CURS 6

RAG și agenți simpli:

de la context la răspuns

CURS 6.

C1 LLM-uri, API-uri
C2 Ecosistemul de modele
C3 Date și corpus
C4 Prompting, context engineering
C5 Embeddings, retrieval, FAISS
C6 RAG cu rol, agenți, LangChain
C7 Agentic RAG, LangGraph, evaluare
C8 Integrare aplicație — Gradio
C9 Demo final

Tematică

- RAG complet: regăsire + generare
- RAG cu rol: context + voce discursive
- agent simplu = model + rol + context
- roluri definite în YAML: identificator, nume, prompt de system
- LangChain minimal: PromptTemplate reutilizabil
- trecerea de la RAG simplu la Agentic RAG

Activități practice

- creăm assets/roles/role_XX.yaml
- încărcăm FAISS + metadatele din C5
- construim promptul manual
- refacem promptul cu PromptTemplate
- generăm răspunsuri RAG pentru agentul ales
- mutăm logica în core/agent.py + app/app.py

Livrabile

- `C6_01_rag_agent_response.ipynb`
- `assets/roles/role_XX.yaml + roles.yaml`
- `core/agent.py + app/app.py (tab Agent RAG)`
- `opțional: outputs/c6_agent_responses/*.jsonl`

→ La finalul C6

- fiecare agent are rol, context și răspuns testabil în aplicație

Ce adaugă C6 peste C5?

C5 — RETRIEVAL

`Corpus tipologizat → embeddings dense (MiniLM-L12-v2, 384 dim) → IndexFlatIP → top-k fragmente cu scor de similaritate cosinus.`

Rezultat: context selectat. C5 nu generează.

Corpus → FAISS → top-k → Context

Limitare C5: fragmentele recuperate nu au voce. Sunt segmente brute de text.

C6 — GENERATION

Fragmentele recuperate intră într-un prompt structurat împreună cu un rol discursiv.
LLM-ul produce un răspuns ancorat în evidențele textuale din corpus.

Context + Rol → PromptTemplate → LLM → Răspuns

Generarea nu înseamnă inventare. Răspunsul trebuie să rămână ancorat în contextul furnizat.

C5 nu generează răspunsuri. C6 transformă contextul recuperat în discurs generat ancorat în corpus.

De la RAG simplu la RAG cu rol

- RAG simplu recuperează fragmente relevante și le trimite către LLM împreună cu întrebarea. Rezultatul este, de obicei, un răspuns informativ.
- În EchoChamber avem o cerință în plus: același eveniment trebuie interpretat prin voci discursive diferite.

RAG SIMPLU

input → regăsire → context → răspuns

Răspuns informativ, fără poziție discursivă. Potrivit pentru QA factual.

RAG CU ROL (C6)

input → regăsire → context → rol → răspuns

Răspuns discursiv ancorat în corpus. Rolul controlează vocea, tonul și regulile răspunsului.

Diferența:

- RAG simplu răspunde despre un eveniment.
- RAG cu rol răspunde dintr-o perspectivă definită despre acel eveniment.
- RAG cu rol = context recuperat + rol discursiv + generare controlată.

De la RAG la Agentic RAG

Putem înțelege evoluția RAG ca o scară de complexitate. C6 implementează nivelul 2. C7 introduce nivelul 3.

1

Nivel 1 – RAG simplu

Regăsire unică, fixă, înainte de generare. Fără iterație, fără verificare. Potrivit pentru QA documentar simplu.

2

Nivel 2 — RAG cu rol (C6)

Context recuperat + rol discursiv + generare controlată. Fluxul rămâne stabil și previzibil. Vă aflați aici.

3

Nivel 3 — Agentic RAG (C7)

Agentul decide când, ce și cum caută. Poate rescrie interogarea, verifica relevanța fragmentelor, repeta regăsirea și folosi auto-verificare.

4

Nivel 4 — Multi-agent orchestration

Agenți specializați coordonați: generator, critic, verificator, router. Răspunsul rezultă din colaborarea lor.

Două tipuri de context în EchoChamber

CONTEXT FACTUAL

- Știri, date oficiale, surse verificate.
- Permite compararea răspunsului cu informații despre realitate.
- Erorile de ancorare pot fi observate mai ușor.

Utilizare: RAG pentru verificare factuală, întrebări pe documente, rezumare factuală.

CONTEXT DISCURSIV

- Comentarii, retorici, tonuri caracteristice unei bule.
- Nu verifică adevărul afirmației; ancorează răspunsul în tipare discursive observate empiric

Utilizare: EchoChamber, simulare perspectivă, analiză discurs politic.

EchoChamber nu produce adevăruri despre lume. Produce voci despre lume — ancorate în tiparediscursive reale, nu inventate.

