Architektura Danych i Strategia Zarządzania Pamięcią dla Autonomicznego Agenta AI "Lolek": Raport Techniczny

1. Wstęp: Paradygmat Ciągłości Poznawczej w Systemach Autonomicznych

Projektowanie autonomicznego agenta AI, takiego jak "Lolek", w oparciu o stos technologiczny Next.js, Vercel AI SDK, Gemini oraz Vercel Postgres (Neon), wymaga fundamentalnej zmiany w podejściu do architektury oprogramowania. Tradycyjne aplikacje internetowe opierają się na paradygmacie bezstanowości (stateless), gdzie każde żądanie HTTP jest traktowane jako izolowane zdarzenie, a trwałość danych ogranicza się do zapisu transakcyjnego. Jednakże, w kontekście autonomicznych agentów, ten model jest niewystarczający. Aby agent wykazywał cechy inteligencji, adaptacji i "życia", musi on funkcjonować w paradygmacie ciągłości poznawczej, gdzie baza danych przestaje być jedynie magazynem logów, a staje się cyfrowym odpowiednikiem hipokampu – struktury odpowiedzialnej za konsolidację pamięci krótkotrwałej w długotrwałą oraz nawigację przestrzenną i koncepcyjną.[1]

Niniejszy raport stanowi wyczerpującą analizę techniczną i projekt architektury danych dla agenta Lolek. Skupia się na rozwiązaniu kluczowego problemu systemów LLM (Large Language Models): amnezji kontekstowej. Modele językowe, takie jak Gemini, posiadają ograniczone okno kontekstowe. Chociaż rośnie ono z każdą iteracją modeli, przesyłanie całej historii interakcji przy każdym zapytaniu jest ekonomicznie nieefektywne i technicznie obciążające. Rozwiązaniem jest zewnętrzna, trwała pamięć (Long-Term Memory), która pozwala agentowi na przywoływanie faktów, zdarzeń i umiejętności proceduralnych w odpowiednim momencie.[3]

W dalszej części dokumentu przedstawiono szczegółową strategię implementacji wielowarstwowej architektury pamięci, inspirowanej badaniami nad *Generative Agents* oraz frameworkiem *MemGPT*. Architektura ta integruje pamięć epizodyczną (zdarzenia),

semantyczną (wiedza) i proceduralną (umiejętności), wykorzystując zaawansowane mechanizmy wektorowe dostępne w PostgreSQL poprzez rozszerzenie pgvector.[5] Szczególny nacisk położono na specyfikę środowiska Vercel Edge i Neon Serverless, identyfikując wyzwania związane z połączeniami TCP w środowiskach bezserwerowych oraz proponując rozwiązania oparte na adapterach sterowników Prisma.[7]

2. Architektura Poznawcza Agenta: Teoria i Implikacje dla Danych

Zanim przejdziemy do fizycznego modelu danych, konieczne jest zdefiniowanie procesów poznawczych, które ten model musi wspierać. Autonomia agenta nie wynika z samego modelu językowego, ale z pętli sprzężenia zwrotnego między percepcją, pamięcią a planowaniem.

2.1 Strumień Pamięci (The Memory Stream)

Podstawową strukturą danych w proponowanym systemie jest Strumień Pamięci. W przeciwieństwie do typowych tabel Messages w aplikacjach czatowych, które przechowują jedynie wymianę zdań między użytkownikiem a botem, Strumień Pamięci rejestruje pełne spektrum "doświadczenia" agenta. Badania nad *Generative Agents* wskazują, że skuteczne symulowanie zachowań ludzkich wymaga zapisu nie tylko komunikacji, ale także obserwacji środowiskowych i wewnętrznych myśli.[6]

Strumień ten musi być polimorficzny, obsługując różne typy zdarzeń pamięciowych:

1. **Obserwacje (Observations):** Bezpośrednie percepcje wejściowe, np. "Użytkownik przesłał plik PDF", "Jest godzina 14:00".
2. **Myśli (Thoughts):** Wewnętrzny monolog agenta, generowany w reakcji na obserwacje, np. "Użytkownik wydaje się zaniepokojony bezpieczeństwem danych".
3. **Refleksje (Reflections):** Syntetyczne wnioski wyższego rzędu, powstające w procesie analizy własnych wspomnień, np. "Użytkownik priorytetyzuje prywatność nad wygodę".
4. **Wywołania Narzędzi (Tool Calls):** Zapis akcji podjętych przez agenta, np. "Wywołanie API do bazy pogody".
5. **Wyniki Narzędzi (Tool Results):** Odpowiedzi z systemów zewnętrznych, które stają się nowymi obserwacjami.[9]

Implikacją dla schematu bazy danych jest konieczność stworzenia uniwersalnej tabeli zdarzeń,

która jest ściśle uporządkowana chronologicznie, ale jednocześnie pozwala na szybkie wyszukiwanie semantyczne. Każdy wpis w tym strumieniu musi posiadać wektor osadzenia (embedding), co umożliwia wyszukiwanie skojarzeniowe, a nie tylko po słowach kluczowych.[1]

