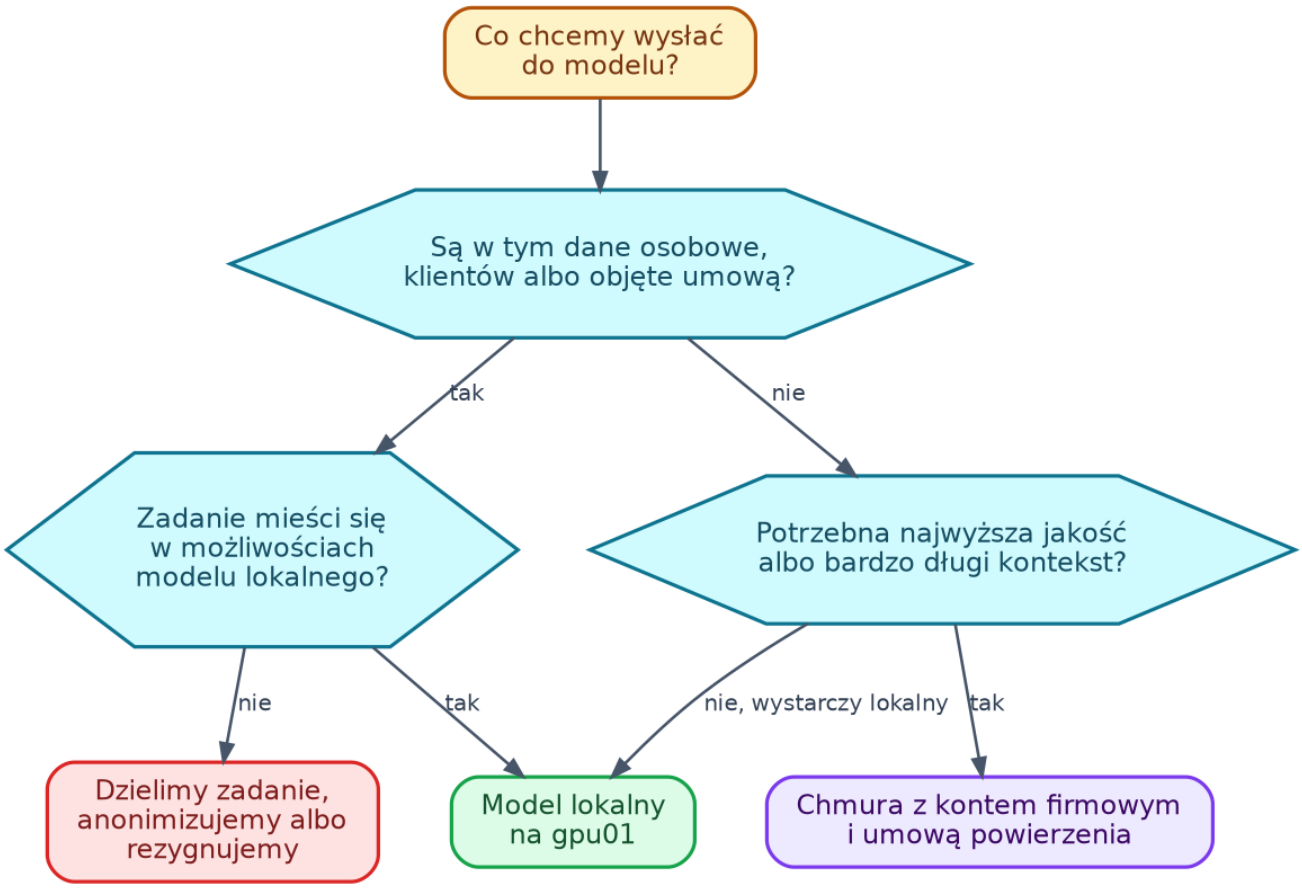
Czwartek, 11:20. Pan Zbigniew pyta, czy można używać czatu do streszczania umów z odbiorcami hurtowymi, w których są ceny i marże. Tłumaczymy, że takie umowy nie powinny wychodzić poza firmę, więc na `gpu01` stawiamy pilotaż z modelem lokalnym. Tomek po tygodniu mierzy `ollama ps`: model 12B mieści się na karcie w całości i odpowiada w kilka sekund, a próba z modelem 32B dzieli się z procesorem i pół minuty czeka na pierwsze słowo.

