

D-RAG & RoLit-KG: Advanced Knowledge Graph Systems KGQA + Romanian Literary KG Construction

Roberto Hordoa
D-RAG (KGQA) Mihai Deaconu Bogdan
KGC / RoLit-KG pipeline

Work split: D-RAG (Roberto Hordoa) ▪ KGC (Mihai Deaconu Bogdan)

January 12, 2026

Goal (10 minutes)

- Present two complementary systems:
 - **D-RAG**: differentiable retrieval-augmented generation for KGQA
 - **RoLit-KG**: production pipeline to build a Romanian literary KG in Neo4j
- Show how they connect: **KG construction →KGQA**

Agenda

- ① D-RAG: problem, architecture, results, how to reproduce
- ② RoLit-KG: datasets, pipeline, schema/QC, scaling, results
- ③ Integration: using RoLit-KG as a knowledge source for D-RAG-style QA

Work split (credits)

- **D-RAG (KGQA)**: Roberto Horduan
 - Retriever + sampler + generator integration, training pipeline, results + analysis
- **KGC / RoLit-KG (KG construction)**: Mihai Deaconu Bogdan
 - Data ingestion, extraction, entity resolution, graph QC, Neo4j export + analytics

D-RAG: the KGQA problem

- Task: answer a natural-language question using evidence from a knowledge graph
- Standard RAG limitation:
 - retrieval is **discrete** (select a subgraph), breaking end-to-end gradients
- D-RAG goal: make retrieval **trainable with the generator loss**

D-RAG: core idea (differentiable retrieval)

- **Retriever (GNN)** assigns a selection probability per fact/triple
- **Sampler** uses **Gumbel-Softmax (straight-through)** to pick facts while preserving gradients
- **Differentiable prompting** injects selected graph embeddings into the LLM latent space
- Generator trains to answer; retriever trains to reduce noise while keeping recall

D-RAG: implementation highlights in this repo

- Retriever: ReaRev-style instruction-conditioned GNN (`src/model/retriever.py`)
- Sampler: Independent Binary Gumbel-Softmax (`src/model/sampler.py`)
- Generator: Nemotron-3-Nano-30B with LoRA + differentiable prompting
(`src/model/generator.py`)
- Training schedule: Phase 1 pre-train (10 epochs) + Phase 2 joint (20 epochs)

D-RAG: Implementation Challenges (Beyond the Paper)

- **Hybrid Architecture Compatibility:**

- We used **Nemotron-3-Nano-30B** (Mamba/Transformer hybrid) instead of Llama-3 (standard Transformer).
- *The Issue:* Standard HuggingFace generate() functions failed (produced empty/garbage outputs) when injecting continuous graph embeddings into the hybrid architecture.
- *The Fix:* We engineered a **custom greedy decoding loop** to manually handle autoregressive generation with `inputs_embeds`.

- **Training Stability & Optimization:**

- **Loss Balancing:** The paper's GradNorm approach proved unstable in our mixed-precision setup. We achieved convergence using a robust **static weight** ($\lambda = 0.1$).
- **LoRA Adapters:** Essential for the 30B parameter model (Rank=64, $\alpha=128$); the paper implied full fine-tuning which was infeasible for this scale.
- **Hard Capping:** Added a strictly enforced **top-100 fact cap** before the Gumbel-Softmax step to prevent OOM errors on dense heuristic subgraphs.

Phase 1 supervision: “heuristics” subgraphs (CWQ)

- Phase 1 pre-trains the retriever using **per-question subgraphs**
- We build training targets from **heuristic paths** on the gold subgraph:
 - Input: RoG-CWQ provides a per-question graph (triples), plus q_entity/a_entity
 - We cap triples per example (default -limit_triples 50) for speed/memory
 - Construct an undirected adjacency and run BFS up to 4 hops to find $q \rightarrow a$ paths
 - If no path is found: fallback to a few 1-hop edges touching seed entities (q/a)
- Edge labeling idea:
 - a triple is labeled positive iff it matches consecutive entity pairs along any path

Undocumented Implementation Details: “Solving the Black Box”

The paper describes *what* to do (heuristic supervision) but not *how* to engineer it robustly.
We had to devise specific algorithms:

- **1. Heuristic Label Construction (The “Pathfinding” Algorithm):**
 - *Paper*: “heuristic subgraphs are extracted via SPARQL query parsing” [cite: 240].
 - *Our Implementation*:
 - Parsed the gold subgraph into an undirected adjacency structure.
 - Ran **BFS (up to 4 hops)** between `q_entity` and `a_entity`.
 - **Crucial fallback**: If BFS finds no paths (disconnected subgraphs), we label 1-hop neighbors of seed entities as “positive” to prevent zero-positive / zero-signal steps.
- **2. Projector Architecture:**
 - *Paper*: Mentions a “projector” mapping GNN to LLM space [cite: 193].
 - *Our Implementation*: a 2-layer MLP ($\text{Linear} \rightarrow \text{ReLU} \rightarrow \text{Linear}$) mapping $D_{\text{GNN}} \rightarrow D_{\text{LLM}}$ for Nemotron-30B ($D_{\text{LLM}} = 2688$).
- **3. Dynamic Context Capping:**
 - *Paper*: silent on memory management for dense graphs.
 - *Our Implementation*: enforced hard caps for safety (50 triples per example in Phase 1 heuristics generation; top-100 fact cap in Phase 2 inference/training).

Heuristics generation: what we actually write to JSONL

- We generate `data/train_heuristics_cwq_train.jsonl` from a RoG-CWQ split:

```
python scripts/generate_cwq_heuristics.py \
--input data/cwq/ComplexWebQuestions_train.json \
--output data/train_heuristics_cwq_train.jsonl \
--limit_triples 50
```

- Each line includes (minimum):
 - `question: str`
 - `triples: [[head, rel, tail], ...] (capped)`
 - `paths: [[e0, e1, ...], ...] (entity sequences)`
 - `answer: str| [str] and graph_size`

Dataset shape (CWQ) and training time

- **CWQ (ComplexWebQuestions):**
 - multi-hop KGQA questions with per-question gold subgraphs (RoG-CWQ format)
 - fields we consume for heuristics: question, graph, q_entity, a_entity, answer
- **Phase 1 training:**
 - retriever warmup on heuristic labels (BCE + ranking, $\rho = 0.7$), per-question subgraphs
- **Training time note (our run):**
 - Phase 2 on CWQ: ~7 hours / epoch on an NVIDIA A100

Key D-RAG selection parameters (sampler thresholds)

- **Gumbel-Softmax temperature** (-temperature, default 0.5)
 - lower $\tau \Rightarrow$ more discrete selections; higher $\tau \Rightarrow$ smoother gradients
- **Max facts cap** (-max_facts_cap, default 100)
 - keep at most top- k facts before thresholding (compute/memory control)
- **Probability threshold** (-prob_threshold, default 0.01; paper uses 0.01)
 - filter selected/top facts below this probability; fallback to top-1 if empty
- **Phase 1 loss mixing** (ρ , default 0.7): BCE vs ranking loss in retriever warmup

D-RAG: results on CWQ (paper vs ours)

- ComplexWebQuestions (CWQ), reported metrics: Hits@1 and Gen F1

Method	Hits@1	Gen F1
Static Cascade (paper)	54.3	60.6
Dynamic Cascade (paper)	55.9	61.9
SubgraphRAG (paper)	57.0	47.2
GNN-RAG (paper)	66.8	59.4
D-RAG (paper reported)	63.8	70.3
D-RAG (ours, 20 epochs)	79.0	71.2

- Paper comparison table is summarized from `docs/drag_documentation.tex`
- Observed training overhead vs cascade baselines: ~6.8%–8.0%

D-RAG: reproduce the 20-epoch Phase 2 run

- Ensure a Phase 1 checkpoint exists (checkpoints_cwq_subgraph/phase1_best.pt)

```
python -m src.trainer.train_phase2 \
--heuristics_path data/train_heuristics_cwq_train.jsonl \
--val_heuristics_path data/train_heuristics_cwq_val.jsonl \
--phase1_checkpoint checkpoints_cwq_subgraph/phase1_best.pt \
--checkpoint_dir checkpoints_cwq_phase2_20ep \
--epochs 20 \
--batch_size 64 \
--lr 5e-5 \
--temperature 0.5 \
--ret_loss_weight 0.1 \
--max_facts_cap 100 \
--val_generation \
--eos_loss_weight 1.0
```

- Practical runtime: ~7h/epoch on CWQ (A100) ⇒ plan multi-day training for 20 epochs.

Why a KG for Romanian literature?

