Poznańskie Centrum Superkomputerowo-Sieciowe
Poznan Supercomputing and Networking Center

Towards Reliable RAG Systems for Institutional Knowledge Bases

Lessons from the PCSS Internal Chatbot Pilot

Marcin Wolski & Maciej Łabędzki, March 2026

Introduction & Context

Why we started

- Employees waste time searching for information.
- Frequent operational questions:
    - “Where is the actual delegation request form?”“
    - Why is my access card not working?”“
    - Where can I find the current logo or template?”
- Real time loss and increased load of supporting teams

Business motivation

1. **Reduce** the workload on support teams.
2. Improve employee **efficiency**.
3. Unify best practices and procedures.

- Strengthen knowledge culture and digital consistency.

PCSS currently employs ~470 people

Project goals

Safe Local Chatbot

A secure on-premises chatbot ensuring data privacy and policy compliance.

Use of RAG Architecture

Chatbot responses are grounded in verified internal knowledge.

Extensible System Design

The system is prepared for future integrations, workflow automation and agent-based processes.

Our Journey

Prototype (2024)

Early experiments, first data source, initial retrieval tests.

Pilot (2025)

Knowledge base expansion, automated evaluation, first RAG iterations.

Transition (2025/206)

Evaluation with selected employees, quality improvements, UX enhancements

Production & Integration (2026)

new data sources, intranet integrations, agents.

Deployments 2026+

Deployments in other public institutions

Scale and Complexity

Diverse Data Sources

Multiple heterogeneous sources: web portals, wikis, PDFs, Excel files, emails, departmental documents.

Contributors

~30 contributors, mostly from non-technical departments.

Document Processing Scale

~2000 documents expanded into thousands of chunks.

Users expect consistency

- End users treat the chatbot as a single source of truth.
- Data providers expect AI to deal with data quality
- Consistency requires:
    - information owners,
    - update cycles,
    - version control,
    - standard formats..

To zdjęcie, autor: Nieznany autor, licencja: CC BY-NC-ND

Simple is RAG not enough

Challenges with **Mixed Data**

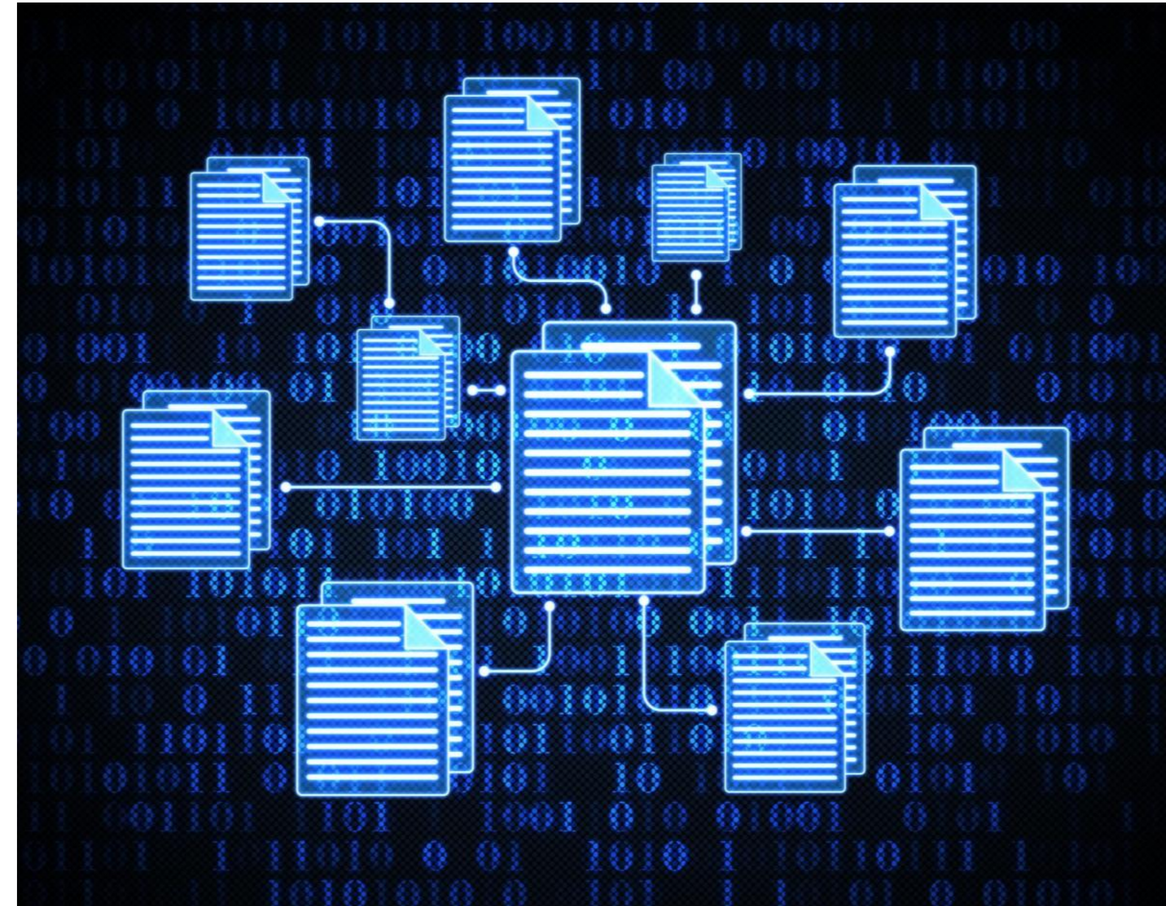
Heterogeneous formats lead to uneven retrieval.

