ChatGPT

Proposed Architecture for an LLM-to-KG Pipeline in Biomedicine

Overview and Goals

This design outlines a pipeline to automatically construct a biomedical knowledge graph (KG) using Large Language Models (LLMs). The goal is to extract **(subject, predicate, object)** triplets from literature (e.g. PubMed articles) and organize them into a knowledge graph. We focus on a flexible, exploratory approach: - **Triplet-centric extraction:** Use LLMs to identify factual triples from text as the primary building blocks of the KG. This forms a **"graph scaffold"** of concepts and relations [1] [2], which can be enriched into an ontology later. - **Multiple output formats:** Support simple graphs (open-schema triples) initially, with the possibility to evolve into formal ontologies (with typed entities and hierarchies) or other representations. - **Downstream flexibility:** Ensure the KG can support search, question answering, and reasoning use-cases by designing an appropriate storage and retrieval mechanism. - **Modularity and scalability:** Structure the system into independent modules (ingestion, LLM extraction, embedding, storage, etc.) that can be improved or expanded (e.g. adding more data sources, better retrievers, prompt tuning) without redesigning the whole pipeline.

Pipeline Stages and Components

The end-to-end process is broken into modular stages. Each stage can be implemented and optimized independently, and existing components (the provided **Embedder**, **LLMClient**, and **POP** prompt orchestrator) are integrated where appropriate:

1. **Data Acquisition & Preprocessing:** Gather biomedical text from sources like PubMed (e.g. abstracts or full-text from open-access journals). This may involve using APIs (NCBI E-utilities) or curated datasets. Preprocessing includes cleaning the text (removing references, OCR artifacts, etc.) and splitting it into manageable segments:
2. *Segmentation:* Break documents into paragraphs or sentences. This is crucial for LLM context window limits. For example, each sentence or paragraph will be processed for triplets separately [2].

3. *Term Highlighting (optional):* Use NLP tools (e.g. spaCy) to extract candidate **entities (nouns)** and **relation keywords (verbs)** from each segment [2]. These candidates can guide the LLM to focus on relevant terms in that segment.

4. **LLM-Based Triplet Extraction:** For each text segment, use an LLM to generate triples representing factual relations in that text. We define a prompt template (using the **POP PromptFunction** class) to ensure consistent output. Key design choices here:

5. *Prompt format:* Instruct the LLM to output triples in a structured format (e.g. a JSON array of `{"subject": "", "predicate": "", "object": ""}` triples). Prior work shows LLMs can output ontology artifacts in JSON or OWL/Turtle given a suitable prompt [3]. The **LLMClient** allows

attaching a JSON schema or function call for structured output, which we leverage for reliability. By enforcing a schema and using a low temperature (e.g. 0.2), we get more stable, reproducible triplet outputs [4].

6. *In-context guidance:* We inject any known context into the prompt. For example, we can insert the candidate entities/verbs from the NLP preprocessing into the prompt (e.g. *"The sentence mentions X, Y, Z; extract relations involving these terms"*) [2]. This constrained prompting helps the LLM focus and avoid spurious triples.
7. *Few-shot examples:* We include a couple of demo examples in the prompt illustrating the format (especially if using GPT-4 or similar). For instance, Papaluca *et al.* used a prompt with two example sentences and their triples before the test sentence [5] [6]. These examples (especially domain-specific ones) can improve output consistency.
8. *Retrieval augmentation (optional):* If a knowledge base of known biomedical facts exists, retrieve the top relevant facts or example triples for the segment and include them as **"context"** in the prompt. This **RAG approach** can significantly boost extraction accuracy [7] [8] (discussed more below). Retrieved context might be presented as *additional triplets* ("Context: (A, R, B), (C, R2, D)...") or as analogous examples ("Example text: ... Triplets: ..."). The pipeline can use the **Embedder** module here: embed each segment and find similar segments or triples from a reference corpus (using cosine similarity in the embedding vector space) [9] [10].

9. *LLM Execution:* Using the **LLMClient**, we call a chosen model (e.g. GPT-4 via OpenAI, or a local model like Mistral-7B) with the prepared prompt. The POP PromptFunction wraps this call, injecting any dynamic data (the text segment, retrieved context, etc.) into the prompt template. The LLM's response is then either parsed as JSON or directly captured via function-call output. We implement **retry logic** to handle formatting errors – e.g. if the model's first attempt is not valid JSON, we can prompt again or use the parsing functions as in prior work [4].