- Narrative texts contain characters, places, events, and recurring motifs
- A KG enables:
 - entity-centric exploration (Who interacts with whom? where?)
 - cross-document aggregation (recurring entities, hubs, communities)
 - provenance-aware evidence (every edge tied to text evidence)
- Designed to scale from small samples to full corpora

- **RO-Stories** (Hugging Face: `readerbench/ro-stories`)
 - Romanian narrative paragraphs (12,516 docs in the full corpus)
 - Field used: paragraph →text
- **HistNERo** (Hugging Face: `avramandrei/histnero`)
 - Historical Romanian NER dataset (token classification)
 - Converted into doc-level text + spans with char offsets for ingestion

Pipeline at a glance

- ① Ingest → normalize (Unicode NFC + diacritics)
- ② Chunk text (overlap for context)
- ③ Extract:
 - NER: regex / transformer (XLM-R NER) / gold HistNERo spans
 - Relations: heuristic or **LLM (Ollama JSON schema)** with evidence grounding
- ④ Resolve entities (lexical for scale; embedding-based resolution optional/roadmap)
- ⑤ Graph-level QC (constraints, dedupe, junk suppression)
- ⑥ Optional Event nodes derived from relations
- ⑦ Export Neo4j Cypher + analytics report

Schema & ontology (lightweight, constraint-friendly)

- Core node types:
 - :Work (document-level)
 - :Mention (surface span in text)
 - :Entity with secondary labels :Character/:Person/:Location/:Event
- Key edges:
 - (Work)-[:HAS_MENTION]->(Mention)
 - (Mention)-[:REFERS_TO]->(Entity)
 - (Mention)-[:COREFERS_WITH]->(Mention) (derived)
 - Entity relations: INTERACTS_WITH, LOCATED_IN, TRAVELS_TO, ...

Quality control at the graph level

- Enforce constraints (examples):
 - LOCATED_IN / TRAVELS_TO: target must be Location
 - forbid self-loops; drop type-impossible edges
- Deduplicate edges:
 - canonical direction for symmetric predicates (e.g., INTERACTS_WITH)
 - dedupe on (src, pred, tgt, doc, chunk)
- Detect/suppress junk hubs:
 - stopwords / very short tokens becoming high-degree entities

Scaling: retrieval + caching

- Problem: full-corpus runs can be dominated by model calls
- Solution:
 - Content-addressable cache by hash:
 - NER candidates cached per chunk
 - LLM relation JSON cached per chunk + entity table
 - Retrieve only relevant prior context:
 - pass previous chunk as context **only if shared entities overlap**
- Outcome: incremental rebuilds become much faster

Results (from current documentation)

- Production run example (103 docs):

Metric	Value
Documents	103
Mentions extracted	1,158
Entities resolved	30 (97% reduction)
Relations	102,316
Runtime	57s

- Full-corpus lexical run (12,519 docs): 13,106 chunks, 181k mentions, 31,721 entities, 257k relations

How to run (end-to-end)

- Download corpora:

```
python scripts/download_rolit_datasets.py --output_dir data --limit 0
```

- Run pipeline (NER + LLM relations + QC + events + caching):

```
python run_full_pipeline.py \  
    --ro_stories_jsonl data/ro_stories_full.jsonl \  
    --histnero_jsonl data/histnero_full.jsonl \  
    --output_dir outputs/rolit_kg_full_run \  
    --run_name rolit_kg_full_run \  
    --ner_engine transformers \  
    --ner_model Davlan/xlm-roberta-base-ner-hrl \  
    --relations_engine ollama \  
    --ollama_url http://inference.ccrolabs.com \  
    --ollama_rel_model llama3.2:3b \  
    --ollama_rel_timeout_s 180 \  
    --cache_dir outputs/cache \
```

Neo4j load & example queries

- Load:

```
:source outputs/rolit_kg_full_run/cypher/constraints.cypher  
:source outputs/rolit_kg_full_run/cypher/load.cypher
```

- Starter queries live in: docs/rolit_kg_starter_queries.cypher

Next steps

- Improve extraction recall while maintaining precision:
 - more relation-specific validators and confidence calibration
- Faster scaling:
 - batching, concurrency, and ANN-based entity resolution
- Better grounding:
 - align fictional mentions with historical entities (HistNERo) + external KBs
- Product:
 - Neo4j Bloom / small UI / QA over graph

How D-RAG and RoLit-KG connect

- RoLit-KG gives you a **fresh KG** from Romanian text with provenance + Neo4j export
- D-RAG consumes a KG/subgraph dataset for KGQA training and inference
- Integration path (pragmatic):
 - RoLit-KG → export triples (Entities + Relations) into a KGQA format
 - Build question/answer supervision (or synthetic QA) and train D-RAG on RoLit-KG-derived facts
- Net: **KG construction + differentiable KGQA** in one repository

Questions?