Architektura Systemów GraphRAG: Strategiczna Implementacja Grafów Wiedzy w Ekosystemie Next.js i Postgres

1. Wstęp: Ewolucja Pamięci Agentów AI

Współczesne systemy sztucznej inteligencji, w szczególności agenty autonomiczne takie jak "Lolek", stoją przed fundamentalnym wyzwaniem kognitywnym: fragmentacją kontekstu. Tradycyjne podejście RAG (Retrieval-Augmented Generation), oparte wyłącznie na wektorowej reprezentacji danych (Vector RAG), choć skuteczne w wyszukiwaniu semantycznego podobieństwa, zawodzi w zadaniach wymagających rozumienia złożonych struktur relacyjnych. Gdy agent AI opiera się jedynie na pgvector lub podobnych rozwiązaniach, jego "pamięć" jest zbiorem odizolowanych fragmentów tekstu, pozbawionych strukturalnego spoiwa łączącego fakty w logiczną całość.

Problem ten, określany w literaturze jako brak "holistycznego zrozumienia", objawia się niezdolnością agenta do nawigowania po wielopoziomowych zależnościach (multi-hop reasoning). Agent może wiedzieć, że "Paweł lubi Reacta" i że "Projekt X używa Reacta", ale bez jawnej struktury grafowej często nie potrafi wywnioskować, że "Paweł jest kluczowym interesariuszem w Projekcie X". Przejście od płaskiej pamięci wektorowej do architektury GraphRAG (Graph-based Retrieval Augmented Generation) stanowi ewolucyjny skok, przekształcający pasywną bazę wiedzy w dynamiczną strukturę zdolną do wnioskowania.

Niniejszy raport stanowi wyczerpujące studium architektoniczne wdrożenia GraphRAG w nowoczesnym stosie technologicznym opartym na Next.js i PostgreSQL (Neon). Analiza obejmuje krytyczne decyzje infrastrukturalne, automatyzację procesu budowania grafu wiedzy przy użyciu LLM oraz projektowanie hybrydowych schematów retencji danych, które łączą precyzję wyszukiwania wektorowego z głębią analizy grafowej.

---

2. Warstwa Danych: Infrastruktura i Wybór Silnika

Grafowego

Fundamentem każdego systemu GraphRAG jest decyzja dotycząca silnika składowania danych relacyjnych. W kontekście ekosystemu opartego na Next.js i hostingu serverless (Neon), wybór ten jest determinowany przez kompromis między spójnością infrastruktury a wydajnością operacji grafowych. Analiza koncentruje się na dwóch dominujących paradygmatach: podejściu zintegrowanym (PostgreSQL z rozszerzeniami) oraz podejściu poliglota (PostgreSQL + dedykowana baza grafowa).

2.1. Podejście Zintegrowane: Apache AGE w Ekosystemie Postgres

Apache AGE (A Graph Extension) to projekt inkubowany przez Apache Software Foundation, mający na celu przekształcenie relacyjnej bazy PostgreSQL w system wielomodelowy, obsługujący zarówno standardowe zapytania SQL, jak i zapytania grafowe w języku openCypher.[1]

2.1.1. Architektura i Integracja

Apache AGE funkcjonuje jako rozszerzenie do PostgreSQL. W przeciwieństwie do natywnych baz grafowych, które stosują tzw. index-free adjacency (bezindeksowe sąsiedztwo), AGE emuluje strukturę grafu wewnątrz tabel relacyjnych, wykorzystując typ danych jsonb do przechowywania właściwości węzłów i krawędzi.[2] Z perspektywy architektury systemu, kluczową zaletą jest unifikacja. Wszystkie dane – wektory, tabele użytkowników oraz graf wiedzy – znajdują się w jednym klastrze, co upraszcza zarządzanie transakcjami, backupami oraz pulą połączeń w aplikacjach Next.js.

2.1.2. Wyzwania w Środowisku Serverless (Neon)

Mimo teoretycznych zalet, wdrożenie Apache AGE w środowisku serverless takim jak Neon napotyka na istotne bariery infrastrukturalne.