Jak to działa

Rozwiń: jak to działa pod spodem

Kiedy lokalnie, a kiedy w chmurze

Pojęcie: model lokalny. Model językowy, którego plik z parametrami (wagami) leży na naszym dysku i liczy się na naszym procesorze albo karcie graficznej. Żadne zapytanie nie opuszcza wtedy naszej sieci, chyba że sami to skonfigurujemy.



Decyzja: lokalnie czy w chmurze, zależnie od danych, możliwości modelu i potrzebnej jakości (kliknięcie otwiera pełny rozmiar)

Schemat zaczyna od danych, bo to one najczęściej przesądzają. Poniższa tabela zbiera pozostałe kryteria.

Kryterium	Model lokalny	Model w chmurze
Dane	zostają u nas, o ile sami ich nie wyślemy dalej	wychodzą do dostawcy, potrzebna umowa powierzenia
Jakość	zwykle niższa niż najlepsze modele zamknięte, dobra przy streszczeniach, szkicach, prostych skryptach	zwykle najwyższa, także przy długich zadaniach wieloetapowych
Koszt	sprzęt kupujemy raz, potem płacimy prąd i czas administratora	płacimy za tokeny, koszt rośnie z użyciem
Utrzymanie	sterowniki, aktualizacje, miejsce na dysku, kopie zapasowe po naszej stronie	po stronie dostawcy

Kryterium	Model lokalny	Model w chmurze
Dostępność	działa bez internetu, ale pada razem z naszym serwerem	zależy od łącza i od dostawcy
Nowe modele	musimy je sami pobrać i sprawdzić licencję	dostępne zwykle od razu

W praktyce zespoły rzadko wybierają jedną stronę. Dane poufne trafiają do modelu lokalnego, a zadania, w których liczy się najwyższa jakość i nie ma danych wrażliwych, idą do chmury z kontem firmowym. Zasady, co wolno wklejać gdzie, opisuje strona Polityka użycia AI w firmie: wzór dla Zielonej.

Ollama i llama.cpp

> **Pojęcie: Ollama.** Usługa, która pobiera modele z własnej biblioteki, zarządza ich ładowaniem do pamięci i udostępnia je przez API HTTP na porcie 11434. Wygodna jako serwer dla całego zespołu: jedno polecenie `ollama pull` i model jest gotowy.

> **Pojęcie: llama.cpp.** Projekt otwartoźródłowy, który uruchamia modele zapisane w formacie GGUF na procesorze i na kartach różnych producentów. Daje więcej pokręteł (liczba warstw na karcie, kwantyzacja własnych plików, tryby pracy serwera), a nowe funkcje pojawiają się w nim zwykle wcześniej.

Wybór nie jest wyłączny. Ollama pasuje, gdy chcemy szybko postawić usługę dla ludzi i dla Open WebUI. Samo llama.cpp pasuje, gdy potrzebujemy dokładnej kontroli nad jednym modelem albo własnego pliku GGUF.

Format GGUF i kwantyzacja

> **Pojęcie: GGUF.** Format pliku z modelem, w którym są parametry, słownik tokenów i metadane w jednym pliku. Używa go llama.cpp, a Ollama przechowuje w nim modele po pobraniu.

> **Pojęcie: kwantyzacja.** Zapisanie parametrów modelu na mniejszej liczbie bitów, na przykład 4 zamiast 16 na parametr. Plik robi się kilka razy mniejszy, a jakość spada nieznacznie przy 4 do 5 bitach i wyraźnie poniżej.

Oznaczenia typu `Q4_K_M` mówią, ile mniej więcej bitów przypada na parametr i jakiej metody użyto. Poniższa tabela dotyczy modelu o 8 miliardach parametrów i jest orientacyjna.

