

Eskadra Bielika

Misja Druga

Poznań • 18.07.2026

Maksymilian Małecki



Tech Lead @ Insly — produkcyjny RAG (OWU AI) na platformie insurtech

- 17+ lat w IT · wcześniej NordSecurity, ORO, Arena.pl
- Na scenie: 3× PHPers Summit · DevAI 2025



Agenda

1. **Wstęp teoretyczny** — czym jest Bielik, czym jest RAG i czemu nie duży frontier model
2.  **Logowanie do GCP** +  repozytorium warsztatu
3. **Krok po kroku** — budujemy działający RAG na Cloud Run · *w środku 🍕 przerwa*
4.  **Produkcyjny RAG** — czego brakuje, zagrożenia, koszty

Praktyka, nie teoria — wychodzisz z **działającym, suwerennym RAG-iem** na Bieliku.

Suwerenne i wiarygodne AI



Od dokumentów firmowych do
inteligentnej bazy wiedzy w oparciu
o model Bielik i Google Cloud



 Podłącz się do WiFi



Sieć: WCWI • Hasło: wucewuizabramka

Wstęp teoretyczny

Bielik · czym jest RAG · czemu nie duży frontier model

Który Bielik do czego?

Szablon Eskadry pokazał lineup. Teraz: **który wybrać do swojego use case?**

Zadanie	Model	Pass rate (RAG)	Dlaczego
Klasyfikacja, tagowanie, ekstrakcja	1.5B	—	ultra-szybki, tani
Chatbot on-device / offline	1.5B	—	zmieści się na laptopie / edge
RAG nad dokumentami firmowymi	4.5B ★	90%	nasz warsztat — dobra jakość, GPU L4
Produkcja RAG	7B Minitron	97% vs comp / 100% vs Bieliki	sweet spot koszt/jakość, szybszy od 11B (8.8s)
Kreatywne pisanie, agenty, trudne Q&A	11B	100% (wolniejszy, 13s)	flagship, najwyższa jakość PL
Moderacja wejścia / wyjścia	Guard 0.1B–0.5B	—	lekki filtr safety przed i po LLM

Reguła kciuka: zacznij od **najmniejszego, który spełnia wymagania** — różnica kosztu GPU między 1.5B a 11B jest rzędu **10×**. Minitron 7B bije 11B w szybkości i dokładności na polskiej domenie.

Bielik head-to-head — OWU AI benchmark

30 pytań z ubezpieczeń · hybrid search · Insly, 2026-04-30

	4.5B Q8_0	7B Minitron Q4_K_M ★	11B Q6_K
Pass rate	90%	100%	100%
Avg score	0.785	0.949	0.915
Czas RAG	5.4 s	8.8 s	13.0 s
Halucynacje	2	5	6
Idealne odpowiedzi	11 / 30	26 / 30	21 / 30

7B Minitron wygrywa — wyższy score i 26/30 idealnych odpowiedzi przy 4× szybszym RAG niż 11B. 4.5B świetnie na warsztat (szybki, L4 wystarczy), do produkcji warto awansować na 7B.

Bielik vs konkurencja — OWU AI benchmark

30 pytań z ubezpieczeń · hybrid search · Insly, 2026-04-30

Model	Pass rate	Avg score	Halucynacje
Bielik Minitron 7B	97%	0.916	8
EuroLLM 9B	90%	0.791	7
Gemma 4 8B	73%	0.620	6
Llama PLLuM 8B	53%	0.540	4

Bielik wygrywa na domenę polskiej — EuroLLM blisko, ale znacznie słabszy na złożonych pytaniach prawnych.

Bielik — polski LLM w naszym stacku

- Model `SpeakLeash/bielik-4.5b-v3.0-instruct` (kwantyzacja **Q8_0**)
- Uruchomiony w kontenerze przez **Ollama** — standardowy runtime dla GGUF/Q8_0
- Świetnie rozumie polski język i kontekst kulturowy
- Rozmiar po kwantyzacji: ~5 GB, mieści się na jednej karcie GPU
- **Dlaczego 4.5B, a nie 11B?** — sweet spot jakość/koszt, mniejsze cold starty, mieści się na tańszym L4

Ten sam kontener z Ollamą można łatwo przestawić na `bielik-11b-v3.0` — tylko zmienić nazwę modelu w `Dockerfile` i `main.py`.

RAG natywnie po polsku — Bielik

**Polski AI to nie ciekawostka.
To infrastruktura.**

© 2026 mmx3.pl



0:00 / 0:25



+12 pp precyzji · **−20%** kosztu · dane zostają na miejscu (**RODO**)

Co LLM *wie*, a czego *nie wie*?

LLM to zamrożony mózg:

- wie wszystko, co było w danych treningowych (np. do 2023-2024)
- **nie wie** nic o Twojej firmie, projekcie, regulaminach
- **nie wie**, co się stało wczoraj
- gdy nie wie — **zmyśla** (halucynacje) z pełnym przekonaniem

Pytanie: *"O której jest śniadanie w naszym hotelu?"*

Sam LLM: **"Zwykle między 7:00 a 10:00"** — brzmi sensownie, ale to zgadywanie.

Trzy sposoby, żeby LLM znał Twoje dane

Podejście	Jak działa	Kiedy używać	Minus
Fine-tuning	dotrenowanie modelu na Twoich danych	styl, ton, specyficzne zadania	drogie, powolne, wiedza wciąż się starzeje
Prompt stuffing	wrzuć wszystko do promptu	gdy dokumentów jest mało	limit kontekstu, \$\$\$ za token
RAG ★	wyszukaj <i>tylko</i> to, co istotne i dołącz do promptu	wiedza firmowa, duże korpusy	trzeba zbudować pipeline

RAG = "daj LLM-owi ściągę **dopiero w chwili pytania**".

RAG w jednym zdaniu

Retrieval-Augmented Generation = wyszukaj odpowiedni fragment dokumentu → dołóż go do promptu → niech LLM odpowie *na podstawie kontekstu*.