2.2 Drzewo Refleksji i Rekurencyjna Synteza

Kluczowym aspektem autonomii jest zdolność do generalizacji. Jeśli agent zapamięta jedynie surowe fakty ("Użytkownik pije kawę we wtorek", "Użytkownik pije kawę w środę"), przy setnej interakcji może mieć trudność z wywnioskowaniem ogólnej preferencji bez analizy setek rekordów. Mechanizm Refleksji rozwiązuje ten problem poprzez okresowe (np. co 100 interakcji lub cyklicznie przez Cron) skanowanie pamięci i generowanie wniosków.[11]

Struktura danych musi wspierać ten proces poprzez relacje wiele-do-wielu między zdarzeniami pamięci. Jedna refleksja (węzeł nadrzędny) może wynikać z wielu obserwacji (węzły podrzędne). Tworzy to strukturę drzewiastą (Drzewo Refleksji), gdzie liśćmi są surowe obserwacje, a korzeniem są abstrakcyjne cechy osobowości użytkownika lub stan świata. Baza danych musi umożliwiać śledzenie tej genealogii myśli (provenance), aby agent mógł "wyjaśnić", dlaczego podjął daną decyzję, cofając się po drzewie do faktów źródłowych.[12]

2.3 Pamięć Proceduralna i Pętle Samokorekty (Reflexion)

Autonomiczny agent "Lolek" będzie korzystał z narzędzi (Function Calling). Pamięć proceduralna w tym kontekście to biblioteka sprawdzonych sekwencji działań. Architektura *Reflexion* (Shinn et al.) wprowadza koncepcję werbalnego sprzężenia zwrotnego, gdzie agent ocenia własne działania po ich wykonaniu. Jeśli agent wygeneruje błędne zapytanie SQL lub halucynuje odpowiedź, system wymusza "autorefleksję", która jest zapisywana w pamięci epizodycznej.[14]

Dla schematu bazy danych oznacza to konieczność przechowywania metadanych o sukcesie lub porażce danej interakcji. W przyszłych cyklach, przed podjęciem akcji, agent przeszukuje pamięć pod kątem podobnych zadań, które zakończyły się niepowodzeniem, aby uniknąć powtórzenia błędu. Jest to realizacja uczenia się na podstawie "pojedynczej ekspozycji" (one-shot learning), co jest krytyczne w środowiskach, gdzie błędy mogą być kosztowne.[2]

3. Strategia Technologiczna Bazy Danych: PostgreSQL i Wyszukiwanie Hybrydowe

Wybór PostgreSQL (w wariancie Vercel Postgres/Neon) jako silnika bazy danych jest decyzją strategiczną, podyktowaną potrzebą konsolidacji danych relacyjnych i wektorowych w jednym systemie.

3.1 Separacja Obliczeń i Pamięci (Neon Architecture)

Neon, jako bezserwerowy Postgres, oddziela warstwę obliczeniową (Compute) od warstwy przechowywania (Storage). Jest to idealne dla obciążeń generowanych przez agentów AI, które charakteryzują się dużą zmiennością (tzw. "bursty traffic"). Agent może przez godziny nie być aktywny, a następnie przeprowadzać intensywną sesję wnioskowania wymagającą tysięcy operacji wejścia/wyjścia (IOPS) w krótkim czasie. Neon pozwala na skalowanie do zera, co optymalizuje koszty, jednocześnie zapewniając natychmiastową dostępność.[17]

3.2 Wektoryzacja i Indeksowanie: HNSW kontra IVFFlat

Dla agenta Lolek, szybkość i precyzja wyszukiwania wspomnień są krytyczne. PostgreSQL oferuje rozszerzenie pgvector, które wspiera różne typy indeksów. Analiza wydajności i charakterystyki tych indeksów jest kluczowa dla projektu:

Cecha	Indeks HNSW (Hierarchical Navigable Small World)	Indeks IVFFlat (Inverted File with Flat Compression)	Rekomendacja dla Lolka
Złożoność budowy	Wysoka (dłuższy czas tworzenia indeksu)	Średnia (wymaga trenowania na próbce danych)	**HNSW**
Szybkość	Bardzo wysoka (logarytmiczna	Wysoka, ale zależna od liczby	**HNSW**

wyszukiwania	złożoność)	list	
Wpływ aktualizacji	Dobrze radzi sobie z częstymi insertami (incremental build)	Degraduje przy dużych zmianach dystrybucji danych	**HNSW**
Zużycie pamięci	Wyższe (wymaga więcej RAM dla grafu)	Niższe	**HNSW**

Zalecamy użycie indeksu HNSW. Mimo że jest on bardziej zasobożerny przy tworzeniu, jego zdolność do obsługi dynamicznie rosnącego strumienia pamięci bez konieczności częstego przebudowywania (re-indexing) jest kluczowa dla systemu działającego w czasie rzeczywistym. IVFFlat wymagałby okresowej konserwacji (reindeksacji po znaczącej zmianie wektorów), co mogłoby powodować przestoje w dostępności pamięci agenta.[18]

3.3 Wyszukiwanie Hybrydowe (Hybrid Search)

Samo wyszukiwanie wektorowe (semantyczne) ma ograniczenia. Często "gubi" ono precyzyjne słowa kluczowe (np. nazwy własne, unikalne identyfikatory) w szumie semantycznym. Przykładowo, zapytanie o "Błąd 503 w API płatności" może semantycznie pasować do ogólnych problemów z siecią, ale użytkownik szuka konkretnego kodu błędu.