10. **Post-processing & Triple Normalization:** The raw triples from the LLM may contain redundancies or inconsistencies that need cleanup before inserting into the KG:

11. *Normalization:* Clean and normalize entity names (subjects/objects). For example, unify synonyms, strip plurality or capitalization differences ("Vitamin C" vs "vitamin C"), and possibly map abbreviations to full forms. Simple string normalization (lowercasing, removing trailing periods) is done first; a more advanced step could map terms to canonical identifiers (using biomedical dictionaries if grounding to an ontology).
12. *Filtering:* Optionally, discard low-confidence triples. Since we are currently not doing formal evaluation, this could be heuristic (e.g. drop triples with very generic relations like "related to" if they dominate, or use the LLM's own logprob/confidence if available). In an exploratory phase, it might be preferable to keep everything and later prune or mark uncertain triples.
13. *Aggregation:* Merge duplicates. If multiple sources yield the same triple (or semantically same triple), consolidate them and perhaps track provenance (which documents supported this triple). This can be as simple as a dictionary key on the triple string or as complex as an ontology merge (outside our current scope).

14. *Orphan node handling:* If the process isolated some terms that were not placed in any triple (e.g. a sentence with no verb relation yielded no triple but contained an entity of interest), we can include those as isolated nodes so that potentially they can be linked later [4]. This ensures important concepts aren't lost even if relations were not extracted in that context.

15. **Knowledge Graph Storage:** After extracting and cleaning triples, we need to store the growing knowledge graph in a queryable, scalable format:

16. *Graph database:* We can load the triples into a property graph database like **Neo4j**. Each unique subject/object becomes a node, and each predicate becomes a relationship (edge) connecting nodes. This allows efficient graph queries (e.g. find all relations of gene X) and graph algorithms. We can also attach node/edge properties (e.g. source document, confidence score).
17. *RDF triple store:* Alternatively, use an RDF triple store (e.g. Apache Jena, GraphDB) to store triples in **subject-predicate-object** form with URIs. RDF/OWL is ideal if we plan to integrate or **align with existing biomedical ontologies** (for example, linking "aspirin" to a UMLS or MeSH URI). It also enables SPARQL queries and reasoning over ontology schema. The LLM can be instructed to output in an RDF serialization like Turtle or JSON-LD for direct import [3], though converting from JSON triplets is straightforward as well.
18. *JSON/Document storage:* For quick prototyping, we might simply store triples in a JSON or CSV file or a document database (e.g. MongoDB). For example, each triple as a JSON object, or grouped by source. This is less efficient for complex querying but easy to set up and can be indexed by the Embedder for semantic search. JSON-LD is a good compromise – it's a JSON format that includes context for linking to ontologies, so it can be later ingested into an RDF system if needed.

19. *Vector index:* In parallel, we maintain a vector index of the triples (or node embeddings) using the **Embedder**. This index (using e.g. FAISS or an API) will allow semantic similarity searches. For instance, given a user query or a new sentence, we can retrieve relevant triples by embedding the query and finding nearest neighbors in the triple embedding space [10]. This complements exact graph queries with a **semantic lookup** capability.

20. **Ontology Enrichment (Future Extension):** Once a substantial triple collection is built, we can consider converting it into a richer ontology:

21. *Derive entity types:* Analyze the nodes and relations to infer a type hierarchy. For example, if many subjects are disease names, we introduce a class "Disease" and assert those as instances. This can be partly automated by LLM or external knowledge (e.g. a dictionary of biomedical entity types). Some triples might explicitly be *"(A, is a, B)"* which hints at taxonomy (A is a subtype of B). Those can be used to form **TBox** axioms (class-subclass relations) [11].
22. *Link to existing ontologies:* For grounding, map our nodes to identifiers in known ontologies (SNOMED CT, Gene Ontology, DrugBank, etc.). This could involve string matching or an LLM-based entity linking step. Grounding improves interoperability but may be challenging and is not required in early exploration.
23. *Add ontological relations:* If needed, distinguish between different relation semantics – e.g. some predicates might be hierarchical ("is-a"), others causal ("inhibits", "causes"), etc. We can define these in an ontology schema so that users can, for instance, query for any "is-a" path or do logical reasoning. Initially, we treat all relations as homogeneous edges in the graph (open schema), but as patterns emerge, we might standardize certain predicates.
24. *Human-in-the-loop validation:* Ontology building often benefits from expert curation. In a later phase, domain experts could review the most frequent relations and entities, correct mistakes, and help establish which relations to treat as key ontology properties. The system can then retrain or re-prompt the LLM to prefer those standardized relations.