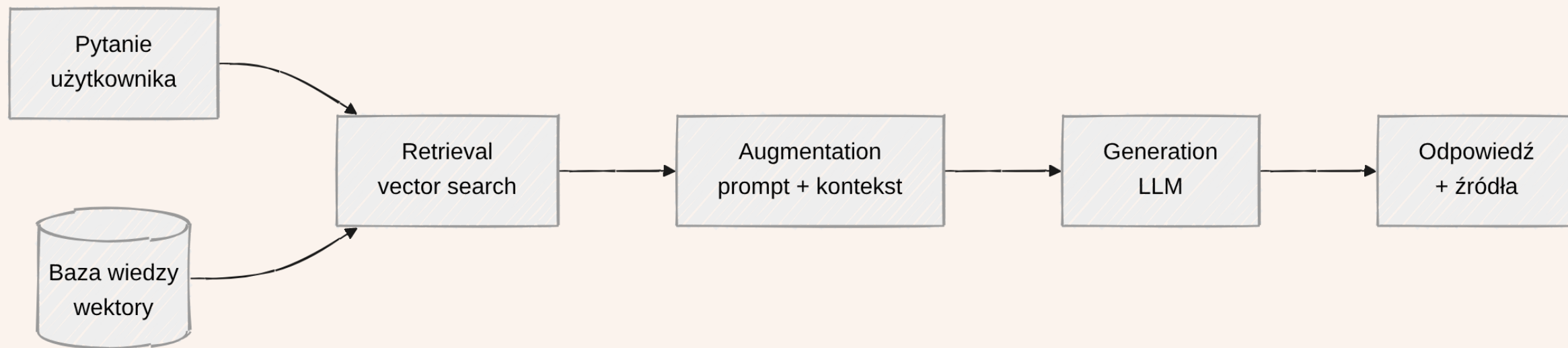
Zysk:

- odpowiedzi oparte o fakty z Twoich dokumentów
- aktualizacja wiedzy bez fine-tuningu
- źródła cytowalne (widać, *skąd* pochodzi odpowiedź)

Koszt:

- trzeba zbudować wektorową bazę wiedzy i warstwę wyszukiwania

RAG — koncept

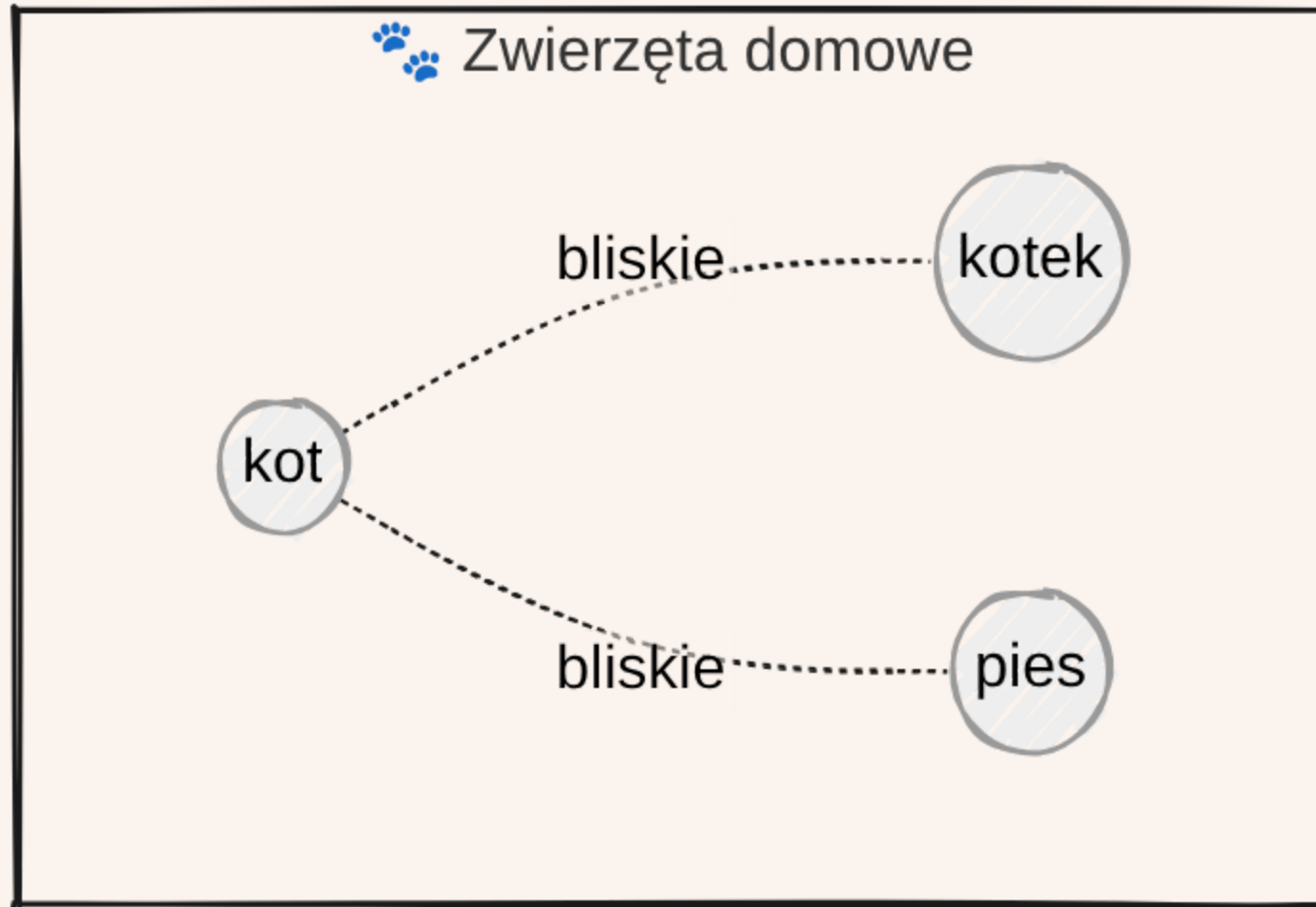


Trzy litery = trzy fazy: **R**etrieval → **A**ugmentation → **G**eneration.

Embeddings – intuicja

Embedding = zamiana tekstu na **wektor liczb** (np. 768 wymiarów).

Magia: bliskie znaczenia $\Rightarrow$ bliskie wektory w przestrzeni.