Strategia dla Lolka zakłada **Wyszukiwanie Hybrydowe**, które łączy:

1. **Wyszukiwanie Semantyczne:** Wykorzystanie kolumny vector i operatora podobieństwa cosinusowego (<=>) do znalezienia koncepcyjnie zbliżonych wspomnień.
2. **Wyszukiwanie Pełnotekstowe (FTS):** Wykorzystanie wbudowanego w Postgres mechanizmu tsvector i tsquery (algorytm BM25) do dopasowania słów kluczowych.

Wyniki z obu metod będą łączone przy użyciu algorytmu **Reciprocal Rank Fusion (RRF)**. RRF normalizuje rankingi z obu źródeł, premiując dokumenty, które pojawiają się wysoko na obu listach, co znacząco zwiększa trafność (Recall) kontekstu dostarczanego do LLM.[21]

3.4 Wyzwania Integracji Prisma w Środowisku Edge

Użycie Vercel Edge Functions narzuca ograniczenia na połączenia bazodanowe. Standardowy sterownik pg dla Node.js opiera się na stałych połączeniach TCP, które nie są obsługiwane w środowisku V8 isolate (Edge Runtime).

Rozwiązaniem jest zastosowanie **Prisma Driver Adapter** dla Neon (@prisma/adapter-neon). Adapter ten przekierowuje zapytania SQL przez protokół HTTP/WebSocket (serverless driver), co pozwala na utrzymanie połączenia z bazą danych nawet w środowisku Edge. Jest to krytyczne dla minimalizacji czasu "zimnego startu" (cold start) funkcji serverless, co bezpośrednio przekłada się na responsywność agenta.[7]

4. Projekt Schematu Bazy Danych (Prisma Schema)

Poniższy schemat Prisma (schema.prisma) jest wynikiem syntezy wymagań kognitywnych i technologicznych. Uwzględnia on obsługę wektorów, partycjonowanie danych per użytkownik (multi-tenancy) oraz struktury dla mechanizmów refleksji.

