Dialogue Benchmark Generation from Knowledge Graphs with Cost-Effective Retrieval-Augmented LLMs

Reham Omar, Omij Mangukiya, Essam Mansour

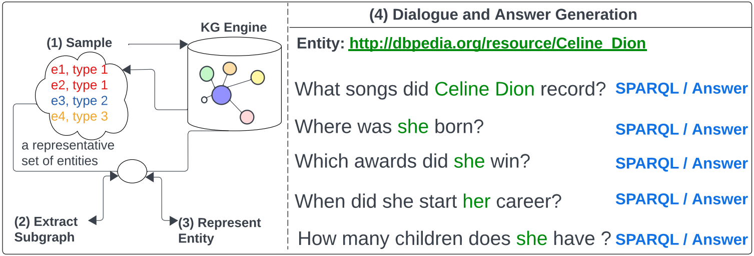
SIGMOD 2025

November 9, 2025

Outline

1.1 What is a Dialogue Benchmark?

- A dataset of multi-turn dialogues used to evaluate dialogue systems.
- Often represented as sequences (Q, A) or full dialogues with context and gold answers.
- **Applications**: educational chatbots, domain-specific assistants, benchmarking research models.



1.2 Limitations of Prior Methods

Traditional (document-based) approaches

- Manual authoring (CoQA, QuAC): high cost, low scalability.
- Template-based systems: brittle and require per-KG templates.

KG-based approaches

- Rule/template systems (CSQA, Head-to-Tail, Maestro): heavy preprocessing, limited dialogue support.
- Do not handle hallucinations or provide end-to-end dialogue generation with validations.

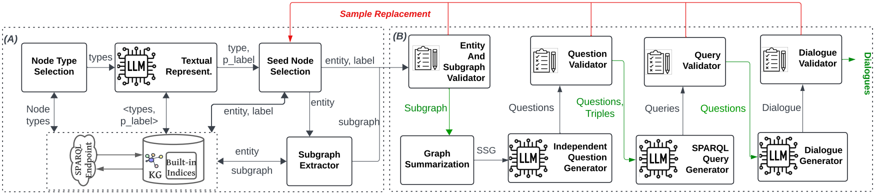
2.1 Why LLM + RAG? Challenges

- **Efficient KG retrieval**: scale — millions/billions of entities.
- **Prompt design**: complex prompts can overwhelm LLMs; few-shot vs zero-shot tradeoffs.
- **Hallucination and synthesis**: LLMs may invent facts not in the KG.
- **Cross-model consistency**: want approach working with GPT-4o, Gemini, Llama-3, etc.

2.2 Contributions / Chatty-Gen Overview

- A fully automated, multi-stage RAG pipeline (Chatty-Gen) for KG-grounded dialogue benchmark generation.
- Key features: **type-aware sampling, textual entity labels, summarized subgraphs, multi-stage generation, assertion validation**.
- Addresses challenges:
  - **retrieval efficiency** (type-aware sampling)
  - **prompt complexity** (summarized subgraphs & multi-stage generation)
  - **hallucination** (assertion-based validators)
  - **cross-LLM compatibility**

3 Chatty-Gen Architecture



(Use Figure 2 from paper: architecture diagram)

- Phase A: **Dialogue Context Extraction** (node type selection, textual representation, seed sampling, subgraph extraction)
- Phase B: **Dialogue Generation** (subgraph summarization, question generation, SPARQL generation, dialogue generation)

4.1 Algorithm 1: Representative Node Type Selection

Goal: Efficiently select representative entity types and determine sampling size per type.

Key Ideas:

- Reduce cost by operating on **types**, not individual entities.
- Remove:
  - Metadata types (e.g., ontology or administrative nodes)
  - Low-frequency types (threshold R)
  - Shadowed parent types (threshold S)
- Output a pruned and meaningful type list with sample sizes.