Embeddings — intuicja (2/2)

- "basen" i "pływalnia" → **blisko siebie**, "basen" i "parking" → **daleko**
- działa *nawet* między językami (multilingual embeddings)

Wektor to "odcisk palca znaczenia" — liczby, z którymi umie pracować komputer.

Vector search — szukanie po znaczeniu

Zapytanie użytkownika → embedding → **szukaj najbliższych wektorów** w bazie.

Cosine similarity = kąt między wektorami. Blisko 1 = bardzo podobne znaczenie.

Chunking — jak pociąć dokumenty

Wrzucenie całego PDF-a jako jednego wektora = kiepskie wyniki.

Rozbij na **fragmenty** (chunks), każdy staje się osobnym wpisem w bazie.

Zasady kciuka:

- Rozmiar: **300–800 tokenów** (ok. 1-2 akapity)
- **Nakładka** (overlap) 10–20% — żeby nie urwać zdania w połowie
- Dziel po **granicach semantycznych** (nagłówki, akapity), nie losowo
- W tym warsztacie: CSV, gdzie **każdy wiersz** = gotowy chunk

"Garbage in, garbage out" — jakość chunków = sufit jakości całego RAG-u.

Chunking — od PDF-a do wektorów

Każdy wybór modelu jest wtórny

wobec jakości chunkingu i metadanych.

© 2026 mmx3.pl

▶ 0:00 / 0:30



Pipeline RAG — trzy fazy

Ingestion (raz / przy aktualizacji danych): dokumenty → chunki → wektory → baza.

Query (każde pytanie): pytanie → wektor → top-k chunki → prompt + kontekst → LLM → odpowiedź.

Dlaczego w ogóle o tym mówimy?

- **LLM-y halucynują** — brakuje im aktualnej, wewnętrznej wiedzy firmy
- **Dane wrażliwe** nie mogą wyciekać do zewnętrznych API (USA/Chiny)
- **Polski kontekst kulturowy** — modele globalne często "nie łapią" niuansu
- Potrzebujemy rozwiązania: **własne dane + własny model + własna chmura (lub region EU)**

Stąd **suwerenne AI**: pełna kontrola nad danymi, modelem i miejscem przetwarzania.

Po co RAG? ChatGPT vs RAG na OWU

ChatGPT

Czy TOW AutoAssistance obejmuje ochroną zdarzenia na terenie Gruzji?

OWUAI

Gruzja jest klasyfikowana jako kraj europejski przez większość organizacji. TOW Assistance obejmuje Europę, więc zdarzenia w Gruzji powinny być objęte ochroną w wariancie Europejskim. Polecam sprawdzić szczegóły polisy.

NIE.
TOW AutoAssistance wprost wyklucza:
Gruzję, Białoruś, Ukrainę, Rosję
i azjatycką część Turcji.

Zdarzenie w Gruzji nie jest objęte ochroną.

Podstawa: §3 OWU TOW AutoAssistance 2026

▶ 0:00 / 1:00

Gruzja jest WPROST wykluczona z OWU TOW AutoAssistance



§3 OWU TOW AutoAssistance 2026

🔑 Logowanie do Google Cloud

1. Wejdź na **console.cloud.google.com** → zaloguj się **kontem Google** z warsztatu
2. Otwórz **Cloud Shell** (`>_` w prawym górnym rogu) — gotowy terminal w przeglądarce, zero instalacji
3. Odbierz **kredyty GCP** (QR) i ustaw projekt:

```
gcloud config set project <PROJECT_ID>
```



Cały warsztat robimy w **Cloud Shell** — nic nie instalujemy lokalnie.

Setup środowiska — `setup_env.sh`

```
export PROJECT_ID=$(gcloud config get-value project)
export REGION="europe-west1"
export EMBEDDING_SERVICE="embedding-gemma"
export LLM_SERVICE="bielik"
export BIGQUERY_DATASET="rag_dataset"
export BIGQUERY_TABLE="hotel_rules"
```

Wymagane API:

```
gcloud services enable run.googleapis.com \
  cloudbuild.googleapis.com \
  artifactregistry.googleapis.com \
  bigquery.googleapis.com
```

Repozytorium warsztatu



Zeskanuj → sklonuj w **Cloud Shell** i wejdź do katalogu projektu.

Krok po kroku

Co dokładnie robimy na warsztacie

Ścieżka uczestnika — co dziś zrobimy

9 kroków: od pustego projektu GCP do działającego RAG-a. Deploy modeli leci w tle (~15 min) — w tym czasie omawiamy teorię.

1. Projekt GCP + Cloud Shell → `source setup_env.sh`

2. 🚀 Deploy **Bielik LLM** — Cloud Run + GPU L4

3. 🚀 Deploy **EmbeddingGemma** — Cloud Run, CPU

4. Inicjalizacja **BigQuery** — baza wektorowa

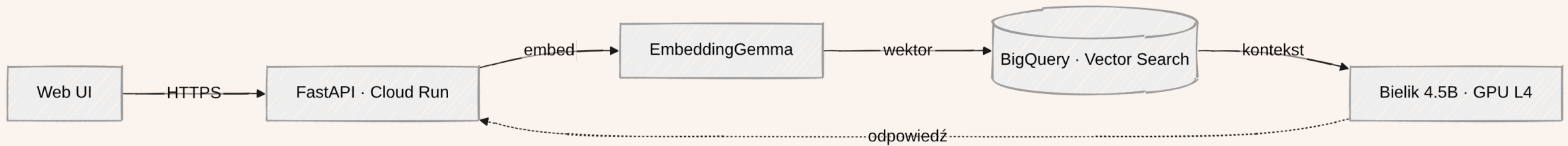
5. Deploy **Orchestration API** — FastAPI

6. `/ingest` — dokumenty → wektory

7. `/ask` — pytanie → RAG w akcji

8. **Web UI** — `/ask` vs `/ask_direct` na żywo

Architektura — z lotu ptaka



Legenda: FastAPI = orchestrator, EmbeddingGemma = tekst → wektor, BigQuery = vector store + search, Bielik = generation LLM. Wszystko **serverless** w **europe-west1** — płacisz tylko za wywołania.