4.1 Definicja Schematu

Code snippet

```
// schema.prisma

generator client {
  provider        = "prisma-client-js"
  // Włączenie podglądu funkcji rozszerzeń PostgreSQL, widoków i adapterów sterowników
  previewFeatures =
}

datasource db {
  provider   = "postgresql"
  url        = env("DATABASE_URL")
```

```
  // Bezpośredni URL potrzebny do migracji, które nie zawsze działają przez proxy HTTP
  directUrl  = env("DIRECT_DATABASE_URL")
  // Deklaracja używanych rozszerzeń: vector dla AI, pgcrypto dla szyfrowania
  extensions = [vector, pgcrypto]
}

// ------------------------------------------------------------------------------
// Model Użytkownika i Bezpieczeństwo (Multi-tenancy Root)
// ------------------------------------------------------------------------------

model User {
  id           String    @id @default(cuid())
  email        String    @unique
  name         String?
  createdAt    DateTime  @default(now())
  updatedAt    DateTime  @updatedAt

  // Relacje
  agents       Agent
  sessions     Session
  apiKeys      ApiKey

  @@map("users")
}

// ------------------------------------------------------------------------------
// Tożsamość Agenta i Konfiguracja
// ------------------------------------------------------------------------------

model Agent {
  id           String    @id @default(uuid())
  name         String
  description  String    @db.Text
  systemPrompt String    @db.Text // Główna instrukcja osobowości (Persona)

  // Konfiguracja modelu (JSONB pozwala na elastyczność bez migracji)
  // Przechowuje np. { "temperature": 0.7, "model": "gemini-1.5-pro", "top_k": 40 }
  modelConfig  Json

  userId       String
  user         User      @relation(fields: [userId], references: [id], onDelete: Cascade)

  // Stan Poznawczy
```

```
  memories      MemoryEvent
  goals         Goal
  documents     Document

  createdAt     DateTime  @default(now())
  updatedAt     DateTime  @updatedAt

  @@index([userId])
  @@map("agents")
}

// ------------------------------------------------------------------------------
// Strumień Pamięci (Episodic & Procedural Memory)
// ------------------------------------------------------------------------------

enum MemoryType {
  OBSERVATION // "Użytkownik powiedział X" lub "System wykrył upload pliku"
  THOUGHT     // Wewnętrzny monolog: "Powinienem sprawdzić bazę wiedzy"
  REFLECTION  // Zsyntetyzowany wniosek: "Użytkownik pracuje nad projektem finansowym"
  TOOL_CALL   // Zapis wywołania narzędzia (np. zapytanie SQL)
  TOOL_RESULT // Wynik działania narzędzia (np. zwrócone wiersze lub błąd)
}

model MemoryEvent {
  id          String      @id @default(uuid())
  agentId     String
  agent       Agent       @relation(fields: [agentId], references: [id], onDelete: Cascade)

  type        MemoryType
  content     String      @db.Text // Treść tekstowa wspomnienia

  // Metadane Kontekstowe
  importance  Int         @default(1) // Ocena istotności (1-10) nadawana przez LLM przy
tworzeniu [25]

  // Wektor Embeddingu dla Wyszukiwania Semantycznego
  // Typ Unsupported jest używany, ponieważ Prisma nie ma jeszcze pełnego mapowania typu
vector w kliencie JS
  // W migracjach SQL zostanie to zmapowane na vector(768) dla Gemini lub vector(1536) dla
OpenAI
  embedding   Unsupported("vector(768)")?

  // Metadane strukturalne (JSONB)
```

```
  // Przechowuje nazwy narzędzi, tagi encji, ID powiązanych wiadomości
  metadata    Json?

  createdAt   DateTime    @default(now())

  // Relacje dla Drzewa Refleksji (Reflection Trees)
  // Refleksja powstaje z wielu wspomnień (derivedFrom) i może przyczyniać się do nowych (contributedTo)
  derivedFrom MemoryRelation @relation("DerivedTo")
  contributedTo MemoryRelation @relation("ContributedFrom")

  // Indeksy
  @@index([agentId])
  @@index([type])
  // Indeks złożony dla optymalizacji sortowania po czasie w ramach agenta
  @@index()
  @@map("memory_events")
}

// Tabela złączeniowa dla relacji wiele-do-wielu między wspomnieniami
// Umożliwia śledzenie genealogii myśli (Provenance)
model MemoryRelation {
  sourceId    String
  targetId    String

  source      MemoryEvent @relation("ContributedFrom", fields: [sourceId], references: [id], onDelete: Cascade)
  target      MemoryEvent @relation("DerivedTo", fields: [targetId], references: [id], onDelete: Cascade)

  @@id([sourceId, targetId])
  @@map("memory_relations")
}

// ------------------------------------------------------------------------------
// Wiedza Semantyczna (RAG - Retrieval Augmented Generation)
// ------------------------------------------------------------------------------

model Document {
  id          String   @id @default(uuid())
  agentId     String
  agent       Agent    @relation(fields: [agentId], references: [id], onDelete: Cascade)
```

```
  title      String
  content    String?  @db.Text // Pełna treść (opcjonalnie, jeśli przechowujemy tylko chunki)
  sourceUrl  String?  // Źródło pochodzenia (URL, ścieżka pliku)

  chunks     DocumentChunk

  createdAt  DateTime @default(now())
  @@map("documents")
}

model DocumentChunk {
  id         String   @id @default(uuid())
  documentId String
  document   Document @relation(fields: [documentId], references: [id], onDelete: Cascade)

  content    String   @db.Text
  chunkIndex Int      // Kolejność fragmentu w dokumencie

  // Wektor dla fragmentu
  embedding  Unsupported("vector(768)")?

  // UWAGA: Indeks full-text search (tsvector) zostanie dodany przez Raw SQL migration
  // Prisma nie wspiera jeszcze w pełni generowanych kolumn tsvector w schemacie

  @@map("document_chunks")
}

// ------------------------------------------------------------------------------
// Planowanie i Cele (Forward-Looking State)
// ------------------------------------------------------------------------------

enum GoalStatus {
  PENDING
  IN_PROGRESS
  COMPLETED
  FAILED
  BLOCKED
}

model Goal {
  id         String     @id @default(uuid())
  agentId    String
  agent      Agent      @relation(fields: [agentId], references: [id], onDelete: Cascade)
```

```
  description String     @db.Text
  status      GoalStatus @default(PENDING)
  priority    Int        @default(1)

  // Cele hierarchiczne (Drzewo Celów)
  parentId    String?
  parent      Goal?      @relation("SubGoals", fields: [parentId], references: [id])
  subGoals    Goal     @relation("SubGoals")

  createdAt   DateTime   @default(now())
  updatedAt   DateTime   @updatedAt

  @@map("goals")
}

// ------------------------------------------------------------------------------
// Bezpieczeństwo i Sekrety
// ------------------------------------------------------------------------------

model ApiKey {
  id          String   @id @default(uuid())
  userId      String
  user        User     @relation(fields: [userId], references: [id], onDelete: Cascade)

  provider    String   // np. "OPENAI", "GOOGLE", "AWS"
  keyId       String   // Identyfikator klucza (nie tajny) dla celów rotacji

  // Zaszyfrowana wartość - NIGDY nie przechowuj kluczy jawnym tekstem
  // Szyfrowanie po stronie aplikacji (AES-256-GCM)
  value       String   @db.Text
  iv          String   // Wektor inicjalizujący (Initialization Vector)

  createdAt   DateTime @default(now())
  expiresAt   DateTime? // Data wygaśnięcia wymuszająca rotację

  @@map("api_keys")
}

model Session {
  id          String   @id @default(cuid())
  sessionToken String   @unique
  userId      String
```

```
  expires     DateTime
  user        User     @relation(fields: [userId], references: [id], onDelete: Cascade)

  @@map("sessions")
}
```

