ESKADRA BIELIKA

Eskadra Bielika

Misja Druga

Poznań, 30.04.2026

Max Małecki

- Tech Lead @ Insly.com - Buduje rozwiązania Ai w ściśle regulowanym środowisku
- 19 lat w IT
  - Od frontend developera
  - Do Tech Leada
  - Od PoC do Produkcji
  - Dzięki agentom umiem kodować nawet w trakcie spotkań.
- Robiłem projekty od Brazylii przez Europę po Chiny.
- Mam Ai na produkcji.

Suwerenne i wiarygodne AI

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

Warsztaty

Instrukcja krok po kroku

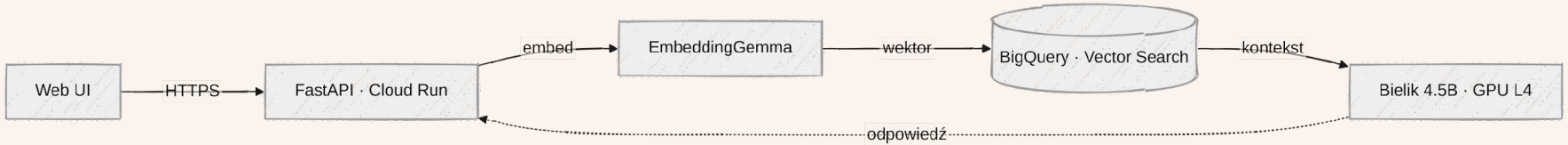
github.com/avedave/eskadra-bielik-misja2

Autor:
Dawid Ostrowski
Senior Program Manager
Developer Relations, Google
linkedin.com/in/avedave

Kupony Google Cloud Credits

https://trygcp.dev/claim/bielik-poznan-304

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.

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

Kluczowe koncepcje

Bielik

Historia

- Architektura Transformers
- Różne modele bazowe (Mistral, Qwen, Nemotron)

Dostępne różne wersje i wielkości

- Bielik Minitron 7B
- **Bielik-4.5B-v3**
- Bielik-1.5B-v3
- Bielik-11B-v2.6

EmbeddingGemma

- Model zoptymalizowany pod kątem tworzenia wektorów
- Embeddingi o stałej długości, głęboki kontekst semantyczny
- Idealny dla RAG
- Mniejszy i szybsze niż pełny LLM
- Łatwa Integracja np. z BigQuery
- Wielojęzyczność