Outdated and Contradictory Information

Old or conflicting documents cause incorrect responses.

Impact of Data Quality

Retrieval struggles with semantically similar but procedurally conflicting documents.

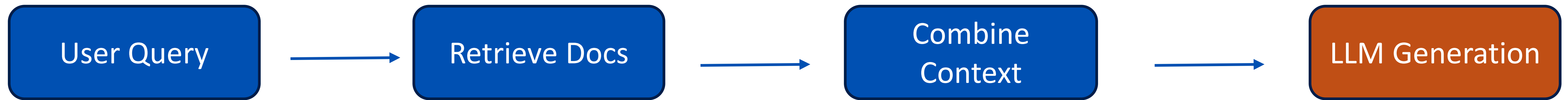


Technical deep dive

02

Technological solutions

RAG — The Technical Essentials



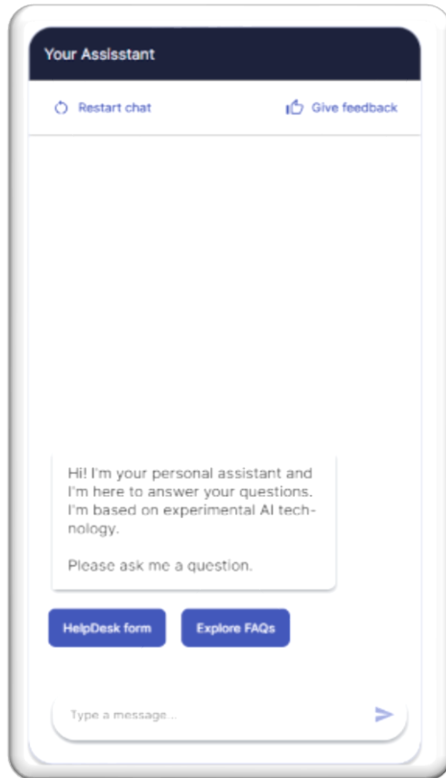
RAG is a **methodology** — an architectural pattern for combining retrieval systems with generative AI. Not a framework. Not a product.

A way of thinking about knowledge-augmented AI.

