# PoliRag: Retrieval-Augmented Generation over PoliformaT

0xbiel

January 30, 2026

**Abstract**

PoliRag is a Retrieval-Augmented Generation (RAG) system designed to turn a user's PoliformaT course content (subjects, announcements, lesson pages, teaching guides, and downloaded resources like PDFs) into a searchable personal knowledge base. At query time, PoliRag retrieves the most relevant snippets from that knowledge base and provides them as grounded context to a language model, improving factuality and reducing hallucinations compared to prompting without retrieval.

## 1 What PoliRag is

PoliRag combines three components:

1. **Ingestion (scraping + extraction).** Content is collected from PoliformaT via a headless browser flow and via authenticated HTTP requests using imported session cookies. Downloaded resources (e.g., zip bundles of course files) are unpacked and PDFs are extracted to clean text.
2. **Indexing (embeddings + vector store).** Each document is embedded into a fixed-length vector using a local embedding model. Documents, metadata, and embeddings are stored on disk in a compact serialized vector index.
3. **Retrieval (semantic search + snippets).** A user query is embedded, compared against stored document embeddings using cosine similarity, and the top matches are turned into concise snippets suitable for LLM context.

The design goal is a *personal, offline-first* RAG workflow: embeddings are computed locally, and the on-disk index can be rebuilt or re-embedded when the embedding model changes.

## 2 How I'm implementing RAG (architecture)

### 2.1 Embedding model (local gguf, embedded into the binary)

The embedder uses `llama_cpp_2` and packages the embedding model directly into the executable using `include_bytes!`. At runtime the gguf bytes are written to a temporary file and loaded by the llama backend. Embeddings are enabled via context parameters.

Because long inputs can exceed the embedding context window, text is chunked by approximate token budget and the final representation is computed by averaging chunk embeddings followed by $\ell_2$ normalization.

Listing 1: Embedding model setup and chunked embedding.

```
use anyhow::{Context, Result};
use std::sync::Arc;
use std::io::Write;
use tempfile::NamedTempFile;

use llama_cpp_2::context::params::LlamaContextParams;
use llama_cpp_2::llama_backend::LlamaBackend;
use llama_cpp_2::model::LlamaModel;
use llama_cpp_2::model::params::LlamaModelParams;
use llama_cpp_2::model::AddBos;

const MODEL_BYTES: &[u8] = include_bytes!("../../embeddinggemma-300m-Q4_0.gguf");

#[derive(Clone)]
pub struct EmbeddingModel {
    state: Arc<LlamaState>,
    context_params: LlamaContextParams,
}

impl EmbeddingModel {
    pub fn new() -> Result<Self> {
        let backend = Arc::new(LlamaBackend::init()?);

        let mut temp_file = NamedTempFile::new()?;
        temp_file.write_all(MODEL_BYTES)?;
        temp_file.flush()?;

        let model_params = LlamaModelParams::default().with_n_gpu_layers(999);
        let model = LlamaModel::load_from_file(backend.as_ref(), temp_file.path(), &
    model_params)
            .context("Failed to load Llama model")?;

        let ctx = LlamaContextParams::default()
            .with_embeddings(true)
            .with_n_batch(2048)
            .with_n_ubatch(2048);

        Ok(Self { state: Arc::new(LlamaState { backend, model: Arc::new(model),
```

```
    _temp_file: Arc::new(temp_file) }),
                  context_params: ctx })
    }

    pub async fn embed(&self, text: &str) -> Result<Vec<f32>> {
        // chunk -> embed each chunk -> average -> L2 normalize
        // (implementation omitted here for brevity)
        Ok(vec![])
    }
}
```

### 2.2 Vector index (documents + metadata + embeddings)

The vector store is a simple persistent in-memory index:

- each `Document` holds `id`, `content`, `embedding`, `metadata`, and `user_id`;
- the full `VectorIndex` is serialized to disk using `bincode`;
- multi-user isolation is enforced during retrieval by filtering on `user_id`.

This prioritizes simplicity and debuggability over advanced ANN structures; it is sufficient for medium-sized personal datasets and makes it easy to re-embed and inspect stored values.