Prompting Strategies: Zero-Shot, Few-Shot, and RAG

A critical design aspect is how we prompt the LLM to extract triples. We consider three strategies:

Method	Approach	Pros	Cons
Zero-Shot	Prompt the LLM with only an instruction (e.g. *“Extract all (subject, predicate, object) triples from the text below”*) and the text, with no examples. The model relies on its pre-trained knowledge to perform Triplet Extraction (TE).	- Simplest setup (no additional data needed). - Avoids any bias from examples – fully open-ended.	- May miss domain-specific relations not seen often in training (especially biomedical jargon). - Output format can be inconsistent without examples; more prone to errors in structure or irrelevant info.
Few-Shot	Provide a few **hard-coded examples** of sentences and their triples in the prompt before the actual text. For instance, include 2–3 exemplar biomedical sentences with correct extracted triples (these can be handcrafted or taken from a dataset). Then ask for triples of the new text.	- Improves format consistency and guides the model on the style/ level of detail expected [5] [6]. - No need for external storage: examples are baked into the prompt.	- Consumes prompt tokens (limits input size for long texts). - Examples might not cover all types of relations; maintaining a small, representative set is challenging. The prompt may also become brittle if examples are not well-chosen.
Retrieval-Augmented (RAG)	**Dynamically fetch context** relevant to the input text from an external knowledge source (a preliminary KG or background corpora). Two modes: (a) retrieve *triplets* similar to those that might appear, and supply them as “context triples”; or (b) retrieve analogous **(sentence, triples)** pairs as on-the-fly examples, replacing any static examples [9] [12]. The LLM sees information closely related to the input.	- Provides factual grounding and domain knowledge the LLM might lack, which **boosts extraction performance** significantly [8] (often matching supervised models). - Adaptable: as KG grows, the model always sees up-to-date relevant facts. Helps with long-tail or complex relations if the retriever finds something similar.	- More complex pipeline: requires a **vector index and retriever** to be in place and kept updated. - The benefit depends on retrieval quality – irrelevant or erroneous context can confuse the LLM. (Studies found the quality of retrieved context strongly correlates with final output quality [13].) - Longer overall prompt (context uses up tokens).

Recommended approach: Start with a *hybrid of zero-shot and few-shot* prompting for simplicity, then move to RAG as the knowledge base grows. For example, in early experiments we might use a few manually crafted examples covering typical biomedical relations (drug-disease, protein-interaction, etc.). Once the pipeline has accumulated some triples, we can switch to retrieval-augmented prompts using those triples as context. This balances initial simplicity with later accuracy gains. Notably, using an LLM in this way avoids the need to predefine a closed ontology of relations – the model can identify novel relations on the fly [14], which is valuable in biomedical discovery. We just need to mitigate the consistency issues of prompting (via format enforcement and context as described).

Schema and Grounding Considerations

One design decision is whether to enforce a specific ontology schema up front or use an open schema (free-form relations) approach: - **Open-schema KG:** In the exploratory phase, treat the triples as an open knowledge graph with no strict type system. Subjects and objects are text strings (e.g. "aspirin", "platelet aggregation") and predicates are mostly verb phrases or relation terms as extracted (e.g. "inhibits", "associated with"). The advantage is **flexibility** – the LLM can output any relation that makes sense, without being limited to a predefined list [14]. This is useful in biomedicine where new mechanisms or relations may appear in literature. The downside is that we may get semantically equivalent relations phrased differently ("inhibits" vs "blocks" vs "suppresses"), or entities in variant forms, requiring later normalization. In open-schema mode, we rely on post-processing and perhaps clustering to align similar relations. - **Ontology-grounded KG:** Here we map triples to a known ontology or schema. For example, we might decide that predicates must come from a controlled vocabulary (like the UMLS relation types or schema.org relations), and entities should be linked to identifiers (e.g. Gene IDs, DOIs for diseases). Grounding to an ontology enables powerful reasoning and consistency checks (e.g. domain/range of relations) and integration with existing knowledge bases. We could prompt the LLM to use specific relation names or to output IDs if we provide a dictionary. However, this requires either a pre-built ontology or iterative schema learning. One approach is **ABox→TBox co-extraction**: first extract many instance-level triples (ABox) and then abstract them into a schema (TBox) [11]. Our pipeline leans toward this approach – build the graph freely, then retrospectively impose structure.