Stack technologiczny

Warstwa	Technologia	Rola
LLM	Bielik 4.5B v3 Instruct (SpeakLeash)	generowanie odpowiedzi po polsku
Embeddings	EmbeddingGemma (Google DeepMind)	tekst → wektor
Runtime modeli	Ollama + Cloud Run + GPU (L4)	serwowanie modeli
Baza wektorowa	BigQuery Vector Search	semantyczne wyszukiwanie
Orchestration	FastAPI (Python)	API + prosty Web UI
Deployment	Cloud Run (serverless)	cała infra bez zarządzania VM

Start! Odpalamy deploy

Teraz — zanim przejdziemy do teorii.

```
# Terminal 1 – Bielik LLM (~15 min)
```

```
cd llm && ./cloud_run.sh
```

```
# Terminal 2 – EmbeddingGemma (~15 min, osobny terminal)
```

```
cd embedding_model && ./cloud_run.sh
```

Oba deploje działają równolegle. Opowiem wam co tam śmiga — wrócimy za ~15 min sprawdzić status.



Przerwa — 20 min

Deploy modeli leci w tle — Bielik (GPU L4) i EmbeddingGemma startują (~15 min).

Wrócimy, gdy usługi będą gotowe. WiFi: `WCWI` . Pytania mile widziane!

Deploye gotowe?

Sprawdź status obu usług

```
gcloud run services describe $LLM_SERVICE --region=$REGION \  
  --format='value(status.conditions)'
```

```
gcloud run services describe $EMBEDDING_SERVICE --region=$REGION \  
  --format='value(status.conditions)'
```

Jeśli OK – załaduj dokumenty do bazy wektorowej

```
curl -X POST $ORCHESTRATION_URL/ingest
```

Coś nie działa? → zajrzyj do **troubleshooting.md** (Ctrl+F na treści błędu).

Bielik na Cloud Run — komenda deploy

```
gcloud run deploy bielik \  
  --source . --region $REGION \  
  --cpu 8 --gpu 1 --gpu-type nvidia-l4 \  
  --memory 16Gi --max-instances 1 \  
  --no-allow-unauthenticated \  
  --no-cpu-throttling \  
  --timeout=600
```

Serverless GPU = zero kosztu gdy nikt nie pyta. **Cold start** L4 to ~30-60s.

Bielik na Cloud Run — flagi i ich rola

Flaga	Po co
<code>--gpu 1 --gpu-type nvidia-l4</code>	najtańsza karta z VRAM dla 4.5B
<code>--memory 16Gi --cpu 8</code>	wagi modelu + bufor Ollamy
<code>--max-instances 1</code>	bez auto-scale (koszty!)
<code>--no-cpu-throttling</code>	GPU bez wstrzymywania
<code>--no-allow-unauthenticated</code>	tylko IAM + ID token
<code>--timeout=600</code>	generacja 4.5B 1-2 min przy długim kontekście

EmbeddingGemma — uwaga praktyczna (★)

Szablon Eskadry opisał *czym* jest EmbeddingGemma. Tutaj: **jak ją zdeployować, żeby było 10× taniej.**

- Używana **dwukrotnie** w pipeline:
 - i. przy *ingestion* — dokumenty → wektory w BigQuery
 - ii. przy *query* — pytanie użytkownika → wektor do wyszukiwania

💡 Działa wyłącznie na CPU — **bez GPU**

EmbeddingGemma jest tak małym modelem, że **nie potrzebuje GPU**. Kontener Cloud Run zdeployowany jest z samym CPU — **znacząco obniża koszty** usługi embeddingu.

```
payload = {"model": "embeddinggemma", "input": text}
response = requests.post(f"{EMBEDDING_URL}/api/embed",
                        json=payload, headers=headers)
embedding = response.json().get("embeddings", [[]])[0]
```

Implikacja kosztowa: EmbeddingGemma to ~\$0.12/h (CPU+RAM), a nie ~\$1.31/h jak Bielik z GPU L4 — **10× taniej** za usługę embeddingu.

BigQuery Vector Search

- Pełnoskalowa hurtownia danych, która **dodatkowo** robi wyszukiwanie wektorowe
- Zero zarządzania infrastrukturą, skalowanie do miliardów wierszy
- Schemat tabeli jest prosty:

```
schema = [  
    bigquery.SchemaField("id",      "STRING",  mode="REQUIRED"),  
    bigquery.SchemaField("content", "STRING",  mode="REQUIRED"),  
    bigquery.SchemaField("embedding", "FLOAT64", mode="REPEATED"),  
]
```

Tabela: `rag_dataset.hotel_rules`

Wyszukiwanie wektorowe w SQL

```
SELECT base.content, distance
FROM VECTOR_SEARCH(
  TABLE `project.rag_dataset.hotel_rules`,
  'embedding',
  (SELECT [0.12, -0.03, ...] AS embedding),
  top_k ⇒ 3,
  distance_type ⇒ 'COSINE'
)
```

- **COSINE distance** — standard dla podobieństwa semantycznego
- **top_k = 3** — bierzemy trzy najbliższe dokumenty jako kontekst
- Całość — natywny SQL, bez dodatkowych bibliotek

FastAPI — klej, który trzyma wszystko razem

Cztery endpointy w `orchestration/main.py`:

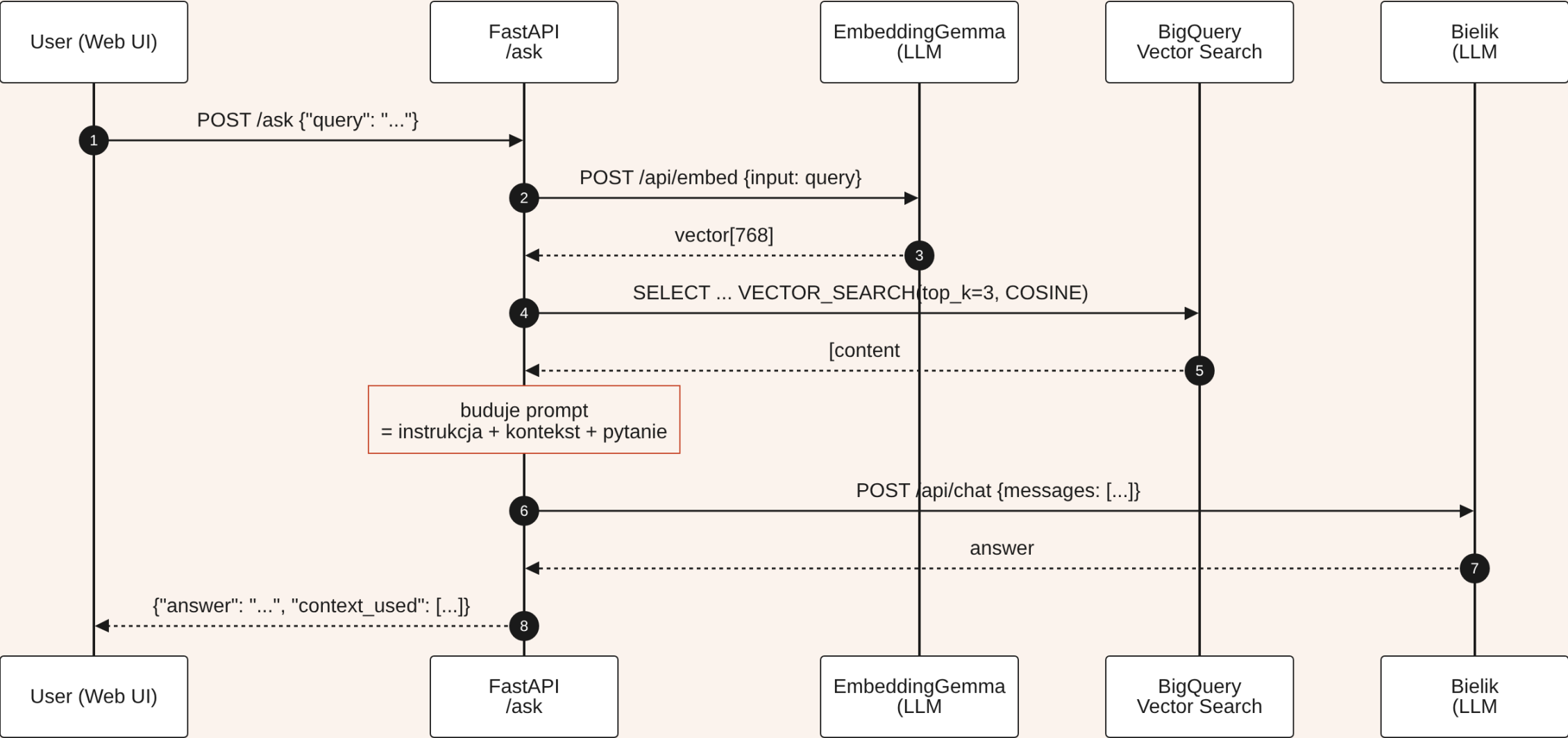
Endpoint	Rola
GET /	serwuje prosty Web UI
POST /ingest	CSV → embeddings → BigQuery
POST /ask	RAG : embedding → vector search → prompt → Bielik
POST /ask_direct	baseline : pytanie wprost do Bielika, bez kontekstu

`/ask` vs `/ask_direct` — to jest serce demo: pokazujemy różnicę na żywo.

Przepływ zapytania `/ask` — krok po kroku

1. Użytkownik: *"Jak często powinien być mierzony poziom chloru w basenie?"*
2. **EmbeddingGemma** → zamienia pytanie na wektor
3. **BigQuery Vector Search** → znajduje top-3 najbliższe fragmenty regulaminu
4. Orchestrator **buduje prompt**: instrukcja + kontekst + pytanie
5. **Bielik** generuje odpowiedź opartą *tylko* na kontekście
6. API zwraca `{answer, context_used}` → UI pokazuje odpowiedź i źródła

Sequence diagram: FastAPI ↔ LLM ↔ BigQuery ↔ LLM



Dwa wywołania LLM na zapytanie: **embedding** (LLM #1) → BigQuery → **generation** (LLM #2).

System Prompt (fragment z `main.py`)

```
prompt = (  
    f"Jesteś pomocnym asystentem odpowiadającym na pytania "  
    f"dotyczące zasad hotelowych. "  
    f"Odpowiedz na poniższe pytanie bazując TYLKO "  
    f"na dostarczonym kontekście.\n\n"  
    f"KONTEKST:\n{context_text}\n\n"  
    f"PYTANIE:\n{query}"  
)
```

Klucz: instrukcja *"bazując TYLKO na dostarczonym kontekście"* minimalizuje halucynacje.

Dane demo — hotel_rules.csv

- Przykładowy dataset: regulamin hotelu (basen, śniadanie, parking, cisza nocna...)
- Każdy wiersz = jeden "fakt" → jeden wektor w BigQuery
- Zasilenie bazy jednym `curl` em:

```
curl -X POST "$ORCHESTRATION_URL/ingest" \  
  -F "file=@vector_store/hotel_rules.csv"
```

W realnym świecie: PDF-y, Confluence, Notion, SharePoint → chunking → `/ingest`.

Demo — porównanie na żywo

Web UI wysyła **jednocześnie** zapytanie do `/ask` i `/ask_direct` :

<code>/ask_direct</code> (sam Bielik)	<code>/ask</code> (Bielik + RAG)
"Zwykle poziom chloru mierzy się... raz dziennie" (ogólniki, halucynacja?)	"Zgodnie z regulaminem — co 2 godziny" (+ cytowany fragment)

Pokazujemy też **dokładne fragmenty** z BigQuery, z których model skorzystał → pełna *wiarygodność*.

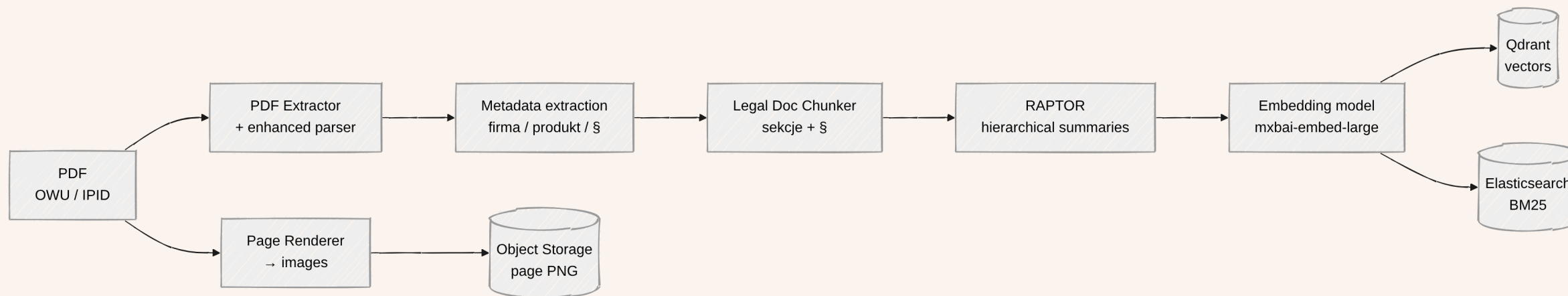
Produkcyjny RAG

Czego brakuje naszemu warsztatowemu RAG-owi · zagrożenia · koszty

Co warto rozszerzyć?