Listing 2: Document model and persistent vector index.

```
#[derive(Serialize, Deserialize, Clone, Debug)]
pub struct Document {
    pub id: String,
    pub content: String,
    pub embedding: Vec<f32>,
    pub metadata: HashMap<String, String>,
    pub user_id: String,
}

#[derive(Serialize, Deserialize, Default)]
struct VectorIndex {
    documents: Vec<Document>,
}
```

### 2.3 Retrieval (cosine similarity + snippet extraction)

At query time PoliRag:

1. embeds the query;

2. scores each candidate document with cosine similarity;
3. sorts by score, applies a relevance threshold, and returns the top-$k$;
4. extracts a compact snippet by finding windows with high query-word density.

Listing 3: Semantic search and cosine similarity.

```rust
pub async fn search(&self, query: &str, user_id: &str, top_k: usize)
    -> anyhow::Result<Vec<(Document, f32)>>
{
    let query_embedding = self.embedder.embed(query).await?;
    let db = self.db.lock().unwrap();

    let mut scores: Vec<(Document, f32)> = db.documents.iter()
        .filter(|d| d.user_id == user_id)
        .map(|d| (d.clone(), cosine_similarity(&query_embedding, &d.embedding)))
        .collect();

    scores.sort_by(|a, b| b.1.partial_cmp(&a.1).unwrap());
    scores.truncate(top_k);
    Ok(scores)
}

fn cosine_similarity(a: &[f32], b: &[f32]) -> f32 {
    let dot: f32 = a.iter().zip(b).map(|(x, y)| x * y).sum();
    let na: f32 = a.iter().map(|x| x * x).sum::<f32>().sqrt();
    let nb: f32 = b.iter().map(|x| x * x).sum::<f32>().sqrt();
    if na == 0.0 || nb == 0.0 { 0.0 } else { dot / (na * nb) }
}
```

## 3 Ingestion: PoliformaT scraping and resource processing

PoliRag collects content through a headless login flow and subsequent scraping:

- **Headless authentication:** a headless Chrome session navigates the login portal, detects input fields robustly, submits credentials, then exports cookies (e.g., `JSESSIONID`).
- **Subject discovery:** the portal is scanned for subject links under `/portal/site/...`.
- **Tool scraping:** within each subject, the scraper discovers relevant tools (announcements, lessons, resources, teaching guide) and accumulates extracted text.
- **Resource extraction:** downloaded zip bundles are unpacked; PDFs are extracted to text and normalized (ligatures, punctuation, whitespace).

The output of ingestion is a set of documents that are then added to the vector index, tagged with metadata such as `type` (subject, announcements, guia docent, pdf, etc.) and `filename` where applicable.

## 4 Why this is RAG (and not just search)

PoliRag is RAG because retrieval is used to *augment generation*: the retrieved snippets are not the final answer, but rather the evidence injected into the LLM prompt so the model can answer with citations and course-specific details (deadlines, grading rules, announcements), while remaining anchored to real sources.

## 5 Terminal UI (Ratatui)

PoliRag is driven by a terminal UI built with `ratatui`. The UI is optimized for streaming LLM output and for displaying retrieved context cleanly.

A key piece is a lightweight markdown renderer that converts assistant output into `Vec<Line <'static>>` for display. It supports:

- **Collapsible thinking blocks:** content inside `<think>...</think>` is rendered in a distinct style and can be toggled collapsed/expanded.
- **Code fences:** triple-backtick blocks are detected and rendered with a dedicated code style.
- **Tables:** pipe-delimited rows are buffered and rendered into a boxed table with alignment inferred from the separator row.
- **Inline styles:** basic inline formatting such as `` `code` `` and `**bold**` is mapped to terminal styles.
- **Wrapping:** all content is wrapped to the available viewport width, with indentation heuristics for lists.