4.2 Analiza Krytyczna Schematu

4.2.1 Model MemoryEvent i Pole importance

Tabela MemoryEvent jest sercem systemu. Nie rozdzielamy sztucznie historii czatu od wewnętrznych przemyśleń. Dzięki temu agent, analizując kontekst, ma dostęp do pełnego obrazu sytuacji.
Kluczowym elementem jest pole importance (typ Int). Zgodnie z badaniami Park et al. (Stanford), każde wspomnienie musi zostać ocenione przez LLM w momencie jego powstania. Trywialne powitanie ("Cześć") może otrzymać ocenę 1, podczas gdy informacja o alergii użytkownika otrzyma ocenę 9 lub 10. To pole jest niezbędne do algorytmu wyszukiwania, który zapobiega wypieraniu ważnych, ale starych informacji przez nowe, ale trywialne zdarzenia (problem recency bias).25

4.2.2 Unsupported("vector(768)") i Migracje SQL

Prisma, mimo dynamicznego rozwoju, wciąż traktuje typy natywne PostgreSQL jako funkcję eksperymentalną. Użycie Unsupported jest bezpiecznym podejściem, które pozwala na definicję pola w modelu, ale deleguje zarządzanie jego strukturą do surowego SQL. Wymiar 768 został dobrany pod modele embeddingowe Google Gemini (np. text-embedding-004). W przypadku zmiany modelu na OpenAI text-embedding-3-small (1536 wymiarów) lub large (3072), konieczna będzie zmiana tego parametru.

Aby schemat zadziałał w praktyce, po wygenerowaniu migracji (npx prisma migrate dev --create-only), należy ręcznie zmodyfikować plik SQL, aby dodać indeksy HNSW, których Prisma Schema Language (PSL) nie potrafi jeszcze wyrazić.

SQL

```sql
-- migration.sql (edycja ręczna)

-- Włączenie rozszerzenia vector
CREATE EXTENSION IF NOT EXISTS vector;

-- Tworzenie indeksu HNSW dla tabeli memory_events
-- Parametr 'm': maksymalna liczba połączeń na element (wyższa = lepsza jakość, wolniejsze budowanie)
-- Parametr 'ef_construction': rozmiar dynamicznej listy kandydatów podczas budowy (wyższa = lepsza jakość)
CREATE INDEX memory_embedding_idx ON memory_events
USING hnsw (embedding vector_cosine_ops)
WITH (m = 16, ef_construction = 64);

-- Dodanie kolumny tsvector dla wyszukiwania hybrydowego w document_chunks
ALTER TABLE document_chunks
ADD COLUMN fts_vector tsvector
GENERATED ALWAYS AS (to_tsvector('polish', content)) STORED;

-- Indeks GIN dla szybkiego wyszukiwania pełnotekstowego
CREATE INDEX document_chunks_fts_idx ON document_chunks USING GIN (fts_vector);
```

4.2.3 Drzewo Celów (Goal)

Zdolność planowania jest realizowana przez tabelę Goal z autoreferencją (parentId). Pozwala to na dekompozycję złożonych zadań (np. "Stwórz raport roczny") na podzadania ("Pobierz dane z API", "Przeanalizuj trendy", "Wygeneruj wykresy"). Statusy zadań (PENDING,

IN_PROGRESS, etc.) pozwalają agentowi śledzić postępy i wznawiać pracę po przerwie, co jest kluczowe dla długoterminowej autonomii.[15]

5. Strategia Zarządzania Danymi: Pipeline Ingestii i Retriewalu

Sam schemat bazy danych jest bezużyteczny bez odpowiedniej logiki zarządzania przepływem informacji. Poniżej przedstawiono algorytmy sterujące pamięcią agenta Lolek.

5.1 Pipeline Ingestii (Zapisywanie Wspomnień)

Gdy użytkownik wysyła wiadomość, proces jej zapisu nie jest atomową operacją INSERT. Przebiega on w następujących krokach:

1. **Rejestracja:** Wiadomość jest wstępnie zapisywana w MemoryEvent z typem OBSERVATION.
2. **Ewaluacja Ważności (Importance Scoring):** Równolegle, system wysyła treść wiadomości do lekkiego modelu LLM (np. Gemini Flash) z promptem: *"W skali od 1 do 10, oceń jak ważna jest ta informacja dla długoterminowej pamięci agenta. Zwróć tylko liczbę."*. Wynik jest zapisywany w polu importance.
3. **Osadzanie (Embedding):** Treść jest wysyłana do modelu embeddingowego (Gemini Embeddings), a zwrócony wektor jest aktualizowany w rekordzie.
4. **Generowanie Myśli (Immediate Reaction):** Jeśli waga > 7, agent może natychmiast wygenerować wewnętrzną myśl (THOUGHT) powiązaną z tą obserwacją, co również trafia do strumienia.[9]