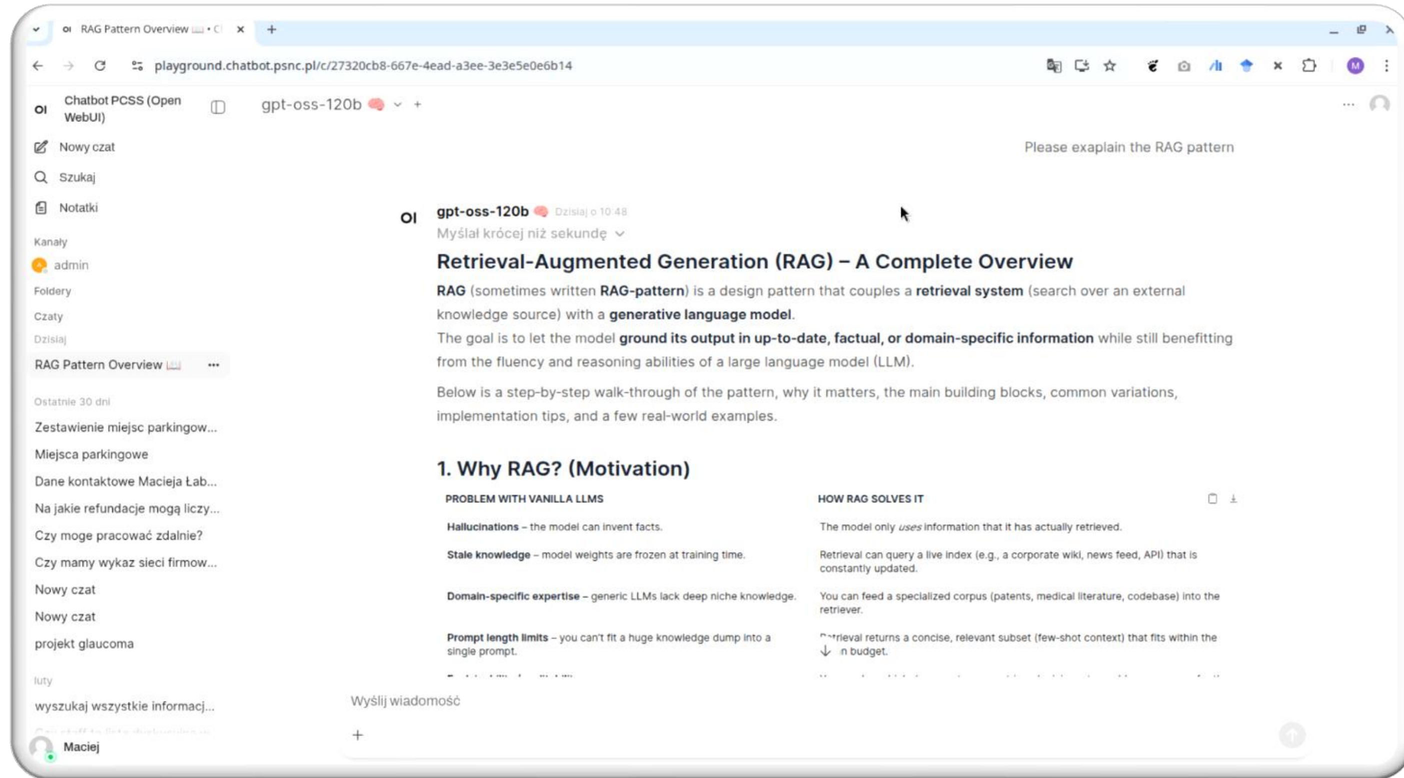
LLMs know how to **reason**, but not what is in your documents (domain knowledge).

RAG connects the two — at query time, not at training time.