Algorithm 1 Node Type Selection

Input: *endpoint*: SPARQL endpoint, *m*: number of Dialogues, *domain*: prefixes of KG, *R*: Rare types threshold, *S*: Shadowed parents threshold

Output: *dist*: A map of node type to the number of entities

$dist, type\_ratio \leftarrow \{\}$

$types, count \leftarrow getKGNodeTypes(endpoint, domain)$

$total \leftarrow Sum(count)$

for every $\langle t, c\rangle \in \langle types, count\rangle$ **do**

  $type\_ratio[t] \leftarrow c/total$

end for

$type\_ratio \leftarrow removeRareTypes(type\_ratio, R)$

$type\_ratio \leftarrow removeShadowedTypes(type\_ratio, S)$

for every $\langle t, ratio\rangle \in type\_ratio$ **do**

  $dist[t] \leftarrow ratio * m$

end for

Return dist

Algorithm 1: Node Type Selection

4.1 Algorithm 1: Steps

1. **Query types & counts.**
   - Run a SPARQL aggregation on `rdf:type` to get (type, count).
2. **Filter metadata / domain.**
   - Remove ontology/meta types and restrict to desired namespace prefixes.
3. **Remove rare types (threshold R).**
   - Compute $\text{ratio}(t) = \frac{\text{count}(t)}{\sum_{t'} \text{count}(t')}$.
   - Drop types with $\text{ratio}(t) < R$ (e.g. $R = 1\%$).
4. **Remove shadowed parent types (threshold S).**
   - For parent p and child c, compute $\text{cover}(p, c) = \dfrac{\text{bothCount}(p, c)}{\text{parentCount}(p)}$.
   - If cover $> S$ (e.g. $S = 99\%$), drop the parent.
5. **Allocate sample budget.**
   - Given total target dialogues m: $\text{dist}[t] = \text{round}(\text{ratio}(t) \times m)$.
   - Resolve leftover by assigning to top-ranked types.

4.1 Algorithm 1: Complexity & Example

Complexity: $O(q_{\text{cost}} + \tau)$ — q_{cost} for SPARQL; τ = number of types.

Tiny example:

- **Type removal (rare):** Person=10,000; Place=2,000; Reservoir=2. Total=12,002. With $R = 1\%$ remove Reservoir. For $m = 120$: Person $\to$ 100, Place $\to$ 20.
- **Shadowed parent example:**
  - parent = `Creator`, parentCount = 1,000
  - child = `Person`, childCount = 1200
  - bothCount (entities labeled both Creator & Person) = 995
  - cover(Creator,Person) = 995 / 1000 = 0.995 = 99.5% $> S = 99\% \Rightarrow$ **drop parent 'Creator'**.

4.2 Textual Entity Representation: Problem & Idea

Problem.

- KG entities are URIs (e.g. `.../Q123456`), not readable names.
- URI local-names may be IDs or include disambiguation tokens like "(film)" — unreliable as natural labels.

Traditional fix (limits).

- Rule-based mapping (use `dbp:title` or `rdfs:label`) fails across heterogeneous KGs (some use `skos:prefLabel`, `foaf:name`, etc.).

Key idea.

- Use an LLM to *select the best textual predicate* for each node type from candidate literal predicates — then use its value as the entity name in prompts.

4.2 Textual Entity Representation: Method

Method (per type/entity).

1. Retrieve candidate literal predicates for a sample entity of the type (predicates with literal objects).
2. Prompt the LLM: ask which predicate best represents the entity's human-readable name.
3. Save mapping `type` $\mapsto$ chosen predicate; when generating, extract that predicate's literal as the display name (replace URI).

Benefits. Portable across KGs, improves readability, reduces hallucination by giving LLM natural entity names.