- **Wsparcie Rozszerzeń:** Neon, jako platforma serverless Postgres, oferuje szerokie

wsparcie dla popularnych rozszerzeń takich jak pgvector czy pg_trgm.[3] Jednakże, Apache AGE jest rozszerzeniem głęboko ingerującym w wewnętrzne struktury PostgreSQL. Wymaga ono specyficznych bibliotek ładowanych na poziomie startu serwera, co w środowiskach współdzielonych (multi-tenant), charakterystycznych dla chmury serverless, jest często blokowane ze względów bezpieczeństwa lub stabilności.[4]

- **Wydajność i Skalowanie:** Zapytania grafowe, zwłaszcza te o dużej głębokości rekurencji, są intensywne obliczeniowo i pamięciowo. W modelu serverless, gdzie zasoby są dynamicznie skalowane (często do zera), "zimny start" połączony ze skomplikowanym trawersowaniem grafu może prowadzić do nieakceptowalnych opóźnień (latency spikes) lub błędów Out Of Memory na mniejszych instancjach.[5]
- **Stabilność Ekosystemu:** Społeczność wokół AGE jest wciąż mniejsza niż wokół natywnych rozwiązań. Użytkownicy zgłaszają problemy z kompatybilnością wersji Postgresa (np. opóźnienia w obsłudze PG16/PG17) oraz mniejszą wydajność w porównaniu do dedykowanych silników przy dużych zbiorach danych.[6]

2.2. Dedykowane Bazy Grafowe: Neo4j i Memgraph

Alternatywą jest architektura "Sidecar", gdzie PostgreSQL (Neon) pełni rolę głównego magazynu danych (Source of Truth) oraz silnika wektorowego, natomiast dedykowana baza grafowa obsługuje wyłącznie strukturę powiązań.

2.2.1. Neo4j: Standard Przemysłowy

Neo4j jest najstarszą i najbardziej dojrzałą bazą grafową, wykorzystującą natywne składowanie grafowe (native graph storage). Oznacza to, że relacje są fizycznie zapisane jako wskaźniki na dysku, co pozwala na trawersowanie grafu w czasie stałym $O(1)$ dla każdego przeskoku, niezależnie od całkowitego rozmiaru bazy danych.[8]

- **Ekosystem:** Neo4j oferuje najbogatszy ekosystem narzędzi, w tym bibliotekę Graph Data Science (GDS) z gotowymi implementacjami algorytmów takich jak PageRank czy Louvain, kluczowych dla zaawansowanego GraphRAG.[9]
- **Dostępność:** Usługa Neo4j AuraDB oferuje bezpłatny poziom (Free Tier) pozwalający na przechowywanie do 200,000 węzłów i 400,000 relacji. Dla projektu agenta osobistego lub firmowego jest to limit w zupełności wystarczający do zbudowania bardzo rozbudowanego grafu wiedzy bez ponoszenia kosztów infrastrukturalnych.[11]

2.2.2. Memgraph: Wydajność Czasu Rzeczywistego

Memgraph to baza grafowa typu in-memory, napisana w C++, która stawia na maksymalną wydajność. Jest w pełni kompatybilna z protokołem Bolt i językiem Cypher używanym przez Neo4j, co czyni migrację między tymi systemami stosunkowo prostą.[8]

- **Przewaga Wydajnościowa:** Testy wydajnościowe (benchmarki) wskazują, że Memgraph może być nawet 120 razy szybszy od Neo4j w operacjach zapisu i mieszanych obciążeniach, dzięki wyeliminowaniu narzutu maszyny wirtualnej Javy (JVM) i operowaniu bezpośrednio w pamięci RAM.[8]
- **Zastosowanie w RAG:** W scenariuszu, gdzie agent musi w czasie rzeczywistym analizować tysiące połączeń w trakcie generowania odpowiedzi (np. podczas rozmowy z użytkownikiem), milisekundowe opóźnienia Memgrapha mogą być kluczowe.
- **Ograniczenia:** Model in-memory wymaga wystarczającej ilości pamięci RAM, co może podnosić koszty przy bardzo dużych grafach. Wersja chmurowa (Memgraph Cloud) oferuje 14-dniowy okres próbny, ale darmowy plan jest bardziej ograniczony czasowo niż "wieczysty" Free Tier w Neo4j AuraDB.[15]