- **Re-ranking** wyników (np. drugi model ocenia trafność)
- **Multi-source ingestion** — konektory do Confluence / Drive / SharePoint
- **Hybrid search** — łączenie wektorów z full-text search
- **Streaming odpowiedzi** — SSE zamiast blokującego POST-a
- **Eval harness** — benchmark jakości po każdej zmianie
- **Role systemowe** — "zachowuj się jak pirat / ekspert IT" (z README)

OWU AI — Data Ingestion Pipeline (offline)



Co robi każdy etap:

- **PDF Extractor + enhanced parser** — tekst z warstw PDF + OCR fallback dla skanów
- **Metadata extraction** — LLM/regex wydobywa *firma / produkt / numer paragrafu* → filtrowalne w retrievalu
- **Legal Doc Chunker** — dzieli po **sekcjach i paragrafach OWU** (nie po naiwnych 500 tokenach)
- **RAPTOR** — buduje *hierarchiczne streszczenia* (klastry dokumentów) → lepszy recall dla pytań ogólnych
- **mxbai-embed-large** — 1024-dim, lepsze od starszych `all-MiniLM` na polskim
- **Hybrid storage** — wektory w **Qdrant**, BM25 w **Elasticsearch**, obrazy stron w Object Storage (potrzebne do cytowań)

RAPTOR — retrieval hierarchiczny

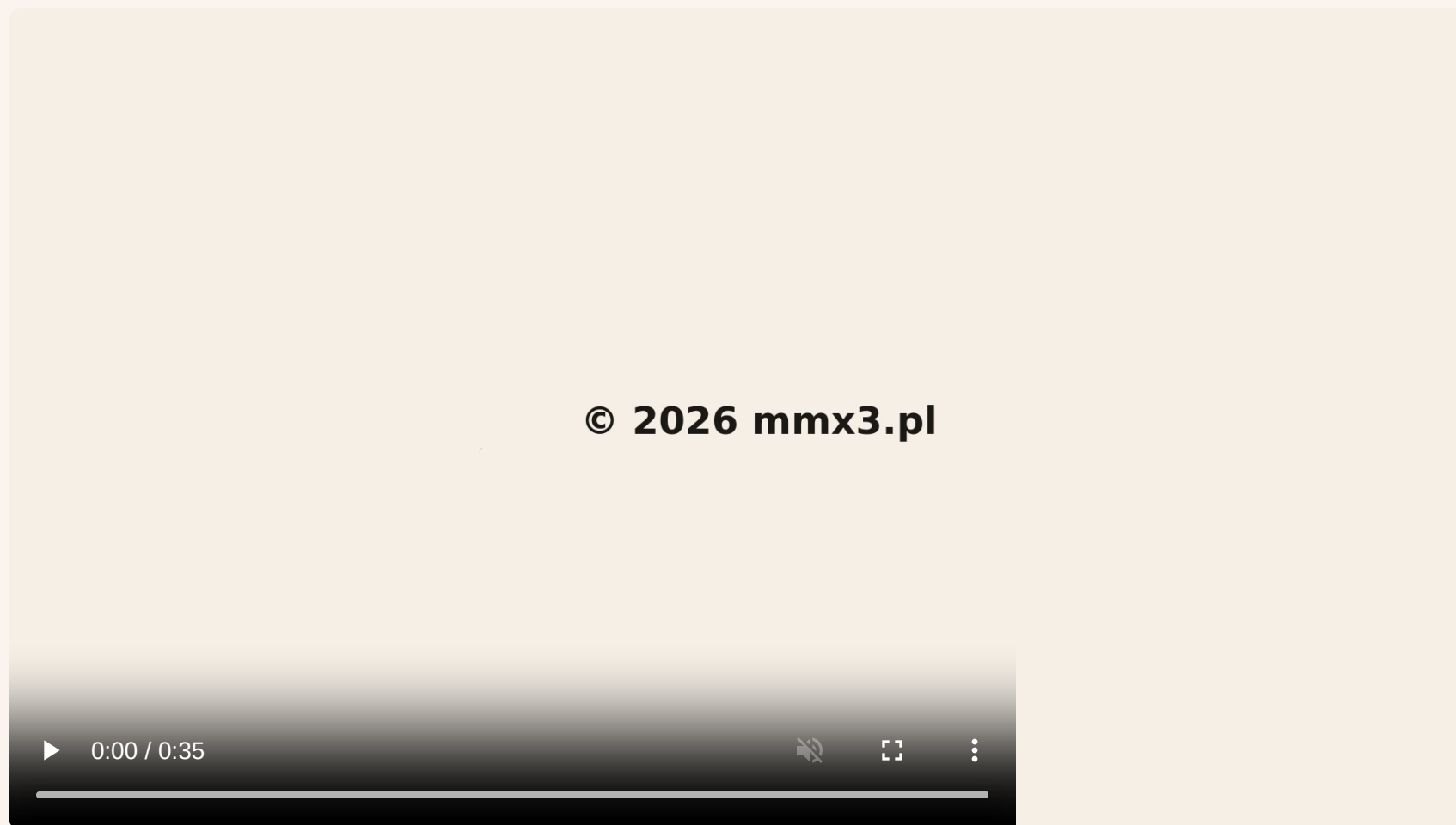
Techniki się nie wykluczają.
Komponują się.