4.2 Textual Entity Representation: *Type*-Level Selection

SPARQL (list literal predicates for a *type*, e.g., `dbo:Film`):

SPARQL (type-level candidates)

```
SELECT DISTINCT ?p WHERE {
  ?s a dbo:Film; ?p ?o .
  FILTER(isLiteral(?o))
}
```

Prompt template (type-level):

> *"Given the following literal predicates observed on **dbo:Film**, which one most likely contains the movie's **title**? Candidates: {rdfs:label, dbp:title, dbo:abstract, ...}"*

Concrete decision for *type* `dbo:Film`:

- LLM selects (for `dbo:Film`): `dbp:title` *(fallback: `rdfs:label` if missing)*.
- Then for any film entity, use the value of `dbp:title` in prompts instead of the URI, e.g., "Who directed **Inception**?".

4.3 Algorithm 2: Seed Entity Sampling and Subgraph Extraction

Algorithm 2 Entity and Subgraph Extraction For Node Type t

Input: *endpoint*: SPARQL endpoint, p_l: entity label of t, n: number of entities of t, BZ: entity batch size, h: number of hops for subgraph, *shape*: Subgraph Shape, d: Direction of predicates

Output: *Entity_Subgraph*: A list of n valid entity, subgraph pair

$entity\_subgraph \leftarrow \{\}$
while $entity\_subgraph.length < n$ **do**
  $e, e_L \leftarrow getEntity(endpoint, p_l, BZ)$
  $g_{size} \leftarrow getGraphSize(endpoint, e, h)$
  $preds \leftarrow countUniquePredicates(endpoint, e, h)$
  $context_{valid} \leftarrow validateContext(g_{size}, preds)$
  $subgraph \leftarrow extractSubgraph(e, endpoint, h, shape, d)$
  $subgraph_{filtered} \leftarrow filterSubgraph(subgraph, p_l)$
  $subgraph_{valid} \leftarrow validateSubgraph(subgraph_{filtered})$
  $entity\_subgraph[e] \leftarrow subgraph_{filtered}$
end while
Return entity_Subgraph

Algorithm 2

Goal: Efficiently sample high-quality seed entities and extract subgraphs.

Key ideas

- **Batch sampling** from SPARQL endpoint (LIMIT/OFFSET) to approximate random selection without heavy latency.
- **Early filtering** on label length, predicate diversity, and subgraph size to avoid low-quality seeds.
- **Predicate blacklist / literal trimming** to remove thumbnails/long abstracts and keep salient facts.
- **Stop early** when per-type quota n_t is met to limit queries and cost.

4.3 Algorithm 2: Steps

1. **Batch entities of type t:** query SPARQL endpoint with `LIMIT BZ OFFSET` to fetch candidate entity URIs.
2. **For each candidate entity e:**
   - Retrieve predicted label e_L (from §4.2).
   - Fetch 1-hop triples (incoming + outgoing).
   - Compute subgraph stats: total triples K_e, unique predicate count P_e.
3. **Quality checks:**
   - Label exists and is short enough (avoid extremely long titles).
   - Predicate diversity $P_e \geq k$ (e.g. $k = 3$) to ensure multiple questions.
   - Subgraph token size within model context limit.
4. **Accept or reject:** accept $(e, SG(e))$ if checks pass; otherwise discard and continue.
5. **Repeat batches** until collected n_t seeds for type t or no more candidates.
6. **Return:** list of accepted seed entity — subgraph pairs for downstream generation.

4.3 Algorithm 2: Complexity & Example "Inception"

Complexity: $O\big(\tau \cdot (q_{2\_cost} + K)\big)$

- τ = number of selected representative types,
- q_{2_cost} = cost per batch SPARQL retrieval from endpoint,
- K = average triples per accepted subgraph.

Example: how "Inception" becomes a seed

- Candidate: `<http://dbpedia.org/resource/Inception>` with label **"Inception"**.
- Label check: "Inception" length small ⇒ **pass**.
- 1-hop extraction yields triples (filtered):
  - (Inception, dbp:title, "Inception")
  - (Inception, dbo:director, Christopher_Nolan)
  - (Inception, dbo:starring, Leonardo_DiCaprio)
  - (Inception, dbo:releaseDate, "2010-07-16")
- $P_e = 4 \geq k(= 3)$ and token estimate $<$ model limit ⇒ **accept as seed**.