2.3. Rekurencyjne CTE w Postgres: Rozwiązanie Pośrednie

Dla projektów, które nie mogą pozwolić sobie na drugą bazę danych, a nie mają dostępu do Apache AGE, istnieje trzecia droga: modelowanie grafu w czystym SQL przy użyciu Rekurencyjnych Wspólnych Wyrażeń Tablicowych (Recursive CTE).[17]

- **Implementacja:** Graf jest reprezentowany przez dwie tabele: nodes i edges. Zapytania wykorzystują klauzulę WITH RECURSIVE do przechodzenia przez hierarchie.
- **Wady:** Wydajność CTE drastycznie spada przy głębokich relacjach. Badania pokazują, że przy złożonych zapytaniach rekurencyjnych, dedykowane bazy grafowe (lub nawet biblioteki w pamięci jak NetworkX) są rzędami wielkości szybsze.[5] CTE jest trudne w utrzymaniu i bardzo podatne na błędy przy pisaniu zapytań o ścieżki (pathfinding).

2.4. Rekomendacja Architektoniczna: Model Hybrydowy

Biorąc pod uwagę cel projektu – stworzenie agenta o "holistycznym zrozumieniu" w

środowisku Next.js/Neon – rekomenduje się architekturę hybrydową **Postgres + Neo4j AuraDB**.

Kryterium	Apache AGE (Neon)	Neo4j AuraDB	Memgraph Cloud
Kompatybilność z Neon	Niska / Ryzykowna	Wysoka (Zewnętrzna usługa)	Wysoka (Zewnętrzna usługa)
Koszt (Mała Skala)	W cenie bazy Postgres	Darmowy (Free Tier)	Ograniczony Trial
Wydajność Grafowa	Średnia (SQL Wrapper)	Wysoka (Native Graph)	Bardzo Wysoka (In-Memory)
Ekosystem Algorytmów	Podstawowy	Zaawansowany (GDS)	Zaawansowany (MAGE)
Złożoność Operacyjna	Niska (Jeden system)	Średnia (Dwa systemy)	Średnia (Dwa systemy)

Werdykt: Należy odrzucić Apache AGE w kontekście Neon ze względu na ryzyko braku wsparcia i problemy z wydajnością serverless. Neo4j AuraDB (Free Tier) jest optymalnym wyborem na start – oferuje stabilność, darmową instancję "forever free" i doskonałą integrację z ekosystemem LangChain/LlamaIndex. Memgraph warto rozważyć w fazie produkcyjnej, jeśli opóźnienia staną się wąskim gardłem.

---

3. "Architekt" - Automatyzacja Budowy Grafu Wiedzy

Centralnym wyzwaniem w GraphRAG jest transformacja nieustrukturyzowanych notatek w ustrukturyzowane węzły i krawędzie. Proces ten, jeśli ma być wykonywany automatycznie przez agenta ("Lolek"), wymaga rygorystycznego potoku przetwarzania (pipeline), aby uniknąć chaosu w danych (np. duplikacji encji "Paweł", "P. Nowak", "Kierownik").