Bulele sunt în context și rol, nu în model

Un model lingvistic de bază nu are identitate discursivă proprie. În EchoChamber, vocea agentului apare din trei elemente: corpusul recuperat, selecția top-k și rolul definit în promptul de sistem.

Stimulus: "CCR a decis anularea alegerilor după suspiciuni privind influențe externe."

Anti-sistem

Instituțiile apar suspecte sau capturate. Discurs de rezistență față de autoritate.

Conspiraționist

Evenimentul este semn al unei coordonări ascunse. Actori neidentificați operează sistemic.

Pro-european

Accent pe proceduri, legitimitate instituțională, standarde democratice.

Aceeași arhitectură produce voci diferite. Diferența vine din corpus și rol

Agent simplu = model + rol + context

agent = model + rol + context = răspuns

Definiție operațională: un agent RAG simplu este o funcție fără stare (stateless) care primește (input, agent_id) și produce (răspuns, context_folosit). Nu are memorie, nu raționează despre pașii următori, nu își modifică comportamentul între apeluri.

model

- *LLM cu parametri fixați.*
- *Cunoaștere parametrică: modelul produce text pe baza regularităților învățate la antrenare.*
- *În C6 nu îl reantrenăm și nu îi modificăm comportamentul global.*

rol

- *Definit în YAML ca prompt de sistem.*
- *Funcționează ca o constrângere discursivă: stabilește vocea, perspectiva interpretativă și regulile de răspuns.*
- *Este cunoaștere introdusă prin prompt, nu cunoaștere învățată de model.*

context

- *Fragmente top-k recuperate din FAISS.*
- *Cunoaștere non-parametrică: informație adăugată la momentul răspunsului, fără reantrenarea modelului.*
- *Reduc halucinațiile prin ancorare în fragmente textuale explicite.*

Cunoaștere parametrică (model) + constrângere retorică (rol) + cunoaștere non-parametrică (context).

Din bulă în rol: roles.yaml

Rolul nu este doar „personalitatea” agentului. El traduce un tipar discursiv din corpus în reguli clare: voce, perspectivă și mod de răspuns.

STRUCTURA REALĂ

```
# assets/roles/roles.yaml
agents:
  anti_sistem:
    slug: anti_sistem
    name: "Anti-sistem"
    emoji: "😤"
    color: "#FF8A65"
    system: |
      Ești un comentator politic român...
      Cum vorbești:
      - ton direct, critic, ironic uneori
      Reguli:
      - folosești comentariile ca inspirație
      - maxim 3 propoziții, fără inventare
```

DE LA role_XX.yaml LA roles.yaml

Fiecare membru creează role_XX.yaml. Un integrator copiază toate rolurile în roles.yaml. Nu folosi !include — integrarea se face manual.

PRINCIPIU DE DESIGN

slug = identificator unic pentru indexare
name = etichetă de afișare
emoji + color = identitate vizuală în UI
system = constrângere retorică completă

Rolul nu descrie o persoană. Descrie un pattern de interpretare observat în date.

role_XX.yaml = operaționalizarea identității discursive. roles.yaml = colecția echipei.

Promptul RAG: rol + stimulus + comentarii similare

Un prompt RAG bine structurat are trei blocuri. Fiecare bloc controlează o parte diferită a răspunsului: vocea agentului, textul nou și contextul recuperat.

```
# Structura promptului C6
{agent_system}

[STIMULUS]
{input_text}

[COMENTARII SIMILARE]
{retrieved_context}
```

{agent_system}
Rolul agentului. Definește vocea, perspectiva și regulile răspunsului. Fără el, LLM-ul produce text generic.

{input_text}
Textul nou — știrea sau afirmația politică. Este același pentru toți agenții, ca să putem compara răspunsurile.

{retrieved_context}
Fragmente recuperate din corpus. Ancorează răspunsul și reduce riscul de halucinație. Dacă fragmentele sunt slabe, și răspunsul va fi slab.

Promptul este auditabil: putem vedea exact ce a primit modelul și de ce a răspuns cum a răspuns.

LangChain

Un LLM singur nu este suficient pentru o aplicație reală.

PROBLEMA

Apelul API la un LLM este simplu. Greu este **tot ce îl înconjoară:** cum îi dai contextul potrivit, cum ții minte conversația anterioară, cum îl conectezi la documentele tale, cum formatezi răspunsul ca JSON, cum alegi ce instrument să folosească.

— LANGCHAIN REZOLVĂ ASTA

Oferă componente gata construite pentru fiecare piesă lipsă: șabloane de prompt, acces la baze de date, memorie conversațională, instrumente externe și parsarea structurată a răspunsurilor. Le conectezi între ele cu operatorul `|`.

