ChatGPT

Projekt architektury pamięci agenta Lolek

Aby agent Lolek mógł zapamiętywać epizody, fakty i procedury oraz bezpiecznie integrować narzędzia, proponujemy kompleksową strukturę bazy danych rozszerzoną o różne rodzaje pamięci. W schemacie Prisma uwzględniamy modele do przechowywania: historii konwersacji, **pamięci epizodycznej** (wydarzeń), **semantycznej** (faktów z embeddingami wektorowymi) i **proceduralnej** (reguł, instrukcji), a także długoterminowych zadań i logów decyzji. Dodatkowo projektujemy tabele **Dokumentów** z polami tekstowymi i wektorami (pgvector) oraz **Narzędzi i Poświadczeń**, by agent mógł bezpiecznie korzystać z Gmaila, GCP itp. Poniżej kluczowe modele Prisma:

```
model Conversation {
  id         Int       @id @default(autoincrement())
  title      String?
  createdAt  DateTime  @default(now())
  messages   Message[]
}

model Message {
  id             Int          @id @default(autoincrement())
  conversation   Conversation @relation(fields: [conversationId], references:
[id])
  conversationId Int
  role           String       // 'user', 'assistant', 'system' itp.
  content        String
  timestamp      DateTime     @default(now())
}

model EpisodicMemory {
  id            Int       @id @default(autoincrement())
  eventTime     DateTime  // czas zdarzenia
  description   String    // opis zdarzenia (np. "Wgrałem plik X")
  conversation   Conversation? @relation(fields: [conversationId], references:
[id])
  conversationId Int?
  createdAt     DateTime  @default(now())
}

model SemanticMemory {
  id        Int      @id @default(autoincrement())
  fact      String   // ustrukturyzowana informacja/fakt
  embedding Unsupported("vector(1536)")  // wektor semantyczny (pgvector)
  createdAt DateTime @default(now())
}
```

```
model ProceduralMemory {
  id          Int      @id @default(autoincrement())
  instruction String   // procedura/reguła („gdy pytanie o raport, sprawdz
tabelę Y”)
  createdAt   DateTime @default(now())
}

model Document {
  id        Int      @id @default(autoincrement())
  title     String?
  content   String   // zawartość dokumentu lub metadane
  embedding Unsupported("vector(1536)")  // wektor do wyszukiwania semantycznego
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
}

model LongTermTask {
  id          Int      @id @default(autoincrement())
  title       String
  description String?
  status      String   // np. 'PENDING', 'IN_PROGRESS', 'DONE'
  createdAt   DateTime @default(now())
  dueDate     DateTime?
}

model DecisionLog {
  id           Int      @id @default(autoincrement())
  decisionTime DateTime @default(now())
  context      String   // opis sytuacji lub zapytania
  action       String   // podjęta akcja
  rationale    String?  // (opcjonalne) „rozumowanie” agenta
  outcome      String   // wynik (sukces, błąd itp.)
  task         LongTermTask? @relation(fields: [taskId], references: [id])
  taskId       Int?
}

model Tool {
  id          Int     @id @default(autoincrement())
  name        String  // np. 'Gmail', 'GCP'
  description String?
  config      Json?   // konfiguracja API lub OAuth
}

model Credential {
  id           Int     @id @default(autoincrement())
  tool         Tool    @relation(fields: [toolId], references: [id])
  toolId       Int
```

```
  accessToken  String  // zaszyfrowany token dostępu
  refreshToken String?
  expiresAt    DateTime?
  scopes       String? // zakresy uprawnień OAuth
}
```

- **Pamięć epizodyczna:** tabela `EpisodicMemory` loguje konkretne zdarzenia z agentem – czas i opis wykonanej czynności. Wspomnienia epizodyczne przechowujemy w ustrukturyzowany sposób, jak sugeruje literatura [1]. Na przykład przy wgraniu pliku X dodajemy nowy wpis z `eventTime` i `description`. Agent może przeszukiwać te wpisy np. SQL-owo (filtrując po dacie, słowach kluczowych), aby przypomnieć sobie konkretne incydenty [1].
- **Pamięć semantyczna:** tabela `SemanticMemory` przechowuje uogólnione fakty i relacje wraz z embeddingiem wektorowym. Dzięki kolumnie `embedding` (pgvector) i indeksowi wektorowemu agent wykonuje semantyczne wyszukiwanie (RAG) w zbiorze wiedzy [2] [3]. W praktyce pytając o daną informację, agent najpierw konwertuje zapytanie na wektor i znajduje podobne wektory w `SemanticMemory`, a następnie łączy wynik z filtrem kontekstowym (np. kontekst dziedziny). Takie semantyczne wyszukiwanie wspiera uogólnienie faktów [2].