3.1. Projektowanie Ontologii (Schemat Grafu)

Pierwszym krokiem, często pomijanym, jest zdefiniowanie ścisłej ontologii. Pozwalanie LLM na "dowolną inwencję" w tworzeniu typów węzłów prowadzi do grafu o niskiej użyteczności. Należy zdefiniować "Słownik Dozwolonych Typów".[19]

Przykładowa Ontologia JSON:

JSON

```
{
 "nodes": [
  { "type": "Osoba", "attributes": ["imie", "rola", "dzial"] },
  { "type": "Projekt", "attributes": ["nazwa", "status", "termin"] },
  { "type": "Technologia", "attributes": ["nazwa", "wersja"] },
  { "type": "Firma", "attributes": ["nazwa", "branza"] }
 ],
 "relationships":
}
```

3.2. Narzędzia Ekstrakcji: LangChain vs LlamaIndex

Do realizacji zadania ekstrakcji najlepiej wykorzystać wyspecjalizowane biblioteki, które narzucają strukturę wyjściową LLM.

3.2.1. LangChain LLMGraphTransformer

Moduł ten (część langchain-experimental) pozwala na zdefiniowanie allowed_nodes i allowed_relationships. Działa on poprzez inżynierię promptów, która instruuje model (np.

GPT-4o), aby analizował tekst i zwracał obiekty grafowe wyłącznie zgodne ze schematem.[19]

- **Zaleta:** Łatwa integracja z łańcuchami (Chains) i obsługa wielu backendów grafowych.
- **Przykład użycia:**

```TypeScript
const transformer = new LLMGraphTransformer({
 llm: model,
 allowedNodes: ["Osoba", "Projekt"],
 allowedRelationships:,
 strictMode: true
});
```

3.2.2. LlamaIndex SchemaLLMPathExtractor

LlamaIndex oferuje podejście PropertyGraphIndex, które jest bardziej zorientowane na gotowe indeksy. SchemaLLMPathExtractor działa podobnie do rozwiązania LangChain, ale jest ściślej zintegrowany z mechanizmami przechowywania (StorageContext) LlamaIndex.[22]

- **Zaleta:** Lepsza obsługa "implicit extraction" (wnioskowania ukrytych relacji) i wbudowane mechanizmy persystencji.

3.3. Problem Rozwiązywania Tożsamości (Entity Resolution)

Najtrudniejszym aspektem automatyzacji jest deduplikacja. Jeśli notatka A wspomina "Pawła", a notatka B "Pawła Nowaka", system nie powinien tworzyć dwóch osobnych węzłów.

Strategia Wektorowa (Vector-Based Entity Linking):

1. Przed dodaniem nowego węzła "Paweł", system generuje jego embedding (wektor).
2. Wykonywane jest wyszukiwanie wektorowe w bazie istniejących węzłów typu Osoba.
3. Jeśli znaleziony zostanie węzeł o podobieństwie > 0.95 (np. "Paweł Nowak"), system zakłada, że to ta sama osoba i łączy dane (MERGE) zamiast tworzyć nowy byt.[24]
4. W przypadku niepewności (podobieństwo 0.8-0.9), agent może zapytać użytkownika: "Czy 'Paweł' z nowej notatki to ta sama osoba co 'Paweł Nowak' z działu IT?".

3.4. Architektura Przetwarzania w Tle (Background Jobs)

Proces ekstrakcji grafu jest czasochłonny (wymaga wielu wywołań LLM). W architekturze Next.js (Serverless) nie można wykonywać go w ramach standardowego żądania HTTP (które ma limity czasu, np. 10-60 sekund). Niezbędne jest użycie kolejki zadań.

Rekomendowane rozwiązanie: Inngest lub Trigger.dev.26
Narzędzia te pozwalają na definiowanie funkcji działających w tle, które są odporne na awarie i limity czasu serverless.
Przepływ Danych (Pipeline):

1. **Zdarzenie:** Użytkownik zapisuje notatkę w aplikacji Next.js.
2. **Trigger:** API Next.js wysyła zdarzenie note.created do Inngest.
3. **Funkcja Przetwarzająca (The Architect):**
    - **Krok 1 (Chunking):** Podział notatki na semantyczne fragmenty z zachowaniem kontekstu (overlap).
    - **Krok 2 (Ekstrakcja):** Uruchomienie LLMGraphTransformer na każdym fragmencie.
    - **Krok 3 (Rezolucja):** Sprawdzenie duplikatów w Neo4j.
    - **Krok 4 (Zapis):** Wykonanie transakcji Cypher (MERGE) zapisującej węzły i krawędzie.