© 2026 mmx3.pl

▶ 0:00 / 0:30



HyDE — domykanie luki semantycznej



Recall 0.61 → 0.79 — LLM generuje hipotetyczną odpowiedź; jej embedding "domyka" lukę między językiem pytania a językiem dokumentu.

Ewaluacja RAG — 5 metryk

RAG bez ewaluacji to demo.
Z ewaluacją: produkt.

© 2026 mmx3.pl

▶ 0:00 / 0:25



Halucynacje — 5 checkpointów

**Wolimy «nie wiem»
niż pewnie błędną odpowiedź.**

© 2026 mmx3.pl

▶ 0:00 / 0:43



Guardrails — izolacja kontekstu

Guardrails to nie filtr na output.
To architektura.

© 2026 mmx3.pl

▶ 0:00 / 0:36



Bezpieczeństwo i uprawnienia

- Usługi Cloud Run `--no-allow-unauthenticated` — tylko uwierzytelnione wywołania
- Orchestrator pobiera **ID token** przez `google.oauth2.id_token.fetch_id_token`
- IAM: rola `roles/run.invoker` dla użytkownika/serwisu
- Dane **nie opuszczają** regionu `europe-west1` — suwerenność spełniona

```
token = google.oauth2.id_token.fetch_id_token(request, audience)
headers = {"Authorization": f"Bearer {token}"}
```

💰 Frontier API vs self-hosted Bielik (per 1M tokenów)

Ceny publiczne, kwiecień 2026:

Model	Input \$/1M	Output \$/1M	Uwagi
Bielik 4.5B self-hosted (L4)	~\$1.20	~\$9.10	<i>amortyzacja przy 100% GPU</i>
Gemini 2.5 Flash	\$0.30	\$2.50	budżet, ograniczona jakość PL
Gemini 3 Flash	\$0.50	\$3.00	lepszta jakość, wciąż tani
GPT-5 (bazowy)	\$0.63	\$5.00	mainstream
Gemini 3 Pro	\$2.00	\$12.00	flagship, 2M kontekstu
Claude Sonnet 4.6	\$3.00	\$15.00	premium
Claude Opus 4.7	\$5.00	\$25.00	top reasoning

Kontekstowy koszt self-hostingu zakłada GPU wysycone na 100%. Przy low-traffic realny koszt/token może być **10-100× wyższy**, bo płacisz za czas bez inferencji.

Break-even — kiedy własny hosting się zwraca?

Koszt bazowy Bielika 24/7: ~\$942/miesiąc. Tyle musi kosztować API przy danym wolumenie:

API	Miesięczny wolumen output = \$942
Gemini 2.5 Flash (\$2.50/1M)	~ 377M (~12M/dzień)
Gemini 3 Flash (\$3/1M)	~ 314M (~10M/dzień)
GPT-5 / Haiku 4.5 (\$5/1M)	~ 188M (~6M/dzień)
Gemini 3 Pro (\$12/1M)	~ 79M (~2.5M/dzień)
Claude Sonnet 4.6 (\$15/1M)	~ 63M (~2M/dzień)
Claude Opus 4.7 (\$25/1M)	~ 38M (~1.2M/dzień)

Skalowanie — jak myśleć o przyszłości

Dźwignia	Efekt na koszt	Efekt na UX
<code>--min-instances 0</code>	✔ płacisz tylko za użycie	✗ cold start 30-60s dla L4
<code>--min-instances 1</code>	✗ baseline ~\$940/mies dla Bielika	✔ pierwsza odpowiedź natychmiast
<code>--max-instances N</code>	kontrola budżetu (cap)	przy burst-traffic ograniczenie równoległości
Większy L4 → A100/H100	3-10× droższy	szybsza inferencja, 11B / 70B modele
Mniejszy model (1.5B)	~50% taniej (mieści się na T4 lub CPU)	niższa jakość odpowiedzi
Batch requestów	amortyzacja cold startu	wyższa latencja pojedynczego requestu
Quantization (Q4 zamiast Q8)	~2× mniej VRAM	lekka utrata jakości

Zasada: w GenAI koszt to **GPU × czas** — każdy procent optymalizacji (batch, cache, mniejszy model) zwraca się bezpośrednio.

Pułapki jakości RAG (chunking, embeddings, prompt, ewaluacja, monitoring) — opisane w `troubleshooting.md` w sekcji *Pułapki jakości RAG*.

Co zabieramy ze sobą

- **RAG = prosty wzorzec**: embed → search → prompt → generate
- **Polski LLM jest realny** — Bielik na L4 daje sensowne odpowiedzi
- **Google Cloud** dostarcza klocki (Cloud Run + BigQuery), które się składają w minutach
- **Suwerenność** nie wyklucza nowoczesnego AI — można mieć jedno i drugie
- **~200 linii Pythona** wystarczy, żeby to wszystko spiąć

Appendix

Materiały dodatkowe — dostępne na żywo podczas Q&A

Debug Cloud Run — po pierwsze: logi

Kiedy coś nie działa: zawsze zacznij od logów usługi.

Cloud Run loguje **wszystko**, co Twoja aplikacja wypisuje na `stdout` / `stderr` — plus automatyczne wpisy o requestach, startach kontenera i zdarzeniach platformy.

Dwie ścieżki, te same dane:

-  **Cloud Console (web)** — najszybciej, klika się myszką, filtry w UI
-  **Cloud Shell / `gcloud`** — gdy chcesz tailować albo automatyzować

W warsztacie pracujemy głównie w Cloud Console — wszystko robimy w przeglądarce.

Która ścieżka do czego?