- **Pamięć proceduralna:** tabela `ProceduralMemory` zawiera instrukcje lub reguły realizacji zadań (np. „gdy pytam o raport, sprawdź tabelę Y"). To zapisane procedury działania nawykowego agentów – analogiczne do ludzkiej pamięci proceduralnej [4] [5]. Każdy wpis to gotowa reguła lub szablon działania, który agent może zastosować bez ponownego wnioskowania (np. na podstawie `ProceduralMemory.instruction`).

- **Dokumenty (wiersz z wektorem):** tabela `Document` przechowuje ważne pliki i notatki (e-maile, raporty, kod itp.) wraz z embeddingiem. Dzięki temu agent może przeszukiwać dokumenty zarówno klasycznie (np. pełnotekstowo przez SQL/GIN/BM25), jak i semantycznie (przez pgvector) [6] [7]. Można np. filtrować dokumenty po dacie nadania lub nadawcy (SQL), a następnie zebrać najbardziej podobne wektorowo do zapytania. Wykorzystujemy hybrydowe przeszukiwanie: kombinujemy przeszukiwanie leksykalne z wektorowym, co poprawia precyzję i recall [6] [7].

- **Zadania długoterminowe:** tabela `LongTermTask` śledzi cele i zadania wieloetapowe (np. „Wdrożenie aplikacji na Vercel"). Pozwala planować i śledzić status działań wykonywanych przez agenta. Agent zapisuje tam zadania wygenerowane przez użytkownika lub siebie, co stanowi element jego długoterminowej pamięci planistycznej.

- **Logi decyzji:** `DecisionLog` rejestruje każdą podjętą przez agenta decyzję – czas, kontekst (np. zapytanie lub problem), wykonaną akcję, uzasadnienie (rationale) i wynik. Dzięki temu po błędzie można przeanalizować, dlaczego agent zdecydował tak, a nie inaczej, oraz zapisać lekcję z tej sytuacji. Przykładowo, jeśli deploy na Vercel zakończył się niepowodzeniem, wpisujemy log z opisem błędu i podjętymi krokami. Po poprawie wnioskujemy nowe **lekcje** – mogą być zapisane w `ProceduralMemory` lub w nowej tabeli `Lesson` wiążącej się z logiem błędu. Na podstawie literatury AI wiemy, że kluczowe jest śledzenie rezultatów poszczególnych decyzji, by móc wyciągać wnioski [8] [9].

Każdy model Prisma ma indeksy i więzy zgodne z potrzebami (np. pełnotekstowe GIN na polach tekstowych, HNSW na kolumnie `embedding`), by przyspieszyć zapytania. Dla pamięci semantycznej i dokumentów

używamy rozszerzenia pgvector (wektor długości np. 1536) [10], a w Prisma deklarujemy je jako `Unsupported("vector(1536)")` (w przyszłości dostępne wprost) [11]. Dzięki takiej strukturze agent Lolek posiada **trzy warstwy pamięci** – epizodyczną, semantyczną i proceduralną – jak opisują źródła [1] [3] [5].

Hybrydowa strategia wyszukiwania

Aby agent odpowiadał precyzyjnie na pytania (np. „znajdź maila z wczoraj"), łączymy **wyszukiwanie klasyczne** z **wektorowym** (hybrydowe RAG). Przykładowy scenariusz: najpierw filtrujemy dane za pomocą SQL (np. `WHERE date = yesterday` lub zapytania pełnotekstowego LIKE/BM25 po temacie/treści), co gwarantuje ścisłe spełnienie kryteriów. Następnie na uzyskanym zestawie wykonujemy wyszukiwanie semantyczne: konwertujemy zapytanie do wektora i znajdujemy najbardziej podobne embeddingi w `Document` / `SemanticMemory`. Jest to podejście zgodne z opisem „hybrid search" – łączy moc semantyki (cosine) z precyzją leksykalną (BM25) [7] [6].