Ollama

- Uruchamiaj modele LLM lokalnie
- Repozytorium LLMów
- Zarządzanie modelami

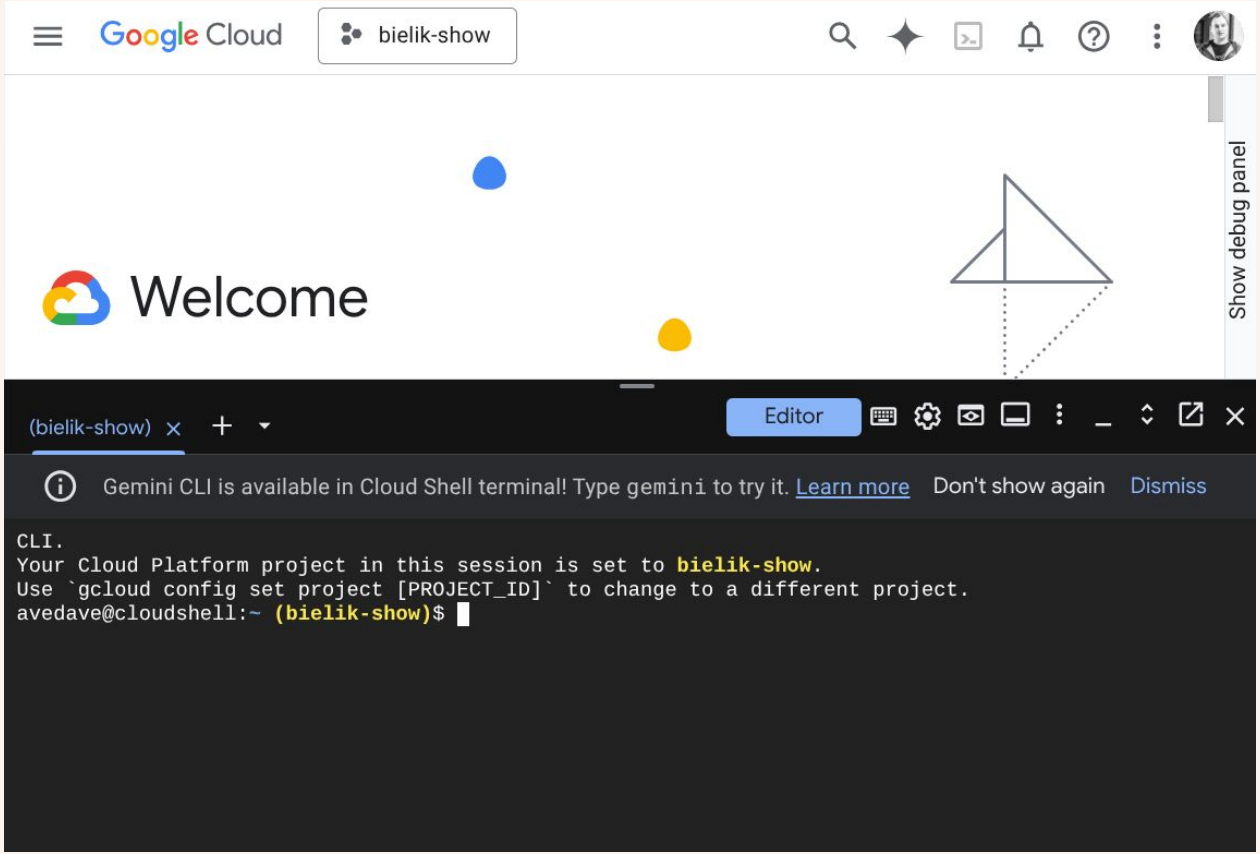
BigQuery

- Hurtownia Danych w Chmurze
- Automatyczne skalowanie pamięci i mocy obliczeniowej
- Natywny typ danych do przechowywania wektorów / embeddingów
- Wbudowany Vector Search - wyszukiwanie semantyczne.
- Generowanie embeddingów w SQL.
- Wywoływanie LLMs bezpośrednio w SQL
- Ekosystem API: SQL, Python (BigFrames), etc.