Scenariusz	Rekomendowana ścieżka
"Chcę na oko zobaczyć co się dzieje"	 Cloud Console → zakładka LOGS
"Chcę stream wszystkich errorów w tle"	 <code>gcloud run services logs tail</code>
"Szukam konkretnego stringu w 24h historii"	 Logs Explorer + query
"Koreluję logi z 2-3 usług naraz"	 Logs Explorer z filtrem <code>service_name=~" ... "</code>
"Skrypt / CI ma walidować zawartość logów"	 <code>gcloud logging read --format json</code>
"Post-mortem po incydencie sprzed tygodnia"	 Logs Explorer z Custom Time Range

Reguła kciuka: interaktywne debugowanie → Console. Automatyzacja → gcloud.

gcloud run services logs — podstawy

Dwie komendy dedykowane Cloud Run:

Live tail wybranej usługi (Ctrl+C, żeby zakończyć):

```
gcloud run services logs tail bielik --region $REGION
```

Snapshot ostatnich 50 linii:

```
gcloud run services logs read orchestration-api \  
  --region $REGION --limit 50
```

Komenda	Zachowanie	Kiedy używać
logs tail	interaktywne — stream w czasie rzeczywistym	reprodukujesz błąd i chcesz widzieć live
logs read	snapshot — wypłuka i kończy	szybki przegląd historii, skrypty

logs tail i logs read są wrappersami nad Cloud Logging — zawężają automatycznie do danej usługi w danym regionie.

gcloud logging read — zapytania zaawansowane

Gdy potrzebujesz filtrowania między usługami lub po treści:

```
# Tylko błędy z ostatnich 24h (wszystkie usługi Cloud Run):
gcloud logging read \
  'resource.type="cloud_run_revision" AND severity>=ERROR' \
  --limit 20 --format json
```

```
# Filtr po konkretnej usłudze + tekst w payload:
gcloud logging read \
  'resource.labels.service_name="orchestration-api"
  AND textPayload:"Błąd"' \
  --limit 10
```

```
# Ostatnie requesty z latencją > 2s do Bielika:
gcloud logging read \
  'resource.labels.service_name="bielik"
  AND httpRequest.latency>="2s"' \
  --limit 10
```

Cloud Console — nawigacja do logów

1. Otwórz console.cloud.google.com, zalogowany jako warsztatowy Gmail
2. Menu hamburger ☰ → **Serverless** → **Cloud Run**
3. Lista usług: `bielik`, `embedding-gemma`, `orchestration-api` — **kliknij nazwę debugowanej**
4. Zakładki na szczegółach usługi:
 - **METRICS** — wykresy (request count, latency, errors)
 - **LOGS** ← **tu spędzasz 90% czasu** ★
 - **REVISIONS** — historia deployów

Co masz w zakładce LOGS?

- **strumień live** — nowe linie pojawiają się u góry
- **pole Search** — wpisz słowo, np. `Error`, `403`, `chłoru`
- **filtr Severity**: Default / Info / Warning / **Error** / Critical
- **filtr Time range**: 1h / 24h / Custom
- **kliknięcie w linię** → rozwija pełny JSON (stack trace, payload, user agent)

Na większość problemów wystarczy filtr Severity=Error + Search. Logs Explorer dopiero gdy chcesz korelować.

Logs Explorer — kiedy i jak otworzyć

Potrzebujesz korelować logi z 3 usług naraz albo szukać po custom query? Logs Explorer.

- W zakładce LOGS → **"View in Logs Explorer"** (prawy górny róg)
- Lub menu ☰ → **Logging** → **Logs Explorer**

Struktura UI:

- **Query editor** (góra) — *Logging Query Language*
- **Histogram** — kliknij, żeby zawęzić czas
- **Log fields** (lewy panel) — filtruj jednym kliknięciem

Logs Explorer — przykłady zapytań

```
# Wszystkie błędy z Bielika w ostatniej godzinie  
resource.type="cloud_run_revision"  
resource.labels.service_name="bielik"  
severity ≥ ERROR
```

```
# Logi orkestracji z konkretnym tekstem  
resource.labels.service_name="orchestration-api"  
textPayload:"Błąd przeszukiwania"
```

```
# Wszystkie trzy usługi naraz  
resource.labels.service_name=~"bielik|embedding-gemma|orchestration-api"  
severity ≥ WARNING
```

Save zapytanie — wraca jednym kliknięciem w kolejnym debugu.

Tabela "objaw → log → znaczenie" oraz pełna lista typowych pułapek przeniesione do `troubleshooting.md` (sekcja *Najczęstsze pułapki*).

Co zmieści się w 24 GB L4?

Wagi modelu + KV cache (~4-8k kontekstu):

Model	Q4	Q6_K	Q8	BF16	Na L4?
Bielik 1.5B	~1 GB	~1.5 GB	~2 GB	~3 GB	✓ z zapasem
Bielik 4.5B ★	~3 GB	~4 GB	~5 GB	~9 GB	✓ nasz warsztat
Bielik Minitron 7B	~4 GB	~6 GB	~8 GB	~14 GB	✓ mieści się też na T4
Llama 3.1 8B	~5 GB	~7 GB	~8 GB	~16 GB	✓ nawet BF16
Bielik 11B	~6 GB	~9 GB	~12 GB	~22 GB	✓ BF16 ledwo
Qwen 2.5 14B	~8 GB	~12 GB	~15 GB	28 GB ✗	⚠ tylko kwantyzowane
Llama 3.3 70B	~40 GB ✗	—	—	—	✗ potrzeba A100 80 GB

Reguła praktyczna: Q6_K to sweet spot — jakość bliska Q8, znaczny zapas VRAM na KV cache długich kontekstów (ważne dla RAG z top-k=10+).

Jak to uruchomić w domu? 🏠

Bez chmury — konsumencka karta + **Ollama**. Najtańszy sensowny wybór:

GeForce RTX 3060 12 GB — **~1100–1300 zł** używana (OLX)

- Najtańsze **12 GB VRAM** na rynku — próg wejścia do RAG-a po polsku
- Zmieści **Bielik 4.5B Q8** (~5 GB) z zapasem · nawet **11B Q4** (~6 GB) + kontekst
- **~170 W** — zwykły PC, bez serwerowni

Warsztat (chmura)	Dom
Cloud Run + GPU L4	RTX 3060 12 GB + Ollama
BigQuery Vector Search	Qdrant / pgvector lokalnie
FastAPI orchestration	bez zmian

Na 8 GB (np. 3050) trzeba mocniej kwantyzować i ciąć kontekst — 12 GB to komfort.

NVIDIA L4 — specyfikacja

Parametr	Wartość
Architektura	Ada Lovelace
VRAM	24 GB GDDR6
Bandwidth	300 GB/s
FP16 (Tensor)	485 TFLOPS
TDP	72W

Dostępna w Cloud Run jako `nvidia-l4` — do 1 GPU per instancję.

L4 to GPU *inferencyjne*, nie treningowe. Mały pobór mocy (72W) = tania w cloud.