5.1 Algorithm 3: Subgraph Summarization

Goal: Produce a compact, answer-free subgraph summary for use in **Independent Question Generation**.

Steps:

1. **Collect predicates** from the seed entity's subgraph.
2. **Select representative triple** for each predicate (first or sampled).
3. **Remove concrete objects (answers)** — retain only entity and predicate, replacing object with placeholder.

Algorithm 3 Summarizing Subgraph Algorithm

Input: SG: Subgraph
Output: SSG: Summarized Subgraph

```
1: p_u ← getUniquePredicates(SG)
2: preds_to_triples ← groupTriples(SG, p_u)
3: SSG ← []
4: for every ⟨p, t_list⟩ ∈ ⟨preds_to_triples⟩ do
5:     t′ ← modifyTriple(e, t_list[0])
6:     SSG ← ssg ∪ t′
7: end for
8: Return SSG
```

Algorithm 3

Why summarize?

- Reduce token usage to fit more context in prompts.
- Prevent revealing answers — encourages question generation from structure.
- Avoid embedding literal answers inside SPARQL queries later.

5.2 Independent Question Generation

- Input: serialized subgraph SG_{ser}, entity label e_L, desired number n_q.
- Task: generate $Q' = [Q_1, \ldots, Q_{n_q}]$ of independent, answerable questions.

$$Q' \;=\; f\big(\mathrm{SG}_{ser},\; e_L,\; n_q,\; I_{IQ}\big) \tag{3}$$

$$(Q',\; T_q) \;=\; f\big(\mathrm{SG}_{ser},\; e_L,\; n_q,\; I'_{IQ}\big) \tag{4}$$

Notes: Eq.(3) = independent-question generator. Eq.(4) = extended output that also returns, for each question, the minimal supporting triple(s) T_q (used later for SPARQL generation).

5.3 SPARQL Query Generation

- Input per question: $(Q_i, T_{q,i})$ — $T_{q,i}$ are the minimal triple(s) supporting Q_i.
- Task: produce SPARQL queries SQ_i that retrieve the answer from the KG.

$$SQ = f\left(Q',\ T_q,\ I_{SQ}\right) \tag{5}$$

Notes: providing T_q reduces prefix/predicate errors and grounds the LLM into using predicates present in the subgraph.

5.4 Dialogue Generation

- Input: independent-question list Q' and entity label e_L.
- Task: turn Q' into a coherent multi-turn dialogue D (first question standalone; later turns use co-reference / pronouns).

$$D \;=\; f\big(Q',\, e_L,\, I_{DG}\big) \tag{6}$$

Notes: Eq.(6) is the final dialogue transform in the multi-stage pipeline (uses e_L to help pronoun/coref handling).

5.5 Validators: Constraints Summary

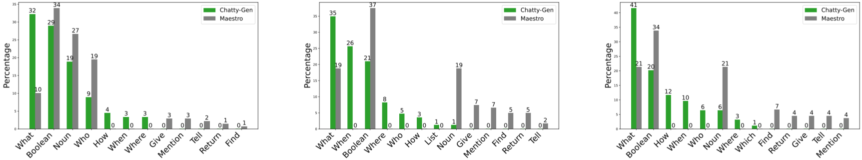
- **Question Validator**: each Q_i must be answerable from SG and explicitly reference entity (for Q_1).
- **Query Validator**: SPARQL syntax check + execution match to expected triple(s).
- **Dialogue Validator**: coherence checks (pronoun usage, Q_1 independence, no one-word Qs).
- Retry policy: up to 3 attempts per stage, otherwise drop seed and resample.

6.1 Evaluation Setup

- Benchmarks generated on three real-world KGs:
  - **DBLP**, **DBpedia**, **YAGO**, **MAG**
- Compared systems:
  - **Chatty-Gen**
  - **Maestro** (state-of-the-art KG question benchmark generator)
- Two hardware setups used:
  - High-memory server for SPARQL + closed-source LLMs
  - GPU server for open-source LLMs

6.2 Question Quality: Type Diversity

- Maestro often generates repetitive and boolean-type questions.
- Chatty-Gen generates more balanced, human-like question types.
- This improves dialogue naturalness and domain coverage.