---

4. Holistyczne Zrozumienie: Schemat Hybrydowy i Wyszukiwanie

Kluczowym celem projektu jest umożliwienie agentowi "holistycznego zrozumienia". Oznacza to zdolność do odpowiedzi na pytania przekrojowe, np. "Jaka jest atmosfera w zespole projektowym Alfa?", co wymaga agregacji wielu faktów, a nie tylko znalezienia jednego fragmentu tekstu.

4.1. Metodologia Microsoft GraphRAG: Podsumowania Społeczności

Microsoft Research zaproponował przełomowe podejście do GraphRAG, które opiera się na

Detekcji Społeczności (Community Detection).[28]

4.1.1. Algorytmy Klastrowania (Leiden / Louvain)

Zamiast traktować graf jako zbiór pojedynczych węzłów, system dzieli go na "społeczności" – gęsto połączone grupy węzłów. Algorytm Leiden (ulepszony Louvain) jest tu standardem, ponieważ gwarantuje spójność klastrów i działa szybciej.[30]

- W Neo4j algorytm ten jest dostępny w bibliotece GDS (gds.leiden).
- W Postgres (bez AGE) implementacja Leiden jest trudna i nieefektywna, co potwierdza słuszność wyboru dedykowanej bazy grafowej.

4.1.2. Hierarchiczne Podsumowania

Dla każdej wykrytej społeczności (np. "Zespół Backendowy", "Projektanci UI"), LLM generuje tekstowe podsumowanie: "Ta grupa obejmuje Pawła i Annę, którzy pracują nad migracją bazy danych przy użyciu PostgreSQL". Te podsumowania są następnie wektoryzowane i zapisywane w pgvector.

4.2. Schemat Hybrydowy: Most między pgvector a Grafem

Aby połączyć "miękką" wiedzę z wektorów ze "sztywną" strukturą grafu, należy zastosować fizyczne powiązanie w bazie danych. Rekomenduję następujący schemat logiczny:

Tabela w Postgres (Neon):

Kolumna	Typ	Opis
chunk_id	UUID	Unikalny identyfikator fragmentu tekstu.
content	TEXT	Treść notatki.

embedding	VECTOR(1536)	Reprezentacja wektorowa (OpenAI/Cohere).
metadata	JSONB	Metadane (autor, data).

Węzły w Neo4j:

- (:Chunk {id: "uuid-z-postgres"}) – Węzeł reprezentujący fragment tekstu.
- (:Osoba), (:Projekt) – Węzły encji.

Kluczowa Relacja: (:Osoba)-->(:Chunk)
Kiedy LLM ekstrahuje informację o "Pawle" z fragmentu tekstu, tworzy relację MENTIONED_IN. Dzięki temu agent może przejść od osoby do wszystkich tekstów, które o niej mówią, i odwrotnie.32

4.3. Strategia Wyszukiwania Hybrydowego (The "Sweeper")

Proces wyszukiwania odpowiedzi przez agenta powinien przebiegać dwutorowo:

1. **Ścieżka Wektorowa (Local Search):**
    - Zapytanie użytkownika jest wektoryzowane.
    - pgvector zwraca 5 najbardziej podobnych fragmentów (Chunks).
    - System pobiera z grafu węzły połączone z tymi fragmentami (kontekst 1. rzędu).
2. **Ścieżka Grafowa (Global/Traversal Search):**
    - Z zapytania ekstrahowane są encje kluczowe (np. "React").
    - System wykonuje przeszukanie grafu (np. "Znajdź wszystkie projekty używające Reacta i osoby w nich pracujące").
    - System przeszukuje również wektorowy indeks **Podsumowań Społeczności**, aby złapać kontekst wysokiego poziomu (np. "Problemy z wydajnością w projektach Reactowych").
3. **Fuzja i Reranking:**
    - Wyniki z obu ścieżek są łączone.
    - Model Reranker (np. Cohere Rerank) ocenia przydatność każdego fragmentu kontekstu.
    - LLM otrzymuje "holistyczny" prompt zawierający zarówno szczegóły techniczne, jak i mapę powiązań.