In practice, we likely start open-schema (maximizing recall of facts) and gradually **ground the KG** as patterns emerge. For example, if we notice many predicates meaning "treats" (treats, alleviates, is therapy for...), we might standardize on a relation *treats*. This can be done via a mapping table or by instructing the LLM in future runs to prefer certain terms. Likewise, we can link recurring entities to a database (for example, normalize drug names to a DrugBank ID). The **grounding process** can also leverage the LLM (e.g., ask the LLM to identify if an extracted entity is a drug, gene, symptom, etc., by comparing to known lists or descriptions).

Human oversight is valuable here: domain experts can review extracted triples and assign types or map them to known ontology classes. The system can incorporate these mappings in the prompt (e.g. "Treat 'aspirin' as Drug:DB00945") or via a lookup step during post-processing.

Integration of Existing Components

We will utilize and extend the provided modules to implement this pipeline: - **Embedder Module:** This will be used in two ways: (a) *Retrieval:* create embeddings for either text segments or triples and build a vector index. The `Embedder` supports multiple backends (OpenAI embeddings, Jina, or local transformer models). For the biomedical domain, a locally fine-tuned model (like BioBERT or PubMedBERT embeddings) could be plugged in by specifying its model name. This vector index enables the RAG approach: given a new sentence, compute its embedding and find similar content/triples in the index to provide as prompt context [10]. (b) *Node embedding:* We can also embed entire entities or concepts for later clustering or semantic search. For instance, find similar diseases by embedding their descriptions or connected triples. - **LLMClient:** This abstraction lets us switch between LLM providers easily (OpenAI, local, etc.). Initially, we might use `OpenAIClient` with GPT-4 or GPT-3.5 for high-quality extraction. The client supports advanced features like function calls and JSON schema (`response_format`) which we'll use to get structured output. In a later phase, we could integrate a local model (via `LocalPyTorchClient`) if we fine-tune one for this task. The modular client means the pipeline can route requests to different models depending on context (e.g., a smaller model for simple cases, GPT-4 for complex cases or verification). - **POP PromptFunction:** This provides a convenient way to define reusable prompt templates. We will create a `PromptFunction` for **triplet extraction** – it will contain the base prompt (with placeholders for the text, and slots for examples or context). The system prompt can include high-level instructions (e.g. "You are a biomedical information extraction assistant…" plus any special formatting requirements). At runtime, the pipeline will call `prompt_function.execute(text=..., ADD_BEFORE=..., ADD_AFTER=...)` to fill in the segment and insert any retrieved context or extra instructions. The POP module handles combining the system and user prompts and calling the LLMClient. By using this orchestrator, we ensure consistency across calls (all segments get the same treatment) and can easily adjust the prompt in one place to improve outputs (prompt tuning). - **Error handling & iteration:** The pipeline will be designed to catch errors or low-confidence outputs and handle them. The PromptFunction/LLMClient can detect if the response was a function call with no arguments or if JSON parsing failed, etc., and we can implement a retry or fallback. For example, if GPT-4 is not available, switch to a backup model; or if the output is empty for a sentence, we might log that sentence for later manual review (to ensure we're not missing important info). These mechanisms align with the idea of an **"engineering workflow"** for LLM-assisted extraction [3] [1] – rather than trusting a single-pass LLM answer, we systematically catch issues and maintain a pipeline that can be refined.

By integrating these components, the pipeline benefits from existing capabilities while remaining extensible. For instance, if we need to support a new prompting style (say, chain-of-thought reasoning to extract a complex relation), we can create a new PromptFunction for that and slot it into the process. If we want to improve retrieval, we can swap in a new Embedder model or vector database with minimal changes to the overall system.