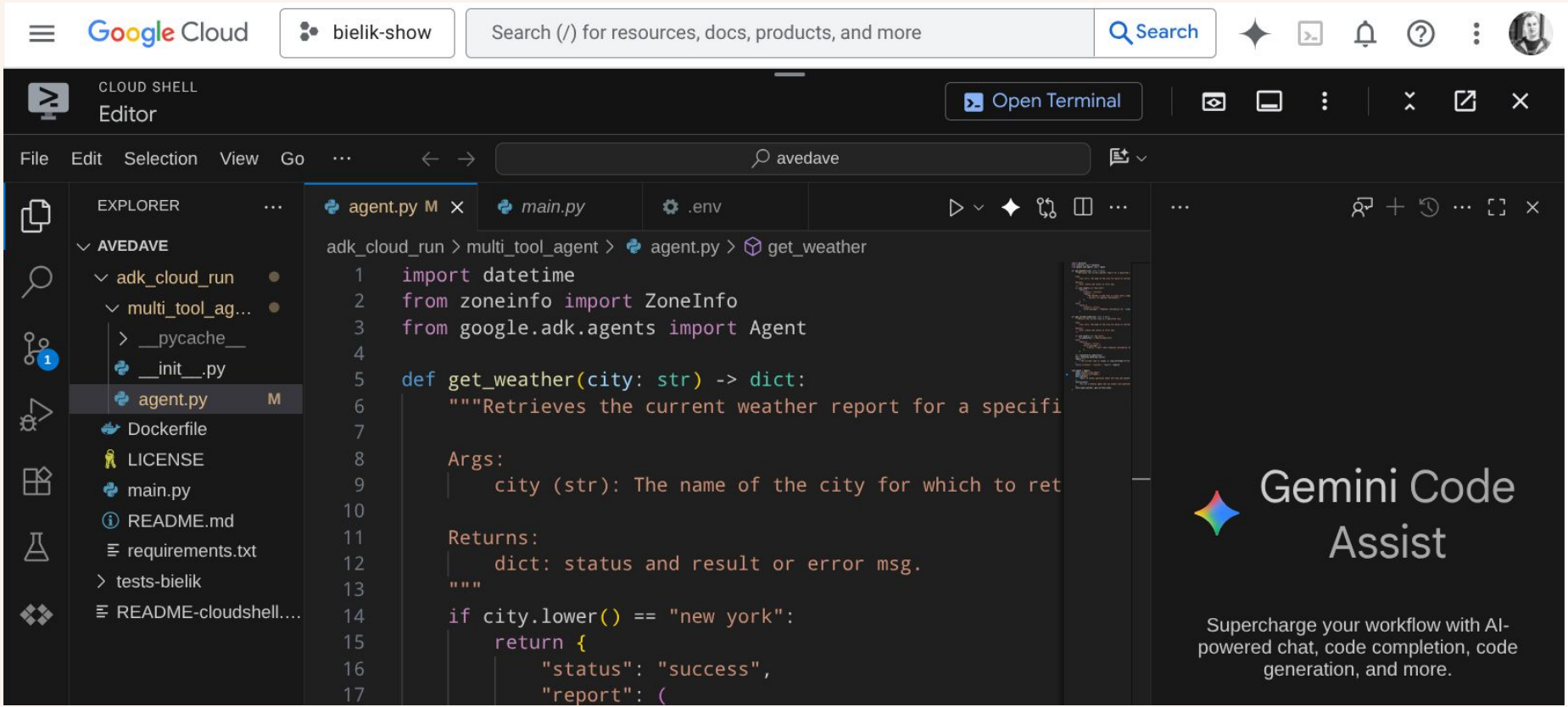
Google Cloud Run

- Szybkie autoskalowanie
- W pełni zarządzana platforma
- Każdy język, każdy framework
- Instancje GPU (L4, RTX PRO 6000)
- 2M darmowych requestów na miesiąc

Google Cloud Shell



Cloud Shell Editor + Code Assist



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 Bielik i	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.

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.

RAG w jednym zdaniu

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

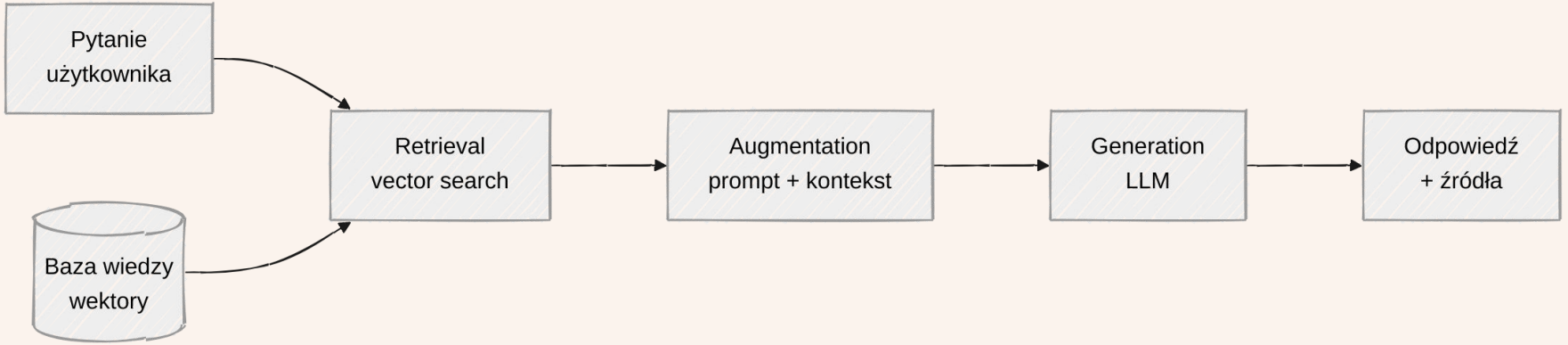
Zysk:

- 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

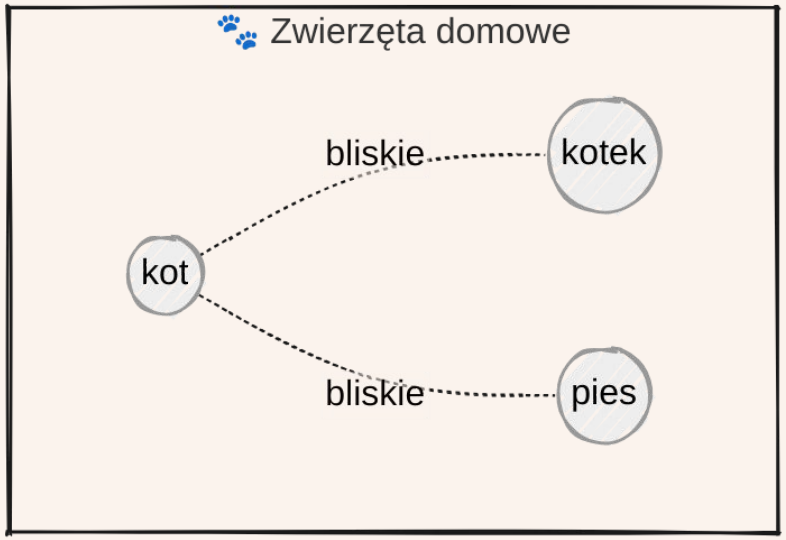


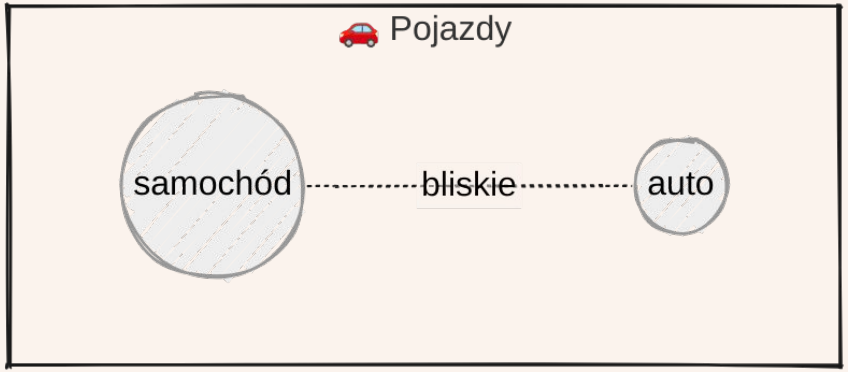
Trzy litery = trzy fazy: Retrieval → Augmentation → Generation.

Embeddings — intuicja

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

Magia: bliskie znaczenia ⇒ 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.

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:

- 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.

Deploye Gotowe?

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 guide

Stack technologiczny

Warstw	Technologi	Rol
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

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 `bielik-11b-v3.0` — tylko zmienić nazwę modelu w `Dockerfile` i `main.py`.

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

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

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+).