---

5. Blueprint Implementacji Technicznej

Poniżej przedstawiono szczegółowy plan implementacji w oparciu o zdefiniowaną architekturę.

5.1. Stos Technologiczny

- **Frontend/Backend:** Next.js 14 (App Router).
- **Baza Relacyjna/Wektorowa:** Neon (Postgres).
- **Baza Grafowa:** Neo4j AuraDB (Free Tier).
- **Orkiestracja AI:** LangChain.js.
- **Kolejkowanie:** Inngest (dla asynchronicznej budowy grafu).
- **Model:** GPT-4o (dla ekstrakcji i syntezy) lub GPT-4o-mini (dla redukcji kosztów ekstrakcji).

5.2. Konfiguracja Połączenia (Drizzle ORM + Neo4j Driver)

W Next.js należy zarządzać dwoma klientami baz danych.

TypeScript

```
// lib/neo4j.ts
import neo4j from 'neo4j-driver';

const driver = neo4j.driver(
  process.env.NEO4J_URI!,
  neo4j.auth.basic(process.env.NEO4J_USER!, process.env.NEO4J_PASSWORD!)
);

export const getGraphSession = () => driver.session();
```

5.3. Definicja Funkcji "Architekta" (Inngest)

TypeScript

```
// inngest/functions/graph-architect.ts
import { inngest } from "@/lib/inngest";
import { LLMGraphTransformer } from
"@langchain/community/graph_transformers/llm";
import { ChatOpenAI } from "@langchain/openai";

export const buildKnowledgeGraph = inngest.createFunction(
 { id: "build-knowledge-graph" },
 { event: "note.created" },
 async ({ event, step }) => {
  const { text, noteId } = event.data;

  // Krok 1: Ekstrakcja Grafu
  const graphDocuments = await step.run("extract-entities", async () => {
   const model = new ChatOpenAI({ modelName: "gpt-4o", temperature: 0
});
   const transformer = new LLMGraphTransformer({
    llm: model,
    allowedNodes:,
    allowedRelationships:,
    strictMode: true
   });
   return await transformer.convertToGraphDocuments();
  });

  // Krok 2: Zapis do Neo4j z Logiką Merge
```

```
  await step.run("persist-graph", async () => {
    const session = getGraphSession();
    try {
     // Przykład zapisu węzłów Osoba
     for (const doc of graphDocuments) {
       for (const node of doc.nodes) {
         if (node.type === 'Osoba') {
           await session.run(
             `MERGE (p:Osoba {name: $name})
                  SET p.last_seen = datetime()`,
             { name: node.id }
           );
         }
         //... logika dla innych węzłów i relacji
       }
       // Tworzenie połączenia z fragmentem tekstu (Chunk)
       await session.run(
         `MATCH (p:Osoba {name: $name})
           MERGE (c:Chunk {id: $noteId})
           MERGE (p)-->(c)`,
         { name: node.id, noteId: noteId }
       );
     }
    } finally {
     await session.close();
    }
  });
 }
);
```

5.4. Logika Wyszukiwania (Server Action)

TypeScript

```
// app/actions/ask-lolek.ts
"use server";

export async function askLolek(question: string) {
  // 1. Równoległe przeszukiwanie
  const = await Promise.all();

  // 2. Budowanie kontekstu
  const context = `
   Znalezione fragmenty notatek:
   ${vectorRes.map(v => v.content).join('\n')}

   Znalezione powiązania w grafie wiedzy:
   ${graphRes.map(g => `${g.source} -> ${g.relationship} -> ${g.target}`).join('\n')}
   `;

  // 3. Generowanie odpowiedzi
  const response = await openai.chat.completions.create({
    messages: [
      { role: "system", content: "Jesteś Lolkiem, asystentem AI. Odpowiadaj holistycznie." },
      { role: "user", content: `Pytanie: ${question}\nKontekst:\n${context}` }
    ]
  });
```

```
  return response.choices.message.content;
}
```