Zapis	Bity na parametr	Rozmiar pliku	Uwagi
F16	16	ok. 16 GB	oryginał bez kompresji
Q8_0	ok. 8,5	ok. 8,5 GB	praktycznie bez strat jakości
Q5_K_M	ok. 5,7	ok. 5,7 GB	bardzo dobry kompromis
Q4_K_M	ok. 4,9	ok. 4,9 GB	typowy wybór, w tej postaci Ollama dostarcza wiele modeli
Q3_K_M	ok. 3,9	ok. 3,9 GB	zauważalne pogorszenie
Q2_K	ok. 3,4	ok. 3,4 GB	jakość mocno spada, ostateczność

Zwykle lepiej wziąć większy model w 4 bitach niż mniejszy w 8 bitach, jeśli mieści się w pamięci. Pewność daje dopiero własna próba na naszych zadaniach, o czym piszemy na stronie Jak sprawdzić, czy to w ogóle działa.

Ile pamięci karty graficznej potrzeba

Pamięć potrzebna na kartę to rozmiar pliku modelu, plus pamięć na kontekst rozmowy, plus około 1 do 2 GB narzutu. Pamięć na kontekst rośnie z liczbą tokenów: dla modelu rodziny Llama 3.1 8B to około 1 GB na 8 tysięcy tokenów, dla wersji 70B około 2,5 GB (wyliczone z liczby warstw i głowic uwagi). Przy długich dokumentach i RAG kontekst potrafi zająć kilka gigabajtów, więc dokładamy zapas.

Wielkość modelu	Plik w ok. 4 bitach	Karta z zapasem na kontekst 8K	Przykład z biblioteki Ollamy
3 do 4 mld parametrów	2 do 3 GB	4 GB	`qwen3:4b` (2,5 GB)
7 do 9 mld	5 do 6 GB	8 GB	`qwen3:8b` (5,2 GB), `llama3.1:8b` (4,9 GB)
12 do 14 mld	8 do 9 GB	12 GB	`gemma4:12b` (7,7 do 8 GB), `qwen3:14b` (9,3 GB)
20 do 26 mld, architektura MoE	14 do 19 GB	16 do 24 GB	`gpt-oss:20b` (14 GB), `gemma4:26b` (16 do 19 GB)
27 do 32 mld	17 do 20 GB	24 GB, bez miejsca na długi kontekst	`qwen3:32b` (20 GB), `gemma4:31b` (19 do 20 GB)
70 mld	43 GB	48 GB	`llama3.1:70b` (43 GB)

Wielkość modelu	Plik w ok. 4 bitach	Karta z zapasem na kontekst 8K	Przykład z biblioteki Ollamy
120 mld, MoE, format MXFP4	65 GB	80 GB	`gpt-oss:120b` (65 GB)

> **Pojęcie: MoE (mixture of experts).** Architektura, w której model ma wiele wyspecjalizowanych bloków, ale przy każdym tokenie liczy tylko kilka z nich. W pamięci musi leżeć cały model, a szybkość zależy od części aktywnej, więc taki model jest szybszy niż gęsty model o tej samej wielkości plików.

Rozmiary pochodzą ze stron modeli w bibliotece Ollamy z 3 października 2026. Kolumna z kartą to nasze wyliczenie z reguły powyżej, a nie obietnica dostawcy. Dokładną liczbę dla własnego przypadku odczytujemy z `ollama ps` po załadowaniu modelu.

Procesor kontra karta graficzna

Generowanie odpowiedzi to głównie czytanie z pamięci: przy każdym tokenie model musi przejść przez całe swoje parametry. O szybkości decyduje więc przepustowość pamięci. Zwykła pamięć RAM w komputerze daje rząd kilkudziesięciu do stu kilkudziesięciu GB na sekundę, a pamięć wydajnych kart graficznych od kilkuset do ponad tysiąca GB na sekundę.