Konkretne wdrożenie: w Postgresie można utworzyć indeksy GIN na tekst (pełnotekstowo) oraz HNSW na embeddingach (pgvector). Przy wyszukiwaniu agent najpierw wykonuje zapytanie tekstowe (np. `to_tsvector + @@` lub `ILIKE`) by znaleźć kandydatów, a następnie doprecyzowuje wynik według odległości wektorowej. Wyniki można łączyć algorytmem RRF (Reciprocal Rank Fusion), jak pokazuje literatura [12]. Dzięki temu agent „nie zgaduje" tylko najbliższego znaczeniowo – daje priorytet trafności kontekstowej. Takie hybrydowe wyszukiwanie jest obecnie standardem np. w Google Search [13] [14] i chmurowych usługach vektora [15] (Vertex AI Vector Search oferuje wbudowane zapytania hybrydowe) [16] [6].

Integracja narzędzi i bezpieczeństwo tokenów

Agent Lolek będzie korzystał z zewnętrznych API (Gmail, GitHub, GCP itd.), więc potrzebujemy bezpiecznie przechowywać **definicje narzędzi** i **poświadczenia**. W schemacie mamy model `Tool` (np. nazwa narzędzia, endpoint, JSON z konfiguracją OAuth) oraz `Credential` (związane z narzędziem, przechowujące tokeny). Kluczowe zasady bezpieczeństwa to:

- **Least privilege i zakresy:** dla każdego tokena definiujemy ograniczony zakres (scopes). Zgodnie z najlepszymi praktykami OAuth każdemu tokenowi nadajemy tylko potrzebne uprawnienia [17]. Dzięki temu agent może np. czytać maile (`gmail.readonly`), ale nie wysyłać czy kasować niczego bez zgody. Zakresy zapisywane są w polu `Credential.scopes`.
- **Bezpieczne przechowywanie:** tokeny trzymamy w zaszyfrowanej postaci w bazie lub (jeszcze lepiej) w osobnym „secret vault" (np. HashiCorp Vault lub Google Secret Manager). Przykładowo, w modelu `Credential.accessToken` powinien być przechowywany szyfrogram. Agent podczas działania pobiera token z bezpiecznego źródła. Wzorem rozwiązań rynkowych (np. Auth0 Token Vault) po autoryzacji użytkownika tokeny zapisywane są w izolowanym magazynie [18] [19]. Następnie agent żąda dostępu jedynie do poświadczeń bieżącego użytkownika [19]. Taki wzorzec zapobiega wyciekom i atakom typu „confused deputy" – każdy token jest związany z tożsamością i zakresem.
- **Konfiguracja narzędzi:** model `Tool` może zawierać identyfikatory klienta OAuth, sekretne klucze itp., ale prawdziwe klucze nigdy nie są trzymane w jawnej postaci. Lepiej odwoływać się do nich poprzez odwołania do bezpiecznego magazynu. Agent korzysta z definicji `Tool` w bazie, a gdy potrzebuje wykonać akcję, pobiera zaszyfrowany token z `Credential` i używa go w żądaniu API.

Dzięki takiej strukturze agent Lolek „wie”, jakie narzędzie ma do dyspozycji i jakie ma do nich uprawnienia, ale nie ujawnia surowych tokenów. Całe powiązanie `Tool` – `Credential` umożliwia implementację mechanizmu *token vault*, gdzie agent otrzymuje token wyłącznie dla właściwego użytkownika i z zasięgiem określonym w `Credential.scopes` [19] [17].

Mechanizm samokorekty (Self-Correction Loop)

Aby agent nie powtarzał tych samych błędów (np. nieprawidłowego deployu), potrzebujemy pętli sprzężenia zwrotnego. W bazie dane te realizujemy głównie przez `DecisionLog` i powiązane wpisy „lekcyjne”. Po wykryciu błędu agent generuje wpis w `DecisionLog` z kontekstem i szczegółami nieudanego działania. Następnie system analizy (lub sam agent) dopisuje wnioski – np. jako nową regułę w `ProceduralMemory` lub rekord w dodatkowej tabeli „LessonsLearned” wskazujący, czego unikać.