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`:

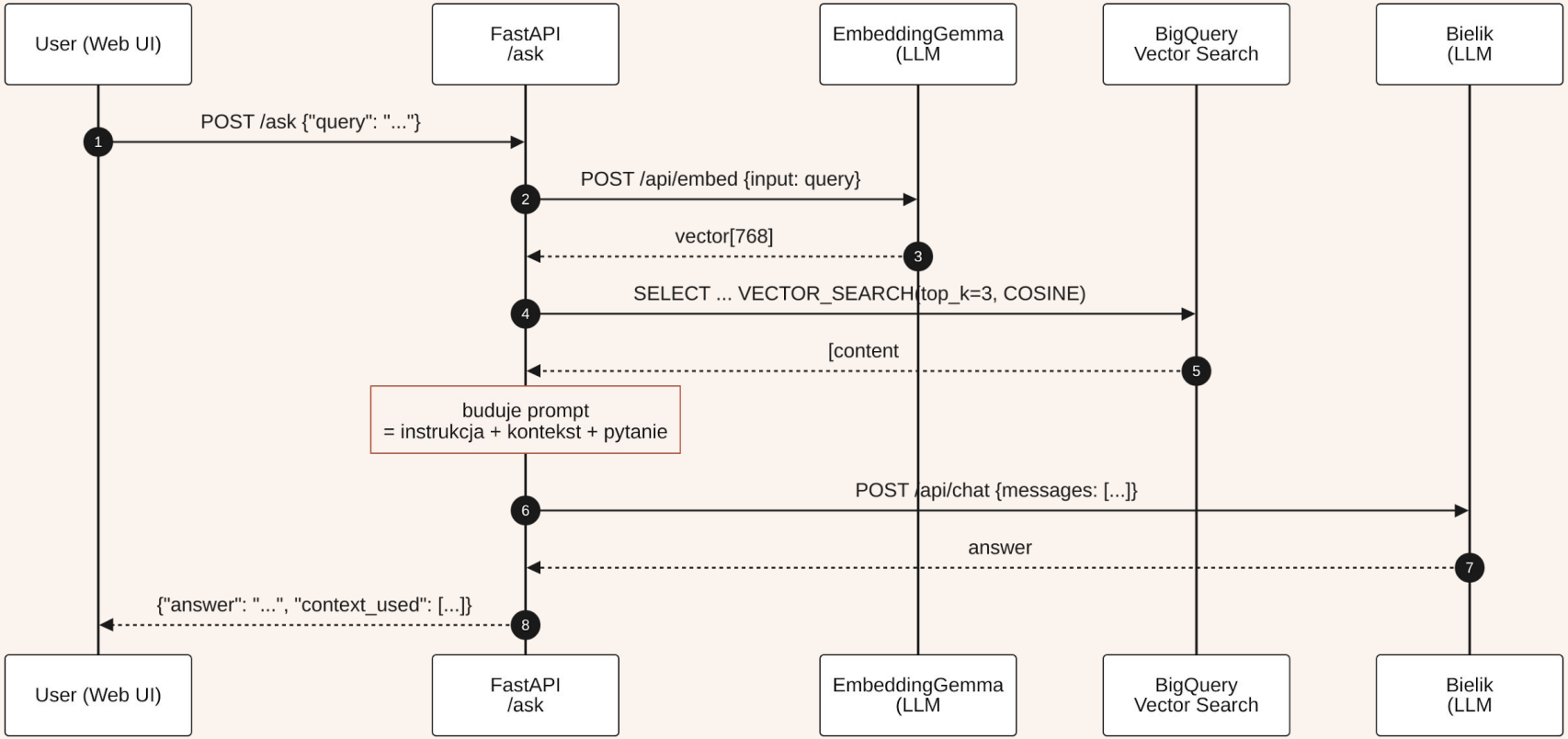
Endpoint	Rola
`GET /`	serwuje prosty Web UI
`POST /ingest`	CSV → embeddings → BigQuery
`POST /ask`	RAG: embedding → vector search → prompt → Bielik
`POST /ask_direct`	baseline: pytanie wprost do Bielika, bez kontekstu

> `/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`.