5.2 Strategia Retriewalu (Konstrukcja Kontekstu)

Największym wyzwaniem jest wybór podzbioru wspomnień, które zostaną przesłane do LLM w oknie kontekstowym ("Context Stuffing"). Nie możemy po prostu pobrać "ostatnich 10 wiadomości". Musimy zastosować algorytm punktacji (Scoring Function), inspirowany pracą *Generative Agents*.

Wzór na wynik relewancji wspomnienia S dla danego zapytania q:

$$S = \alpha \cdot R(czas) + \beta \cdot I(ważność) + \gamma \cdot Sim(q, m)$$
Gdzie:

- $R(czas)$ (Recency): Funkcja wykładniczego zaniku. Wspomnienia świeże mają wyższy wynik.

  $$R(t) = (1 - decay)^{godziny\_od\_utworzenia}$$

  Typowa wartość $decay$ to 0.005 na godzinę, co sprawia, że wspomnienia sprzed tygodnia tracą ok. 50% wagi "świeżości", chyba że są bardzo ważne.
- **$I(ważność)$ (Importance):** Znormalizowana wartość pola importance z bazy danych.
- **$Sim(q, m)$ (Relevance):** Podobieństwo cosinusowe wektorów (zwracane przez operator <=> w pgvector).

Wartości współczynników α, β, γ (wagi) należy dostroić eksperymentalnie. Sugerowane wartości startowe to: $\alpha=1.0, \beta=0.5, \gamma=1.5$. Oznacza to, że semantyczne dopasowanie jest najważniejsze, ale bardzo ważne wspomnienia z przeszłości mogą "przebić" mniej istotne, nowsze zdarzenia.[25]

Implementacja w Prisma (TypedSQL)

Ze względu na złożoność obliczeń matematycznych, najlepiej zaimplementować tę logikę jako funkcję SQL (Stored Procedure) lub użyć queryRaw (w nowszych wersjach Prisma TypedSQL dla bezpieczeństwa typów).

TypeScript

```
// Przykład koncepcyjny użycia Raw SQL dla wydajności
const embedding = await generateEmbedding(userQuery);
const memories = await prisma.$queryRaw`
  SELECT id, content, type, createdAt,
  (
```

```
  1.0 * power(0.995, EXTRACT(EPOCH FROM (NOW() - "createdAt")) / 3600) + -- Recency
  0.5 * ("importance" / 10.0) +                                  -- Importance
  1.5 * (1 - ("embedding" <=> ${embedding}::vector))             -- Relevance (Cosine Sim)
 ) as score
 FROM "memory_events"
 WHERE "agentId" = ${agentId}
 ORDER BY score DESC
 LIMIT 20;
`;
```

5.3 Pętla Refleksji i Samokorekty

Proces refleksji nie może być wyzwalany przy każdym zapytaniu użytkownika ze względu na opóźnienia. Powinien być procesem asynchronicznym (Background Job).

1. **Harmonogram:** Vercel Cron uruchamia endpoint /api/cron/reflect co godzinę lub po przekroczeniu progu sumarycznej wagi nowych wspomnień (np. gdy suma importance nowych zdarzeń przekroczy 100).
2. **Synteza:** Agent pobiera 50 ostatnich nieprzetworzonych obserwacji.
3. **Prompt:** *"Biorąc pod uwagę te 50 zdarzeń, jakie ogólne prawdy o użytkowniku lub świecie możesz wywnioskować? Sformułuj 3-5 refleksji."*
4. **Zapis:** Nowe refleksje są zapisywane jako MemoryEvent (typ REFLECTION) i łączone relacjami MemoryRelation z obserwacjami źródłowymi. To pozwala agentowi "zapominać" szczegóły (niski recency), ale pamiętać wnioski (odświeżony recency nowej refleksji).[11]

Dla mechanizmu **Reflexion** (samokorekty), jeśli agent wykonuje TOOL_CALL (np. SQL Query), a TOOL_RESULT zwraca błąd, system automatycznie generuje nowy THOUGHT z treścią błędu i instrukcją: *"Poprzednia próba nie powiodła się z powodu X. W następnym kroku spróbuj strategii Y."*. Ta sekwencja jest zapisywana, tworząc historię uczenia się.[16]

6. Bezpieczeństwo i Operacjonalizacja

6.1 Szyfrowanie Pola (Field-Level Encryption)

Baza danych zawiera klucze API użytkowników do usług zewnętrznych. Nawet przy szyfrowaniu dysku (TDE) w chmurze, dostęp administratora do bazy mógłby ujawnić te sekrety.
Należy zastosować szyfrowanie warstwy aplikacji. Klucze API powinny być szyfrowane algorytmem AES-256-GCM przed wysłaniem do bazy. Wektor inicjalizujący (IV) musi być unikalny dla każdego rekordu i przechowywany w kolumnie iv. Klucz szyfrujący (ENCRYPTION_KEY) musi znajdować się wyłącznie w zmiennych środowiskowych Vercel i nigdy nie trafiać do repozytorium kodu ani bazy danych.30
Dla Prisma zalecane jest użycie middleware (w starszych wersjach) lub rozszerzenia klienta ($extends), które automatycznie szyfruje pole value w modelu ApiKey przy zapisie i deszyfruje przy odczycie.