Kluczowe zasady architektoniczne (wg. praktyk budowania samodoskonalących się agentów [9] [8]) to:

- **Wersjonowanie pamięci:** każdą propozycję aktualizacji pamięci (np. zmianę `ProceduralMemory` lub dodanie nowego faktu) wprowadzamy dopiero po weryfikacji. Jeżeli feedback biznesowy pokazuje, że poprawka rozwiązała problem, zatwierdzamy ją; w przeciwnym razie wracamy do poprzedniego stanu [9]. Przykładowo, po naprawieniu błędu deployu agent może oznaczyć “złego” wpisu jako nieaktualny i utworzyć poprawny.
- **Audyt i śledzenie:** logi `DecisionLog` łączą wyniki z wcześniejszymi krokami agenta, co umożliwia analizę przyczyn błędów [8]. Dzięki temu agent „wie”, które działania doprowadziły do fiaska i unika ich w przyszłości. Na podstawie logów można automatycznie dodawać odpowiednie reguły do `ProceduralMemory` lub adnotacje do zadań.

Dzięki tym mechanizmom agent Lolek uczy się na doświadczeniach: nie tylko rejestruje błędy, ale też adaptuje swoje reguły działania. W efekcie kolejne próby działania (np. kolejny deploy na Vercel) są podejmowane z nową wiedzą i mniejszym ryzykiem powtórzenia problemu [9] [8].

Podsumowanie: powyższy schemat Prisma oraz strategia zarządzania danymi budują „drugi mózg” agenta: wielowarstwową pamięć (epizodyczną, semantyczną, proceduralną) oraz hybrydowy mechanizm wyszukiwania, wspierany bezpiecznym systemem integracji narzędzi i pętlą samokorekty. Taka architektura pozwoli Lolkowi efektywnie zapamiętywać informacje z przeszłości, korzystać z nich kontekstowo oraz uczyć się na własnych błędach, zgodnie z najlepszymi praktykami agentów AI [1] [7] [19] [9].

Źródła: Przykłady i definicje pamięci pobrano z artykułów IBM (AI Agent Memory) [1] [3] [4] oraz opracowań o architekturze agentów AI [2] [9]. Wskazówki dotyczące hybrydowego wyszukiwania pochodzą z publikacji o RAG [7] [6], a zabezpieczenia tokenów – z praktyk bezpieczeństwa OAuth dla agentów [18] [17].

---

[1] [3] [4] What Is AI Agent Memory? | IBM
https://www.ibm.com/think/topics/ai-agent-memory

[2] [5] AI Agent Memory: Short/Long Term, RAG, Agentic RAG
https://www.decodingai.com/p/memory-the-secret-sauce-of-ai-agents

6 Hybrid search with PostgreSQL and pgvector | Jonathan Katz
https://jkatz05.com/post/postgres/hybrid-search-postgres-pgvector/

7 12 Hybrid Retrieval in RAG: Going Beyond Vector Search for Actionable Results | Medium
https://medium.com/@clearmindrocks/hybrid-retrieval-in-rag-going-beyond-vector-search-for-actionable-results-940be6036435

8 9 How to Build Self-Improving AI Agents through Feedback Loops | Datagrid
https://datagrid.com/blog/7-tips-build-self-improving-ai-agents-feedback-loops

10 ORM 6.13.0, CI/CD Workflows & pgvector for Prisma Postgres
https://www.prisma.io/blog/orm-6-13-0-ci-cd-workflows-and-pgvector-for-prisma-postgres

11 Vector type needed for storing OpenAI embeddings · prisma prisma · Discussion #18220 · GitHub
https://github.com/prisma/prisma/discussions/18220

13 14 15 16 About hybrid search | Vertex AI | Google Cloud Documentation
https://docs.cloud.google.com/vertex-ai/docs/vector-search/about-hybrid-search

17 8 API Security Best Practices For AI Agents | Curity Identity Server
https://curity.io/resources/learn/api-security-best-practice-for-ai-agents/?utm_content=357520214&utm_medium=social&utm_source=linkedin&hss_channel=lcp-25049399

18 19 Handling Third-Party Access Tokens Securely in AI Agents
https://auth0.com/blog/third-party-access-tokens-secure-ai-agents/