(a) DBLP KG (b) YAGO KG (c) DBpedia KG

Figure: Comparison of question type diversity. (Figure 4 in paper)

6.3 Node Type Coverage in KGs

- Chatty-Gen's entity sampling preserves the original node-type distribution of KG.
- Maestro's selection misses important entity types (e.g., Person in DBpedia).

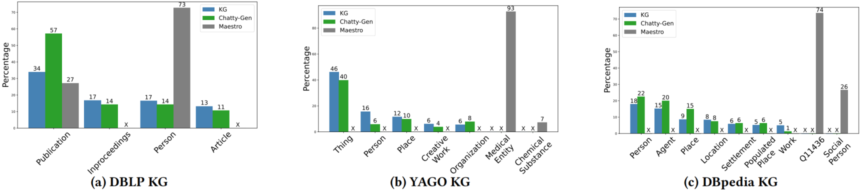


Figure: Seed node type distribution comparison. (Figure 5 in paper)

6.4 End-to-End Time Efficiency (Table 3)

- Maestro requires full KG preprocessing, scaling poorly on large KGs.
- Chatty-Gen retrieves subgraphs on demand, enabling significant speedups.

KG	Maestro (hrs)	Chatty-Gen (hrs)
DBpedia	30.77	0.17
YAGO	5.20	0.10
MAG	5.38	0.12
DBLP	0.12	0.12

Table: End-to-end generation time (Table 3 in paper).

6.5 Consistent Performance Across LLMs

- Multi-stage pipeline significantly reduces hallucinations and SPARQL errors.
- Open-source setups (e.g., LLaMA + CodeLLaMA) reach performance **comparable to GPT-4o**.

Approach	LLM	YAGO								DBLP							
		Success rate %	Dialogue-S	Question-E	SPARQL-E	Dialogue-E	Parsing-E	Time(mins)	# Tokens M ($)	Success rate %	Dialogue-S	Question-E	SPARQL-E	Dialogue-E	Parsing-E	Time(mins)	# Tokens M ($)
Our Multi-stages (3 prompts)	GPT-4o	100	20	0	0	0	0	8	0.04(0.32)	95	20	0	0	1	0	6	0.05(0.38)
	GPT-4	95	20	0	0	1	0	13	0.05(0.71)	100	20	0	0	0	0	11	0.05(0.78)
	GPT-3.5	91	20	1	0	1	0	6	0.05(0.04)	95	20	1	0	0	0	7	0.06(0.05)
	Gemini-1-pro	71	20	0	2	0	6	6	0.06(0.01)	67	20	5	0	0	5	5.2	0.09(0.02)
	Gemini-1.5-pro	22	20	0	1	0	71	20.5	0.18(0.35)	41	20	0	0	1	28	14.5	0.12(0.24)
	LLAMA-3-8b	13	20	2	124	9	0	84	0.36	20	20	2	47	23	8	71	0.33
	LLAMA-3-8b-inst	41	20	0	16	0	13	37	0.13	14	20	0	106	0	16	89	0.39
	LLAMA-2-13b	5	20	5	325	10	1	270	0.79	1	5	32	426	7	22	392	1.30
	CodeLLAMA-7b	7	20	3	162	74	16	234	0.92	1	5	144	156	44	145	356	0.16
	CodeLLAMA-13b	83	20	0	0	4	0	31	0.08	63	20	3	5	2	1	37	0.01
	Mistral-7b-v0.1	17	20	2	88	3	0	70	0.25	3	15	5	323	5	14	189	0.90
	Multi-LLM-1	100	20	0	0	0	0	18	0.06	83	20	0	4	0	0	20	0.07
	Multi-LLM-2	91	20	0	0	0	2	18	0.06(0.28)	71	20	2	4	0	2	23	0.10(0.82)
single prompt	GPT-4o	38	20	11	0	5	15	9	0.03(0.32)	8	20	52	0	8	144	59	0.15(1.30)
	GPT-4	40	20	5	1	1	23	17	0.04(0.73)	21	20	5	1	2	66	46	0.09(1.86)
	GPT-3.5	5	19	2	0	8	242	53	0.36(0.32)	8	20	25	0	5	119	50	0.25(0.24)
	Gemini-1-pro	56	20	0	10	4	1	3	0.03(0.01)	10	20	130	0	12	6	21	0.30(0.07)
	Gemini-1.5-pro	14	20	59	4	0	5	16	0.12(0.27)	6	20	28	0	3	10	51.5	0.43(0.9)
	LLAMA-3-8b	0	0	44	107	158	92	251	1	0	2	170	23	55	452	286	1.23
	LLAMA-3-8b-inst	9	20	10	13	59	43	72	0.26	7	20	51	6	44	71	90	0.38
	LLAMA-2-13b	0	0	40	216	87	401	403	1	0	0	53	39	18	827	479	1.22
	CodeLLAMA-7b	0	0	72	87	196	344	314	1.07	0	1	203	45	30	604	313	1.25
	CodeLLAMA-13b	1	11	78	29	579	168	443	1.07	1	5	188	11	91	677	459	1.26
	Mistral-7b-v0.1	0	0	53	93	420	223	261	0.98	0	1	208	34	118	528	257	1.20