6.2 Rotacja Kluczy i Bezpieczeństwo OAuth

W przypadku tokenów OAuth (tabela Session i integracje), należy wdrożyć politykę krótkiego czasu życia (short-lived tokens) i używać Refresh Tokens. Schemat zawiera pole expiresAt w ApiKey, co umożliwia implementację logiki (np. Cron Job), która sprawdza klucze zbliżające się do wygaśnięcia i inicjuje procedurę odświeżenia lub powiadamia użytkownika.[32]

6.3 Monitorowanie i Logowanie Narzędzi

Każde użycie narzędzia przez agenta jest potencjalnym wektorem ataku (np. Prompt Injection prowadzące do wycieku danych przez SQL Injection w wygenerowanym zapytaniu).
Tabela MemoryEvent z typem TOOL_CALL pełni rolę audytową. Należy wdrożyć mechanizm "Red Teaming" lub "Guardrails" (np. przy użyciu biblioteki NeMo Guardrails lub własnych reguł walidacji w Zod), który sprawdza parametry wywołania narzędzia zanim zostanie ono wykonane. Logi tych weryfikacji również powinny trafiać do strumienia pamięci, aby agent

"wiedział", że próbował wykonać niedozwoloną akcję.12

7. Podsumowanie i Rekomendacje Wdrożeniowe

Przedstawiony projekt transformuje agenta "Lolek" z prostej aplikacji reaktywnej w system posiadający ciągłość poznawczą. Kluczowe filary tej architektury to:

1. **Hybrydowy Model Danych:** Połączenie ustrukturyzowanych relacji SQL (dla użytkowników i sesji) z nieustrukturyzowanym, wektorowym strumieniem pamięci (dla kognitywistyki agenta).
2. **Zaawansowany Retriewal:** Zastąpienie prostego wyszukiwania semantycznego ważonym algorytmem uwzględniającym czas, wagę i relewancję, zaimplementowanym bezpośrednio w silniku bazy danych dla wydajności.
3. **Adaptacja do Edge:** Wykorzystanie adapterów Neon i Prisma do przezwyciężenia limitów połączeń w środowisku serverless.

Rekomendowane Kroki Następne:

1. **Prototypowanie Funkcji Punktacji:** Implementacja i testy obciążeniowe zapytania SQL obliczającego Score na zbiorze 100,000 symulowanych wspomnień.
2. **Implementacja Crona Refleksji:** Uruchomienie procesu tła w Vercel Cron, który będzie cyklicznie syntetyzował pamięć.
3. **Audyt Bezpieczeństwa:** Weryfikacja implementacji szyfrowania AES-256 oraz testy penetracyjne pod kątem wycieku kluczy API.

Taka architektura zapewnia skalowalność, bezpieczeństwo i - co najważniejsze - głębię interakcji, która jest niezbędna dla prawdziwie autonomicznego agenta AI.

Works cited

1. Position: Episodic Memory is the Missing Piece for Long-Term LLM Agents - arXiv, accessed November 25, 2025, https://arxiv.org/html/2502.06975v1
2. Position: Episodic Memory is the Missing Piece for Long-Term LLM Agents - arXiv, accessed November 25, 2025, https://arxiv.org/pdf/2502.06975?
3. MemGPT: Towards LLMs as Operating Systems - Leonie Monigatti, accessed November 25, 2025, https://www.leoniemonigatti.com/papers/memgpt.html
4. MemGPT with Real-life Example: Bridging the Gap Between AI and OS | DigitalOcean, accessed November 25, 2025, https://www.digitalocean.com/community/tutorials/memgpt-llm-infinite-context-