---

6. Wnioski i Rekomendacje

Implementacja GraphRAG to znaczący krok naprzód w budowaniu inteligentnych agentów. Analiza wykazała, że:

1. **Infrastruktura:** Próba wdrożenia **Apache AGE** w środowisku serverless (Neon) niesie ze sobą zbyt duże ryzyko operacyjne i wydajnościowe. Hybryda **Neon (Postgres) + Neo4j AuraDB** jest rozwiązaniem optymalnym, łączącym stabilność relacyjną z mocą natywnego silnika grafowego.
2. **Automatyzacja:** Kluczem do sukcesu "Lolka" jest asynchroniczny "Architekt" (pipeline oparty na Inngest), który w tle przetwarza notatki. Bez ścisłej ontologii i deduplikacji, graf szybko stanie się bezużyteczny.
3. **Holistyka:** Prawdziwe "zrozumienie" powstaje na styku wektorów i grafów. Zastosowanie algorytmów detekcji społeczności (Leiden) oraz fizyczne łączenie węzłów grafu z wektorami treści (relacja MENTIONED_IN) pozwala agentowi widzieć nie tylko słowa, ale i strukturę problemu.

Przyjęcie tej architektury pozwoli Twojemu agentowi nie tylko "pamiętać" fakty, ale "rozumieć" kontekst Twojej firmy, projektów i relacji międzyludzkich w sposób, który do tej pory był nieosiągalny dla prostych systemów RAG.

Works cited

1. Apache AGE Extension - Azure Database for PostgreSQL - Microsoft Learn, accessed November 25, 2025, https://learn.microsoft.com/en-us/azure/postgresql/flexible-server/generative-ai-age-overview
2. Introduction to Apache AGE - Graph Extension for PostgreSQL - DEV Community, accessed November 25, 2025, https://dev.to/markgomer/introduction-to-apache-age-graph-extension-for-postgresql-p2l
3. Supported Postgres extensions - Neon Docs, accessed November 25, 2025, https://neon.com/docs/extensions/pg-extensions
4. Apache AGE as Dynamic Extension - Neon - Answer Overflow, accessed November 25, 2025, https://www.answeroverflow.com/m/1373133925982928976
5. Graph Database or Relational Database Common Table Extensions: Comparing acyclic graph query performance - Stack Overflow, accessed November 25,

2025, https://stackoverflow.com/questions/63112168/graph-database-or-relational-database-common-table-extensions-comparing-acyclic
6. Learn neo4j first or jump straight into the Apache AGE extension for Postgres? - Reddit, accessed November 25, 2025, https://www.reddit.com/r/PostgreSQL/comments/1ckglkt/learn_neo4j_first_or_jump_straight_into_the/
7. Apache AGE performance : r/apacheage - Reddit, accessed November 25, 2025, https://www.reddit.com/r/apacheage/comments/1byu6io/apache_age_performance/
8. Memgraph vs Neo4j in 2025: Real-Time Speed or Battle-Tested Ecosystem? - Medium, accessed November 25, 2025, https://medium.com/decoded-by-datacast/memgraph-vs-neo4j-in-2025-real-time-speed-or-battle-tested-ecosystem-66b4c34b117d
9. Neo4j Vector Index and Search - Developer Guides, accessed November 25, 2025, https://neo4j.com/developer/genai-ecosystem/vector-search/
10. The Basics of Data Modeling - Neo4j, accessed November 25, 2025, https://neo4j.com/blog/graph-data-science/data-modeling-basics/
11. Support resources and FAQ for Aura Free Tier - Neo4j, accessed November 25, 2025, https://support.neo4j.com/s/article/16094506528787-Support-resources-and-FAQ-for-Aura-Free-Tier
12. Neo4j AuraDB – Frequently Asked Questions, accessed November 25, 2025, https://neo4j.com/cloud/platform/aura-graph-database/faq/
13. Memgraph or Neo4j: Analyzing Write Speed Performance, accessed November 25, 2025, https://memgraph.com/blog/memgraph-or-neo4j-analyzing-write-speed-performance
14. Memgraph vs. Neo4j: A Performance Comparison, accessed November 25, 2025, https://memgraph.com/blog/memgraph-vs-neo4j-performance-benchmark-comparison
15. Frequently asked questions - Memgraph, accessed November 25, 2025, https://memgraph.com/docs/help-center/faq
16. Memgraph Cloud, accessed November 25, 2025, https://memgraph.com/docs/getting-started/install-memgraph/memgraph-cloud
17. PostgreSQL Graph Database: Everything You Need To Know - PuppyGraph, accessed November 25, 2025, https://www.puppygraph.com/blog/postgresql-graph-database
18. Implementing Hierarchical Data Structures in PostgreSQL: LTREE vs Adjacency List vs Closure Table - DEV Community, accessed November 25, 2025, https://dev.to/dowerdev/implementing-hierarchical-data-structures-in-postgresql-ltree-vs-adjacency-list-vs-closure-table-2jpb
19. Source code for langchain_experimental.graph_transformers.llm - LangChain overview, accessed November 25, 2025, https://python.langchain.com/api_reference/_modules/langchain_experimental/gr