Cecha	Procesor i RAM	Karta graficzna
Szybkość odpowiedzi	kilka do kilkunastu tokenów na sekundę dla modelu 8B	kilkadziesiąt do ponad stu
Wczytanie długiego promptu	wolne, boli przy RAG	szybkie
Pojemność	dużo RAM, tanio	ograniczona VRAM, drogo
Kiedy	test, model do 8B, nikt nie czeka	usługa dla zespołu, modele 12B i większe

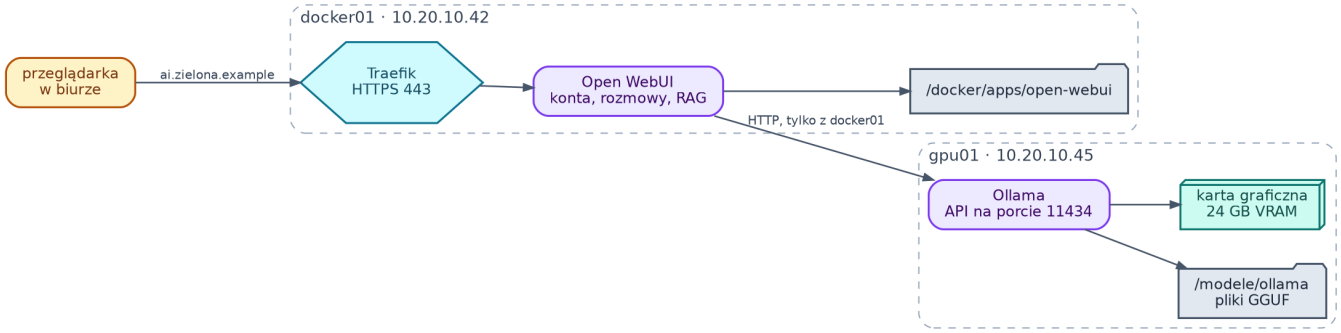
Liczby w tabeli to rzędy wielkości, a nie pomiar. Własny pomiar robimy komendą `ollama run NazwaModelu:Rozmiar --verbose` i odczytujemy `eval rate`. Gdy model nie mieści się na karcie, Ollama dzieli go między kartę i procesor, a `ollama ps` pokazuje proporcję. Ten układ działa, ale tempo spada w stronę procesora.

Obsługa kart: Ollama działa z kartami NVIDIA (sterownik 550 lub nowszy, według dokumentacji), z kartami AMD przez ROCm 7 oraz przez Vulkan i na Apple przez Metal.

Open WebUI

>

> **Pojęcie: Open WebUI.** Aplikacja WWW w stylu czatu, która łączy się z Ollamą (albo z innym API zgodnym z OpenAI) i daje zespołowi konta, historię rozmów, wybór modelu i wbudowane przeszukiwanie dokumentów.



Open WebUI na docker01 rozmawia przez HTTP z Ollamą na gpu01, a dane aplikacji leżą w katalogu zgodnym z konwencją (kliknięcie otwiera pełny rozmiar)

Ruch idzie tak: przeglądarka łączy się przez Traefika z kontenerem, a kontener wysyła zapytania do API na `gpu01`. Rozmowy i konta zapisują się w `/docker/apps/open-webui`, więc ten katalog wchodzi do kopii zapasowej i podlega tej samej klasyfikacji danych co treść rozmów. Ollama nie ma własnych kont, dlatego jej port zamykamy regułą zapory z fiszki i otwieramy wyłącznie dla `docker01`.

Wbudowane przeszukiwanie dokumentów (Knowledge) omawiamy na stronie RAG w praktyce: czat, który zna naszą bazę wiedzy.

Prywatność danych

Model lokalny rozwiązuje jeden problem: zapytania nie jadą do dostawcy. Nie rozwiązuje trzech innych, o których pamiętamy.

1. Rozmowy leżą w bazie Open WebUI. Domyślnie administrator widzi cudze rozmowy w panelu (zmienna `ENABLE_ADMIN_CHAT_ACCESS` ma wartość `True`), więc to polityka firmy decyduje, czy tak zostaje.
2. Ollama ma tryb chmurowy: modele z końcówką `cloud` liczą się na serwerach Ollamy, a nasze prompty wychodzą z firmy. Dokumentacja deklaruje, że nie używa ich do trenowania. W Zielonej ustawiamy `OLLAMA_NO_CLOUD=1`, żeby nikt nie wybrał takiego modelu przez pomyłkę.