understanding
5. MIRIX: Multi-Agent Memory System for LLM-Based Agents - arXiv, accessed November 25, 2025, https://arxiv.org/html/2507.07957v1
6. Generative Agents: Interactive Simulacra of Human Behavior - arXiv, accessed November 25, 2025, https://arxiv.org/pdf/2304.03442
7. Deploy to Vercel Edge Functions & Middleware | Prisma Documentation, accessed November 25, 2025, https://www.prisma.io/docs/orm/prisma-client/deployment/edge/deploy-to-vercel
8. Overview: Deploy Prisma ORM at the Edge, accessed November 25, 2025, https://www.prisma.io/docs/orm/prisma-client/deployment/edge/overview
9. AI SDK - Vercel, accessed November 25, 2025, https://vercel.com/docs/ai-sdk
10. Tool Calling - AI SDK Core, accessed November 25, 2025, https://ai-sdk.dev/docs/ai-sdk-core/tools-and-tool-calling
11. [2304.03442] Generative Agents: Interactive Simulacra of Human Behavior - arXiv, accessed November 25, 2025, https://arxiv.org/abs/2304.03442
12. Computational Agents Exhibit Believable Humanlike Behavior | Stanford HAI, accessed November 25, 2025, https://hai.stanford.edu/news/computational-agents-exhibit-believable-humanlike-behavior
13. Paper Walkthrough: Generative Agents: Interactive Simulcra of Human Behavior - Medium, accessed November 25, 2025, https://medium.com/@marekpaulik/generative-agents-interactive-simulcra-of-human-behavior-648c32a76b9
14. Reflexion - GitHub Pages, accessed November 25, 2025, https://langchain-ai.github.io/langgraph/tutorials/reflexion/reflexion/
15. Reflection Agents - LangChain Blog, accessed November 25, 2025, https://blog.langchain.com/reflection-agents/
16. Training AI Agents to Write and Self-correct SQL with Reinforcement Learning - Medium, accessed November 25, 2025, https://medium.com/@yugez/training-ai-agents-to-write-and-self-correct-sql-with-reinforcement-learning-571ed31281ad
17. How to set up HNSW index? (Nextjs, Prisma, Langchain) - Neon - Answer Overflow, accessed November 25, 2025, https://www.answeroverflow.com/m/1304458054628278272
18. Optimize generative AI applications with pgvector indexing: A deep dive into IVFFlat and HNSW techniques | AWS Database Blog, accessed November 25, 2025, https://aws.amazon.com/blogs/database/optimize-generative-ai-applications-with-pgvector-indexing-a-deep-dive-into-ivfflat-and-hnsw-techniques/
19. Understanding vector search and HNSW index with pgvector - Neon, accessed November 25, 2025, https://neon.com/blog/understanding-vector-search-and-hnsw-index-with-pgvector
20. HNSW Indexes with Postgres and pgvector | Crunchy Data Blog, accessed November 25, 2025,

https://www.crunchydata.com/blog/hnsw-indexes-with-postgres-and-pgvector
21. Hybrid search | Supabase Docs, accessed November 25, 2025, https://supabase.com/docs/guides/ai/hybrid-search
22. Hybrid Search in PostgreSQL: The Missing Manual - ParadeDB, accessed November 25, 2025, https://www.paradedb.com/blog/hybrid-search-in-postgresql-the-missing-manual
23. Hybrid search with PostgreSQL and pgvector - Jonathan Katz, accessed November 25, 2025, https://jkatz05.com/post/postgres/hybrid-search-postgres-pgvector/
24. Prisma Adapter with Auth.js in Next.js 15 on Vercel (Edge Runtime Issue) - Stack Overflow, accessed November 25, 2025, https://stackoverflow.com/questions/79639728/prisma-adapter-with-auth-js-in-next-js-15-on-vercel-edge-runtime-issue
25. 10: Memory & Retrieval for LLMs — OscarAI - Oscar Health, accessed November 25, 2025, https://www.hioscar.ai/10-memory-and-retrieval-for-llms
26. Integrating large language model-based agents into a virtual patient chatbot for clinical anamnesis training - PMC - NIH, accessed November 25, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC12180958/
27. LLM Powered Autonomous Agents | Lil'Log, accessed November 25, 2025, https://lilianweng.github.io/posts/2023-06-23-agent/
28. Agentic AI Systems Applied to tasks in Financial Services: Modeling and model risk management crews - arXiv, accessed November 25, 2025, https://arxiv.org/html/2502.05439v1
29. Building a Self-Correcting AI: A Deep Dive into the Reflexion Agent with LangChain and LangGraph | by Vi Q. Ha | Medium, accessed November 25, 2025, https://medium.com/@vi.ha.engr/building-a-self-correcting-ai-a-deep-dive-into-the-reflexion-agent-with-langchain-and-langgraph-ae2b1ddb8c3b
30. Transparent field-level encryption at rest for Prisma - GitHub, accessed November 25, 2025, https://github.com/47ng/prisma-field-encryption
31. Data Encryption and Hashing: A Scenario using NestJS + Prisma App | by Bonaventuragal, accessed November 25, 2025, https://medium.com/@bonaventuragal/data-encryption-and-hashing-a-scenario-using-nestjs-prisma-app-902b43530dbb
32. Operate Efficiently and Securely: Rotating Prisma Cloud Access Keys - Palo Alto Networks, accessed November 25, 2025, https://www.paloaltonetworks.com/blog/cloud-security/operate-efficiently-and-securely-rotating-prisma-cloud-access-keys/
33. 7 Tips to Build Self-Improving AI Agents with Feedback Loops | Datagrid, accessed November 25, 2025, https://www.datagrid.com/blog/7-tips-build-self-improving-ai-agents-feedback-loops