aph_transformers/llm.html
20. LangChain-Kùzu Integration: Transforming Text into Graphs - Analytics Vidhya, accessed November 25, 2025, https://www.analyticsvidhya.com/blog/2025/01/langchain-kuzu-integration/
21. Source code for langchain_experimental.graph_transformers.llm, accessed November 25, 2025, https://api.python.langchain.com/en/latest/_modules/langchain_experimental/graph_transformers/llm.html
22. Property Graph Construction with Predefined Schemas | LlamaIndex Python Documentation, accessed November 25, 2025, https://developers.llamaindex.ai/python/examples/property_graph/property_graph_advanced/
23. Customizing property graph index in LlamaIndex, accessed November 25, 2025, https://www.llamaindex.ai/blog/customizing-property-graph-index-in-llamaindex
24. Find and link similar entities in a knowledge graph using Amazon Neptune, Part 1: Full-text search | AWS Database Blog, accessed November 25, 2025, https://aws.amazon.com/blogs/database/find-and-link-similar-entities-in-a-knowledge-graph-using-amazon-neptune-part-1-full-text-search/
25. Implementing 'From Local to Global' GraphRAG With Neo4j and LangChain: Constructing the Graph, accessed November 25, 2025, https://neo4j.com/blog/developer/global-graphrag-neo4j-langchain/
26. Next.js Quick Start - Inngest Documentation, accessed November 25, 2025, https://www.inngest.com/docs/getting-started/nextjs-quick-start
27. Frameworks, guides and examples - Trigger.dev, accessed November 25, 2025, https://trigger.dev/docs/guides/introduction
28. Welcome - GraphRAG, accessed November 25, 2025, https://microsoft.github.io/graphrag/
29. GraphRAG: Improving global search via dynamic community selection - Microsoft Research, accessed November 25, 2025, https://www.microsoft.com/en-us/research/blog/graphrag-improving-global-search-via-dynamic-community-selection/
30. Clustering using Louvain algorithm | by Dilip Kumar - Medium, accessed November 25, 2025, https://dilipkumar.medium.com/clustering-using-louvain-algorithm-4d2b2bcfcec7
31. 20181022_Using the Leiden algorithm to find well-connected clusters in networks, accessed November 25, 2025, https://www.cwts.nl/blog?article=n-r2u2a4
32. Knowledge Graph Generation - Graph Database & Analytics - Neo4j, accessed November 25, 2025, https://neo4j.com/blog/developer/knowledge-graph-generation/
33. Vectors and Graphs: Better Together - Graph Database & Analytics - Neo4j, accessed November 25, 2025, https://neo4j.com/blog/developer/vectors-graphs-better-together/