User Interface: Build vs. Adopt



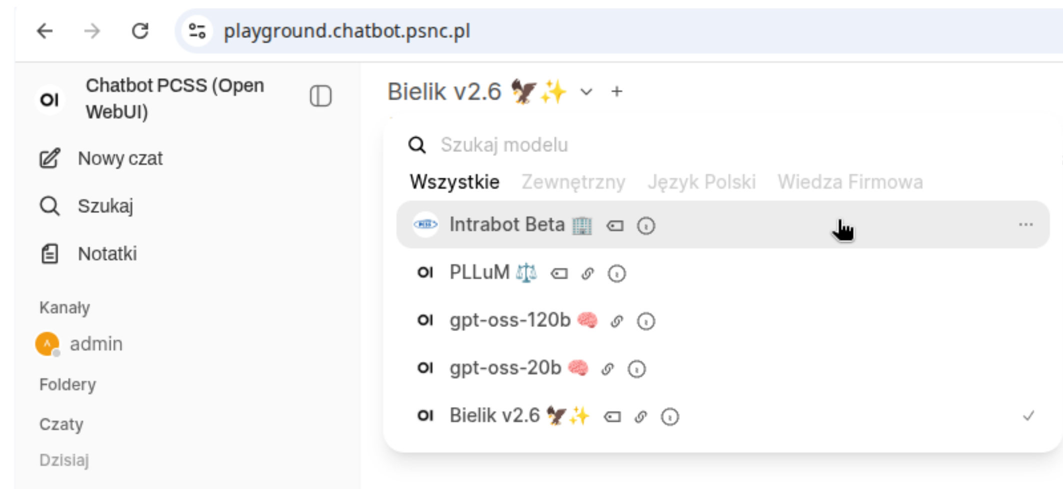
Custom built



Adopted Open WebUI

Why Open WebUI

- **Self-hosted** — data never leaves the institution
- **OpenAI-compatible API** — drop-in integration with many backends
- **RBAC** — user groups, per-model permissions
- **Extensible** — Pipelines plugin framework for custom logic
- **Active development** — weekly releases, large contributor base
- **Multilingual** — including Polish i18n support
- **Model switching -** many models/agents in one place



PCSS RAG with Open WebUI

Open WebUI ships with its own RAG — document upload, vector search, context injection —out of the box. Good for small KBs. We don't use it for big KBs.

Custom RAG vs. Built-in Open WebUI RAG

	Open WebUI RAG	PCSS RAG
Chunking	generic	tuned per data source
Retrieval	single-vector	multi-vector + threshold
Ingestion	manual upload	automated crawlers
Thresholds	fixed	per-deployment
Language	English-first	Polish-optimized

Custom RAG vs. Built-in Open WebUI RAG

Core argument: generic RAG is a reasonable baseline — not an optimum.

Every knowledge base has different structure, language, and retrieval dynamics.

Tuning is not optional — it's the work.

Source Citations: RAG + Open WebUI

Users need to verify answers — **every response must show its sources**.

Two-part implementation:

1. RAG side — metadata attached to every retrieved chunk:

- document title, URL, filename
- retrieval distance score
- last updated timestamp

2. Open WebUI side — displayed via the **Functions** plugin mechanism:

- No fork, no source code modifications
- Open WebUI used as-is from upstream GitHub
- Entire customization through UI or API
- Function intercepts the response and injects formatted source list

Result: user sees the answer + clickable source references — without any changes to the core platform.

Two-Tier Chunking Strategy

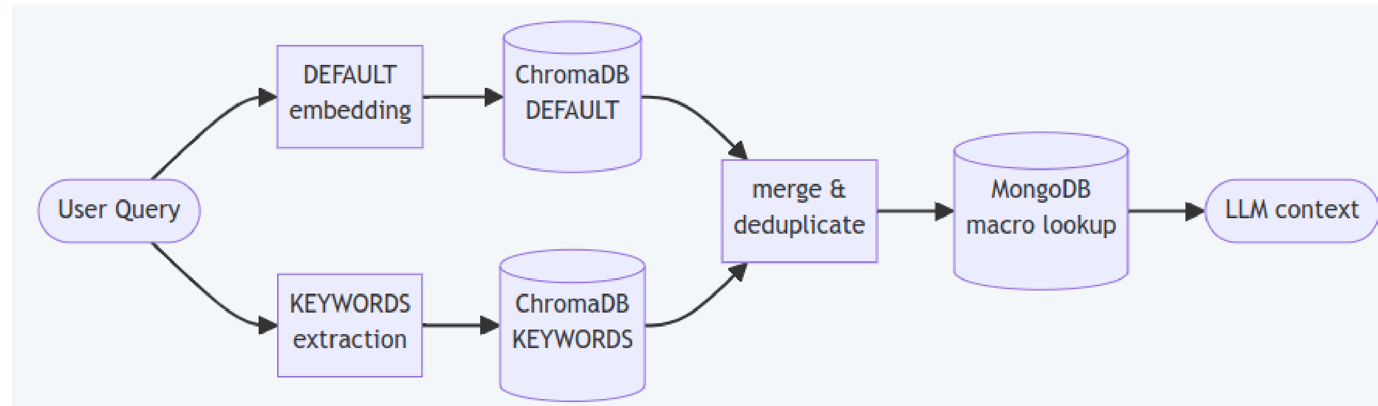
Tier	Storage	Size	Purpose
Macro chunks	MongoDB	~2048 tokens	Full context returned to user
Micro chunks	ChromaDB	~256 tokens	Semantic search (embedded)

Flow at query time:

- Search micro chunks in ChromaDB (fast vector search)
- Map matched micro chunks → parent macro chunk
- Return full macro chunk as context to LLM

Why? Small chunks → better retrieval precision. Full chunks → better answer quality.