Knowledge Graph Storage and Representation

Choosing the right representation for the KG affects how we query and grow the knowledge. The design should allow easy updates (as new triples come in) and support the intended downstream tasks (search, QA, reasoning):

- **Property Graph DB (e.g. Neo4j):** Represents the KG as nodes and edges with properties. This is intuitive: each entity is a node, and each triple becomes an edge of type *predicate* from subject to object. We can store metadata (source, confidence) as properties. Neo4j's Cypher query language can find patterns (e.g. all paths between Gene X and Disease Y) which is useful for discovery. It's also horizontally scalable (with enterprise setups) and can integrate with Graph Data Science libraries for advanced analytics. For our use, a Neo4j instance could be updated in batch after processing a batch of documents, or even in real-time via an API call from the pipeline.

- **RDF Triple Store (Semantic Web):** Stores triples in the RDF format, optionally with an OWL ontology. Each entity and relation is identified by a URI. This is powerful for **interoperability** – many biomedical databases (like MeSH, GO, etc.) are available in RDF and aligning our KG to them means we can directly connect to that knowledge. We can use SPARQL queries to precisely query the graph (including inferencing if an OWL reasoner is on). For example, if our KG knows "aspirin – treats – headache" and an ontology says treats is inverse of "treated_by", a reasoner can infer "headache – treated_by – aspirin". We might not need such formal reasoning initially, but designing with RDF in mind keeps the door open. We could serialize our triples to **Turtle or JSON-LD**. JSON-LD is particularly appealing for a web-based pipeline: our pipeline can output JSON but with an `@context` that maps terms to ontology URIs, effectively making the data semantic web-friendly without changing the JSON structure.
- **Document Stores + Vectors:** If heavy graph querying is not immediately needed, a pragmatic approach is to store each triple (or each subject's neighborhood) as a document in a NoSQL store, and rely on vector search and LLMs for reasoning at query time. For instance, to answer a question, retrieve relevant triples via embedding search, then let an LLM compose the answer. This is more of a *retrieval-augmented QA* approach than a classical KG query, but it aligns with using LLMs' strengths. The pipeline can support this by maintaining the vector index (using the Embedder) and simply storing the triples in a flat manner. The trade-off is that we lose precise graph querying capabilities, but we gain simplicity and flexibility. In fact, we can combine approaches: use a graph DB for structured queries and a vector index for semantic similarity retrieval – whichever suits the task at hand.

Scalability considerations: The storage choice should handle the volume of data we plan to extract. PubMed has millions of abstracts, which could lead to tens of millions of triples. A distributed triple store or sharded graph DB might be necessary in the long run. Early on, we can manage with a local or cloud instance and monitor performance (each triple insertion/query). The design should allow swapping the backend: for example, start with a local Neo4j for development, but later move to Amazon Neptune or an Azure Cosmos DB if needed for scale. Because our pipeline's interface to storage will be through a small set of functions (e.g. `add_triple(subject, predicate, object)` and query functions), changing the backend won't require changes in extraction logic or prompting.

Scalability and Modular Design

To ensure the system can evolve and cover more ground in the biomedical domain, we propose several modular design strategies:

- **Parallel and Distributed Processing:** The pipeline can be parallelized at multiple levels. For instance, different documents or paragraphs can be processed concurrently (since each is independent for extraction). Using job queues or a distributed framework (Spark, Ray, or simply asyncio for I/O-bound calls) will speed up processing large corpora. The LLM API calls (if to OpenAI) can be a bottleneck, so parallel calls or batching will be considered within rate limits. The design should include a scheduler that manages these parallel tasks and merges results into the KG. The *paralleled subtasks* idea is highlighted in prior work [15] – some tasks like term extraction, triple generation, and quality checks can run simultaneously to optimize throughput.
- **Scaling Domain Coverage:** Biomedicine is broad (molecular biology, clinical medicine, pharmacology, etc.). We can modularize by **domain or corpus**. For example, have separate ingestion pipelines for different data sources (clinical trials, patents, etc.) feeding into the same extraction

engine. If certain sub-domains require special handling (e.g. genomic data might need parsing of gene symbols), those can be plug-in modules in preprocessing. The prompting strategy might also branch by domain: we could maintain specialized prompt variants or examples for different topics (a prompt tuned for drug–drug interactions versus one for disease–symptom relations). The architecture should allow adding such specializations without disrupting the core. This could be done by tagging each input with a domain and having the PromptFunction choose a slightly different template or few-shot examples accordingly.