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
```

Kolejność wdrożenia (1/2) — fundament

1. Przygotowanie projektu GCP + Cloud Shell
2. `source setup_env.sh` + włączenie usług
3. Deploy Bielik LLM na Cloud Run (`llm/cloud_run.sh`)
4. Deploy EmbeddingGemma na Cloud Run (`embedding_model/cloud_run.sh`)
5. Inicjalizacja BigQuery (`vector_store/init_db.py`)

Kolejność wdrożenia (2/2) — orchestration + UI

6. Deploy Orchestration API (`orchestration/cloud_run.sh`)
7. Zasilenie bazy i test RAG przez `curl`
8. Eksploracja API
9. Uruchomienie Web UI i porównanie odpowiedzi

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"	⌨️ `gcloud run services logs tail`
"Szukam konkretnego stringu w 24h historii"	🖱️ Logs Explorer + query
"Koreluję logi z 2-3 usług naraz"	🖱️ Logs Explorer z filtrem `service_name=~"--."`
"Skrypt / CI ma walidować zawartość logów"	⌨️ `gcloud logging read --format json`
"Post-mortem po incydencie sprzed tygodnia"	🖱️ Logs Explorer z Custom Time Range

Reguła: 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 — wypluwa 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
```

Ten sam język (Logging Query Language) działa w Logs Explorer — zapytanie napisane w CLI możesz wkleić do UI.

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`, `chloru`
- 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 orchestracji 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.

Demo — porównanie na żywo

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

`/ask_direct` (sam Bielik)	`/ask` (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.

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	amortyzacja przy 100% GPU
Gemini 2.5 Flash	$0.30	$2.50	budżet, ograniczona jakość PL
Gemini 3 Flash	$0.50	$3.00	lepsza 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
`--min-instances 0`	✅ płacisz tylko za użycie	❌ cold start 30-60s dla L4
`--min-instances 1`	❌ baseline ~$940/mies dla Bielika	✅ pierwsza odpowiedź natychmiast
`--max-instances N`	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.

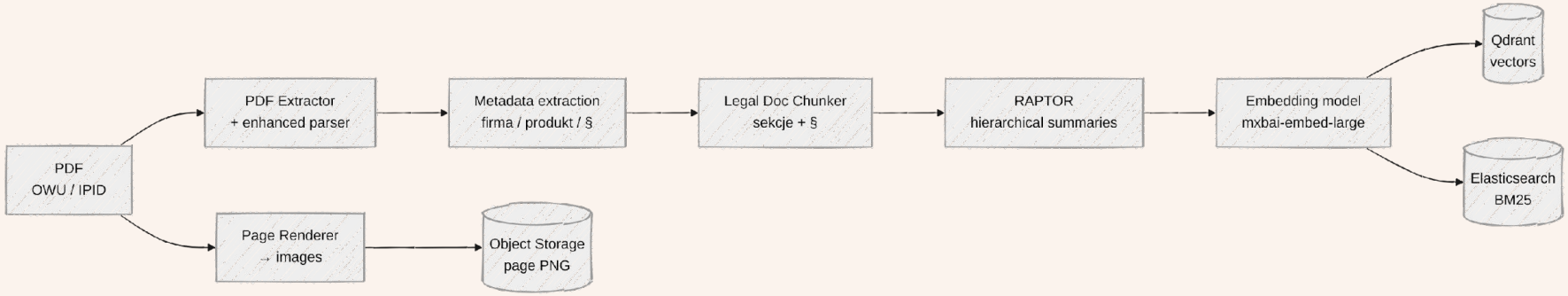
Pułapki jakości RAG (chunking, embeddings, prompt, ewaluacja, monitoring) — opisane w `troubleshooting` w sekcji Pułapki jakości RAG.

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)

A Jak wygląda prawdziy RAG?

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)

Bielik head-to-head

	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

Konkurencja Bielika

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 domenie polskiej — EuroLLM blisko, ale znacznie słabszy na złożonych pytaniach prawnych.

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

https://eskadra.bielik.ai/

Dołącz do Discord