Multi-Vector Representation



Flow at query time:

- Search micro chunks in ChromaDB (fast vector search)
- Map matched micro chunks → parent macro chunk
- Return full macro chunk as context to LLM

Why? Small chunks → better retrieval precision. Full chunks → better answer quality.

Retrieval Pipeline

1. **Contextualization**. If chat history exists, LLM reformulates it into a standalone query
2. **Multi-query loop** *(repeated up to 3×, once per query reformulation)*:
    a. **Embedding**. Query encoded with an embedder model
    b. **Semantic search**. ChromaDB queried per vector variant, k=5 candidates each
    c. **Distance threshold filter**. Cosine distance ≤ 0.7, low-confidence matches removed
3. **Deduplication**. Results across all variants and reformulations merged
4. **MongoDB lookup**. Full macro chunks retrieved
5. **Reranking**. LLM scores and reorders results by relevance

Query Contextualization

Turn	Message
1	*"What services does PCSS offer?"*
2	*"What about the prices?"* ← ambiguous
Contextualized	**"What are the prices of PCSS cloud services?"**

An ambiguous follow-up question cannot be searched directly — it must first be resolved against the conversation history.

A dedicated LLM call transforms the ambiguous question into a **standalone query** before retrieval runs.

Result: vector search receives a complete, self-contained question — conversation context is preserved without being passed to the retriever.

Multi-Query Retrieval

Single query often misses relevant documents.

Strategy: `k_requeries = 3`

User: *"How to request VPN access?"*

	Query sent to retrieval
Q1	How to request VPN access?
Q2	VPN setup procedure for employees
Q3	Remote access configuration request

Trade-off: 3× retrieval latency

Gain: more comprehensive context coverage

Configurable: can be disabled per deployment

Evaluation Framework

Based on **DeepEval** with custom Polish language adaptations (evaluation prompts translated to Polish)

Test data: curated Q&A pairs provided by **selected PCSS evaluators** — domain experts who know what correct answers look like

Approach: **LLM as judge**

Metric	What it measures
Faithfulness	Is the answer grounded in the retrieved context?
Answer Relevancy	Does the answer address the question?
Correctness	Does the answer match the reference answer? (requires human-curated ground truth)

From Naive RAG to Adaptive RAG

The mechanisms we've described transform a simple retrieve-and-generate loop into a system that **adapts to context, data, and user behavior**:

Mechanism	What it adds
Query contextualization	Conversation awareness
Multi-query retrieval	Robustness to phrasing variation
Asymmetric embeddings	Correct use of embedding model's semantic space
Distance threshold	Self-restraint — knows when not to answer
Two-tier chunking	Precision in search, completeness in response
Multi-vector representation	Hybrid semantic + keyword coverage
LLM reranking	Relevance ordering of retrieved context
Token budget management	Reliable context window utilization
Source citations	Transparency — every answer traceable to source
Per-deployment configuration	Domain adaptation

Naive vs. Advanced — When to Use Which?

When naive RAG is enough:

- Small, homogeneous knowledge base (< a few hundred docs)
- Single language, well-structured source documents
- Low stakes — occasional imprecision acceptable
- Rapid prototype / proof of concept

When you need advanced RAG:

- Large, heterogeneous corpus across multiple source types
- Morphologically complex language (Polish, Finnish, Arabic…)
- Institutional deployment — users trust the answers
- Multi-tenant, multiple user groups with different needs

Future Direction: Multi-Agent Architecture

The next evolution beyond a single RAG pipeline:

Core idea: replace one general-purpose retriever with a **network of specialized agents**, each equipped with a **RAG pipeline optimized for its own data source**.

Orchestrator routes the user query to the right agent — or combines answers from multiple agents.

Why this matters: a single RAG tuned for everything is tuned for nothing. Each agent owns its chunking strategy, embedding model, retrieval thresholds, and prompt — independently.

Agent	Own RAG optimized for…
Mailing list agent	Email thread structure, sender context, temporal ordering
Documentation agent	Technical specs, structured manuals, precise terminology
HR knowledge agent	Policy documents, procedures, regulatory language
Web search agent	Live external content, broad coverage over precision

Poznańskie Centrum Superkomputerowo-Sieciowe
Poznan Supercomputing and Networking Center