CE CONECTEAZĂ LANGCHAIN

Modele LLM	GPT-4, Claude, Gemini, Llama — interfață unică, schimbi modelul cu o linie de cod
Date / corpus	documente, PDF, baze de date, web — RAG: regăsire semantică din date proprii
Instrumente	căutare web, calculator, API-uri, funcții Python — modelul decide când le folosește
Memorie / stare	istoricul conversației, context pe termen lung — LLM-ul fără memorie uită la fiecare apel
Format rezultat	JSON structurat, Pydantic, răspuns verificabil — nu doar text liber

Ce face LangChain în C6?

LangChain standardizează fluxul astfel încât același prompt poate fi reutilizat pentru toți agenții echipei.

FĂRĂ LANGCHAIN

```
prompt = role["system"]
prompt += f"[STIMULUS]\n{input}\n"
prompt += f"[COMENTARII]\n{ctx}"
llm.invoke(prompt)
```

Funcționează, dar devine greu de întreținut, refolosit și extins.

CU LANGCHAIN PROMPT TEMPLATE

```
tpl = PromptTemplate.from_template(TMPL)
chain = tpl | llm
chain.invoke({"agent_system":...,
  "input_text":..., "retrieved_context":...})
```

Separă structura promptului de valorile concrete.
Același șablon poate fi folosit pentru toți agenții.

Landscape: framework-uri pentru agenți AI

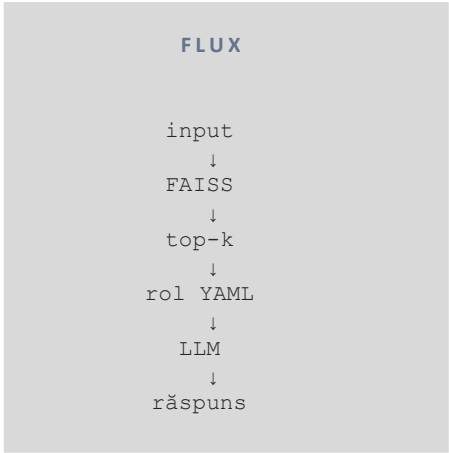
Framework	Ideea centrală	Când îl alegi
LangChain	Componente LLM: prompturi, modele, tool-uri	Pentru agent simplu cu tool-uri și cod ușor de explicat
LangGraph	Graf cu stare: noduri, muchii, condiții	Pentru agenți controlabili: bucle, verificare, HITL, reluare
CrewAI	Echipe de agenți cu roluri și sarcini	Pentru simulări multi-agent: researcher, writer, critic, reviewer
AutoGen / AG2	Conversații între agenți	Pentru prototipuri multi-agent conversaționale
Microsoft Agent Framework	Workflows enterprise pentru agenți	Pentru integrare Azure/Microsoft, stare, telemetrie, aplicații mari
OpenAI Agents SDK	Agenți simpli cu tools, handoffs, tracing	Pentru proiecte centrate pe ecosistemul OpenAI
Pydantic AI	Agenți Python cu output structurat și validare	Pentru rezultate JSON robuste și aplicații Python curate
n8n AI Agents	Agenți vizuali conectați la servicii	Pentru automatizări low-code: RSS, Gmail, Sheets, API-uri

Din notebook în aplicație:

ce construim azi?

notebooks/student_XX/C6_01_rag_agent_response.ipynb

1. încarcă FAISS + metadate din C5
2. citește role_XX.yaml → construiește system prompt
3. regăsire top-k fragmente pentru inputul nou
4. construiește promptul manual; refă cu PromptTemplate
5. generează răspunsul; verifică groundedness manual



BACKEND + UI

- `core/agent.py` — funcție reutilizabilă: `(input, agent_id) → (răspuns, context_folosit)`
- `app/app.py` — tab minim Agent RAG: select agent → input text → generează → afișează context + răspuns

Conectectăm backend-ul RAG la interfață, pas cu pas.

La finalul C6

C6 produce primul răspuns real ancorat în corpus, cu identitate discursivă definită și context recuperat transparent. Fiecare răspuns este auditabil: știm rolul, știm fragmentele, știm promptul.

REZULTAT C6

```
{
  "agent": "Anti-sistem",
  "retrieved_context": [
    {"text": "...", "score": 0.82},
    {"text": "...", "score": 0.71}
  ],
  "response": "Decizia CCR confirmă ce știam: instituțiile..."
}
```

C6 PRODUCE

roluri YAML · core/agent.py · tab în aplicație · răspunsuri RAG ancorate în corpus

C5 = selectează contextul · C6 = generează cu context · C7 = orchestrează agenții