3. Open WebUI ma funkcje, które sięgają do internetu, na przykład wyszukiwanie w sieci. W panelu administratora sprawdzamy, które są włączone, i wyłączamy niepotrzebne.

Kontrolę ruchu wychodzącego z `gpu01` robimy zwykłym narzędziem, na przykład `sudo ss -tnp | grep ollama`, i regułami zapory na `fw01`. Pobranie nowego modelu to jedyny moment, w którym serwer powinien sięgać na zewnątrz.

Licencje modeli otwartych

> **Pojęcie: model z otwartymi wagami.** Model, którego parametry można pobrać i uruchomić u siebie. To nie to samo co otwarte oprogramowanie: dane treningowe i sposób treningu zwykle pozostają tajne, a licencja bywa własna.

Licencja decyduje, czy wolno nam użyć modelu w firmie, w produkcie dla klientów i do trenowania innych modeli. Poniższa tabela pokazuje przykłady rodzin, a nie rekomendację.

Rodzina (przykład)	Licencja według karty modelu	Na co zwracamy uwagę
Qwen3	Apache 2.0	dopuszcza użycie komercyjne, wymaga zachowania informacji o licencji
gpt-oss	Apache 2.0	jw.
Gemma 4	Apache 2.0 (starsze Gemmy miały własne warunki)	przy starszych wersjach czytamy zasady użycia
DeepSeek R1	MIT	modele destylowane dziedziczą licencję modelu bazowego, na przykład Qwen albo Llama
Llama 3.1	własna licencja Meta	próg 700 mln aktywnych użytkowników, obowiązek napisu „Built with Llama", polityka dopuszczalnego użycia

Trzy zasady na co dzień: licencję sprawdzamy dla konkretnej wersji, a nie dla rodziny, bo zmienia się między wydaniami. Modele pochodne (destylaty, wersje dostrojone przez społeczność) mogą mieć licencję bazową, a nie tę, która widnieje na stronie pochodnej. Wynik sprawdzenia zapisujemy w rejestrze narzędzi AI z polityki firmowej, razem z wersją i datą.

Częste błędy

Objaw	Przyczyna	Co robimy
`ollama ps` pokazuje 100% CPU, odpowiedzi trwają minuty	sterownik niezaładowany albo karta niewidoczna dla usługi	`nvidia-smi`, `journalctl -e -u ollama`, po instalacji sterownika restart serwera
Open WebUI ma pustą listę modeli	Ollama słucha tylko na `127.0.0.1` albo zapora blokuje `docker01`	`OLLAMA_HOST=0.0.0.0:11434`, reguła zapory, test `curl` z kontenera
Model nie pamięta początku długiego dokumentu	okno kontekstowe 4096 tokenów domyślnie	`num_ctx` w zapytaniu albo `OLLAMA_CONTEXT_LENGTH` w usłudze
Dysk systemowy się zapełnia po kilku pobraniach	modele w `/usr/share/ollama`	`OLLAMA_MODELS` na osobny wolumen
Pierwsza odpowiedź po przerwie trwa kilkanaście sekund	model wypadł z pamięci po 5 minutach	zwiększamy `OLLAMA_KEEP_ALIVE`
Błąd braku pamięci przy ładowaniu	model albo kontekst za duży na kartę	mniejszy rozmiar, niższa kwantyzacja, krótszy `num_ctx`
Ktoś spoza docker01 wysyła zapytania do API	`0.0.0.0` bez reguły zapory, API nie ma uwierzytelniania	rich rule z jednym adresem źródłowym

Ćwiczenia

1. Mamy kartę 24 GB. Czy model 32B w 4 bitach (plik około 20 GB) zmieści się razem z kontekstem 32 tysięcy tokenów?

Odpowiedź

Nie. Plik zajmuje 20 GB, narzut około 1 do 2 GB, a kontekst 32 tysięcy tokenów dla takiego modelu to kilka GB (dla architektury z 64 warstwami i 8 głowicami uwagi około 8 GB). Razem wychodzi ponad 30 GB. Przy kontekście 8 tysięcy tokenów całość ledwo się mieści, a przy dłuższym Ollama wyniesie część warstw na procesor.