Listing 4: Ratatui markdown rendering with thinking blocks, code fences, and tables.

```rust
use ratatui::style::{Color, Modifier, Style};
use ratatui::text::{Line, Span};

pub fn render_markdown(text: &str, max_width: usize, thinking_collapsed: bool) -> Vec<
    Line<'static>> {
    let mut lines = Vec::new();
    let mut in_code_block = false;
    let mut table_lines = Vec::new();

    // Split <think>...</think> from main content (supports streaming / missing close tag
    )
    let (thinking_content, main_content) = if let Some(start) = text.find("<think>") {
        if let Some(end) = text[start..].find("</think>") {
            let think_end = start + end + 8;
            (Some(&text[start + 7..start + end]), &text[think_end..])
        } else {
```

```
            (Some(&text[start + 7..]), "")
        }
    } else {
        (None, text)
    };

    // Render (optional) collapsible thinking block
    if let Some(think) = thinking_content {
        let header_style = Style::default().fg(Color::Yellow).add_modifier(Modifier::BOLD
    );
        let icon = if thinking_collapsed { ">" } else { "v" };
        lines.push(Line::from(vec![Span::styled(format!(" {} Thinking Process", icon),
    header_style)]));
        if !thinking_collapsed {
            let think_style = Style::default().fg(Color::DarkGray).add_modifier(Modifier
    ::ITALIC);
            for line in think.lines() {
                lines.push(Line::from(Span::styled(line.to_string(), think_style)));
            }
        }
    }

    for line in main_content.lines() {
        if line.trim().starts_with("```") {
            in_code_block = !in_code_block;
            lines.push(Line::from(Span::styled(line.to_string(), Style::default().fg(
    Color::DarkGray))));
            continue;
        }

        if in_code_block {
            lines.push(Line::from(Span::styled(line.to_string(), Style::default().fg(
    Color::Yellow))));
            continue;
        }

        if line.trim().starts_with('|') {
            table_lines.push(line.to_string());
            continue;
        }

        // (flush buffered table when leaving table mode)
        // (headers / separators / wrapping omitted here for brevity)

        lines.push(Line::from(Span::raw(line.to_string())));
    }

    lines
}
```

# 6 Configuration, CLI, and sync ops

## 6.1 Configuration (`config.rs`)

PoliRag persists user settings and cached credentials in a JSON config file stored under the OS application data directory (e.g., via `dirs::data_dir()`). Credentials are stored as an encrypted blob (XOR + base64) so they are not plain-text on disk.

The configuration also tracks which LLM backend to use (LM Studio local endpoint vs OpenRouter), plus the most recently selected model.

Listing 5: Provider selection and config model (excerpt).

```
#[derive(Serialize, Deserialize, Clone, Default, PartialEq)]
pub enum LlmProvider {
    #[default]
    LmStudio,
    OpenRouter,
}

impl LlmProvider {
    pub fn base_url(&self) -> &'static str {
        match self {
            LlmProvider::LmStudio => "http://localhost:1234/v1",
            LlmProvider::OpenRouter => "https://openrouter.ai/api/v1",
        }
    }
}

#[derive(Serialize, Deserialize, Default)]
pub struct Config {
    pub last_model: Option<String>,
    pub cached_credentials: Option<EncryptedCredentials>,
    pub llm_provider: LlmProvider,
    pub openrouter_api_key: Option<String>,
    pub openrouter_model: Option<String>,
}
```

## 6.2 Entrypoint and commands (`main.rs`)

The application exposes a small CLI with `clap`:

- **Menu:** the default interactive terminal UI.
- **Sync:** a headless scrape + index build suitable for cron/automation.
- **ExtractPdf:** an internal subcommand used to isolate PDF text extraction into a subprocess.

On startup, PoliRag initializes logging, loads the vector index path from the global config directory, and initializes the RAG, scrapper, and LLM client.

Listing 6: CLI commands (excerpt).

```
#[derive(Subcommand, Clone)]
enum Commands {
    Sync,
    Menu,
    #[command(hide = true)]
    ExtractPdf { path: String },
}
```

### 6.3 Sync pipeline (`ops.rs`)

The sync operation is responsible for ensuring authentication (cached credentials or environment variables), clearing prior state, scraping subjects, extracting resources, and finally adding documents to the vector index.

Listing 7: Sync flow (excerpt).

```
pub async fn run_sync(rag: Arc<rag::RagSystem>, poliformat: Arc<scrapper::
    PoliformatClient>)
    -> anyhow::Result<()>
{
    if !poliformat.check_connection().await.unwrap_or(false) {
        // try cached credentials; fall back to env vars
        // perform headless login (blocking)
    }

    rag.clear()?;
    let subjects = poliformat.get_subjects().await?;
    let detailed = poliformat.scrape_subject_content(subjects).await?;

    for (sub, dir_path) in detailed {
        // read summary.md, process PDFs, add documents with metadata
        // rag.add_document(...).await?;
    }

    Ok(())
}
```

## 7 Limitations and next steps

- **Index scalability:** the current approach is linear scan; a future upgrade could add ANN (HNSW) for very large corpora.
- **Chunking quality:** word-based chunking is a heuristic; sentence/semantic chunking often improves retrieval.

- **Grounding UX:** adding explicit per-snippet source identifiers and timestamps would make answers easier to trust.