- **Improving the Retriever:** Since the quality of retrieved context is vital for RAG [13], we plan for iteration on the embedding model and retrieval technique. Initially, a general MiniLM or BioBERT-based embedding is used [10]. Over time, we could fine-tune embeddings on our accumulated corpus of triples (so that similar relations are embedded closer) or even train a custom retriever (e.g. using knowledge distillation from the LLM's outputs to improve the embedding space). The retriever is a self-contained component – it could be as simple as an in-memory FAISS index or as elaborate as a dedicated search engine (like ElasticSearch with a synonyms dictionary for biomedical terms). The pipeline design abstracts retrieval so we can replace the mechanism without altering how the LLM prompt is constructed (only the source of the context changes).
- **Prompt and Model Tuning:** As we gather more data, we can refine the LLM approach. One path is **few-shot prompt tuning**: by analyzing errors from initial outputs, we add or adjust the example triples in the prompt, or incorporate additional instructions (e.g. "If multiple triples describe the same relationship, merge them into one"). Another path is fine-tuning or specializing an LLM. For example, we might fine-tune a smaller open-source model on a set of extracted triples to see if it can approximate GPT-4's performance. This model could then be used for the bulk of extraction to save costs, while GPT-4 is reserved for verification or particularly difficult sections. The architecture makes this possible by using the LLMClient interface – we can route requests to different models dynamically (even ensemble them, e.g. ask two models and compare answers for higher confidence).

- **Monitoring and Evaluation:** Even if formal evaluation is not the focus now, the design should log key metrics to guide future improvements. For instance, track how many triples are extracted per document, the proportion of empty outputs, and basic plausibility checks (like does the subject text actually appear in the source sentence – a quick recall check). Logging these can help identify failure modes (maybe certain types of sentences consistently yield nothing, or a particular relation is often malformed). With an **iterative loop**, we can use this feedback to adjust preprocessing or prompting. In a modular sense, one could plug in an "evaluation module" that, say, uses a small set of expert-annotated papers to compute precision/recall of the extracted triples [16], or even have the LLM self-critique its output (by asking a second prompt whether a triple is supported by the text). Those are optional components that can run in parallel and report back, without halting the main pipeline.

- **Extensibility:** The entire pipeline is conceived as a set of modules with clear interfaces (ingest -> segments, segments -> triples, triples -> KG). This means new capabilities can be added as modules. For example, a **question answering module** could be attached on top of the KG: it would accept natural language queries, use the Embedder to fetch relevant triples, and either answer directly or feed them to an LLM for a composed answer. Another example is a **visualization module** to create a graph visualization of the triples for a given topic – since the data is in a standard graph form, this can be added without changing extraction logic. By keeping modules decoupled, we ensure that scaling up (in data volume or in functionality) is manageable.

Conclusion

In summary, the proposed LLM-to-KG pipeline for biomedical knowledge is a **hybrid system** that marries LLM's natural language understanding with structured knowledge graph techniques. We extract rich relational data from unstructured text in the form of triplets, leveraging zero-shot and few-shot LLM capabilities and augmenting them with retrieved context as the KG grows. The architecture emphasizes flexibility: starting open-schema for maximum coverage and gradually introducing ontology elements for grounding and reasoning. Each part of the pipeline – from data ingestion to triple extraction to storage – is designed to be modular, allowing iterative improvement (better models, more data sources, refined prompts) without a complete overhaul. This approach will enable us to populate a robust biomedical knowledge graph that can power search, question answering, and reasoning applications, while continuously adapting to new information and domains. The use of existing components (embedding models, prompt frameworks, and LLM APIs) accelerates development, and lessons from recent research (e.g. on workflow design [17] and augmentation strategies [8]) guide us in balancing precision, recall, and scalability in this LLM-driven KG construction.

---

[1] [2] [3] [4] [11] [15] [16] [17] 2509.00140v1.pdf
file://file-DGpYen5XNWjLcXT8qgvyJs

[5] [6] [7] [8] [9] [10] [12] [13] [14] 2312.01954v1.pdf
file://file-4ogUa8rouzxQhrXURkx1ZG