1. `ollama ps` pokazuje w kolumnie `PROCESSOR` wartość `38%/62% CPU/GPU`. Co to znaczy i co z tym robimy?

Odpowiedź

Model nie mieści się w całości na karcie, więc 38 procent liczy procesor, a 62 procent karta. Odpowiedzi będą wolniejsze, szczególnie przy długim prompcie. Wybieramy mniejszy rozmiar modelu, niższą kwantyzację albo krótszy kontekst, aż kolumna pokaże `100% GPU`.

1. Ola chce użyć pobranego modelu w produkcie dla klientów Zielonej. Co sprawdzamy, zanim to wejdzie do planu?

Odpowiedź

Licencję konkretnej wersji modelu: czy dopuszcza użycie komercyjne, czy nakłada obowiązek napisu, progi użytkowników lub zakazy zastosowań. Jeśli to model dostrojony przez kogoś trzeciego, sprawdzamy też licencję modelu bazowego. Wynik zapisujemy w rejestrze narzędzi AI.

1. Dlaczego po ustawieniu `OLLAMA_HOST=0.0.0.0:11434` od razu dodajemy regułę zapory?

Odpowiedź

Bo API Ollamy nie ma uwierzytelniania. Po otwarciu na wszystkie interfejsy każdy, kto dojdzie do portu 11434, może wysyłać zapytania i pobierać lub usuwać modele. Reguła z jednym adresem źródłowym zostawia dostęp tylko serwerowi z Open WebUI.

Źródła

- Dokumentacja Ollamy, instalacja na Linuksie: https://docs.ollama.com/linux
- Dokumentacja Ollamy, Windows: https://docs.ollama.com/windows
- Dokumentacja Ollamy, FAQ (OLLAMA_HOST, OLLAMA_MODELS, KEEP_ALIVE, kontekst, OLLAMA_NO_CLOUD): https://docs.ollama.com/faq
- Dokumentacja Ollamy, API: https://docs.ollama.com/api
- Dokumentacja Ollamy, obsługiwane karty graficzne: https://docs.ollama.com/gpu
- Dokumentacja Ollamy, modele chmurowe i prywatność: https://docs.ollama.com/cloud
- Dokumentacja Ollamy, import GGUF: https://docs.ollama.com/import
- Biblioteka modeli Ollamy (rozmiary, sprawdzone 3 października 2026): https://ollama.com/library
- Wydania Ollamy: https://github.com/ollama/ollama/releases
- Wydania llama.cpp: https://github.com/ggml-org/llama.cpp/releases
- Open WebUI, szybki start i znaczniki obrazów: https://docs.openwebui.com/getting-started/quick-start/
- Open WebUI, zmienne środowiskowe: https://docs.openwebui.com/reference/env-configuration/
- Sterownik NVIDIA na Rocky Linux 9: https://linuxcapable.com/how-to-install-nvidia-drivers-on-rocky-linux
- Karty modeli i licencje: https://huggingface.co/Qwen/Qwen3-8B, https://huggingface.co/deepseek-ai/DeepSeek-R1, https://huggingface.co/meta-llama/Llama-3.1-8B

- Gemma 4 na licencji Apache 2.0: https://www.theregister.com/2026/04/02/googles_gemma_4_open_weights/
- Otwarte wagi a otwarte oprogramowanie, przegląd licencji: https://www.softwareseni.com/open-weights-vs-open-source-and-the-apache-2-0-licensing-shift/

Revision #18
Created 2026-10-03 09:39:13 UTC by Tomasz Konieczny
Updated 2026-10-26 14:10:18 UTC by Tomasz Konieczny