Figure: LLM performance comparison (Table 4 in the paper).

6.6 Summarized vs Full Graphs

- Compare using **Summarized subgraphs** vs **Full subgraphs**.
- Summarization reduces prompt size → fewer tokens → lower cost.
- At the same time, it **improves SPARQL correctness**.

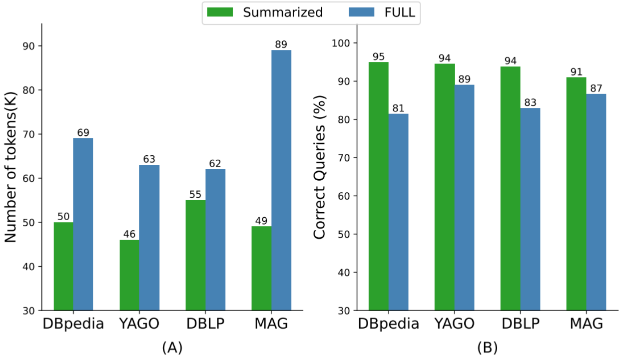


Figure: Token usage and SPARQL correctness comparison (Figure 6 in the paper).

7.1 Conclusion

- **Chatty-Gen** presents a cost-effective pipeline for generating **multi-turn dialogue benchmarks** directly from KGs.
- The system **avoids full KG preprocessing** and instead performs **on-demand subgraph retrieval**, significantly reducing generation time.
- By using a **multi-stage LLM prompting strategy** and optional **subgraph summarization**, Chatty-Gen enables **consistent performance across both proprietary and open-source LLMs**.
- Evaluations show that Chatty-Gen generates dialogues that are **more diverse, entity-grounded, and semantically aligned** with the original KG compared to prior systems (e.g., Maestro).

7.2 Limitations

1. Limited Question Complexity

- Chatty-Gen relies on 1-hop subgraphs and predicate summarization.
- Generated questions are mainly factual (What/Who/When) — lack multi-hop reasoning or comparative logic.
- Future work: adaptive multi-hop retrieval and reasoning-path prompting.

2. Error Propagation in Multi-Stage Pipeline

- Each stage (entity representation $\rightarrow$ question $\rightarrow$ SPARQL $\rightarrow$ dialogue) depends on previous LLM outputs.
- A single semantic or predicate error propagates through later stages, degrading overall quality.
- Validators mitigate syntax-level errors but cannot fully eliminate semantic drift.