Arquitetura v2 — Personal Assistant

Base: Postgres + pgvector, embeddings via API (PT/EN), chat via DeepSeek, voz via Gemini (depois).

Escopo desta versão: documentar a nova arquitetura de forma detalhada, antes de iniciarmos a implementação.

1. Objetivos e princípios

Objetivos

- Memória útil (RAG semântico) para lembrar aprendizados e contexto.
- Agenda prática: recorrências, próximos dias e prioridades.
- Radar: itens sem data para manter no campo de visão.
- Preparar evolução de uso local → VM → multiusuário → app comercial.

Princípios

- Postgres é a fonte da verdade (tarefas, radar, conhecimento e vetores).
- Provedores plugáveis: DeepSeek/Embeddings/Voz/Claude (futuro).
- Separar ingestão (sync) de consulta (search/chat) para controle e custo.
- Privacidade por design: marcação de itens sensíveis e redação/mascara.

2. Componentes do sistema

API (FastAPI)

Serviço único por enquanto: expõe endpoints REST, conversa com Postgres, chama provedores externos quando necessário e monta prompts com contexto.

Banco (Postgres + pgvector)

Um único banco guarda dados operacionais e vetores (RAG): tarefas, radar, itens de conhecimento, chunks e embeddings.

Provedores externos

- **Embeddings Provider** (v2): via API (PT/EN) para gerar vetores de chunks e da query.
- **Chat Provider** (v2): DeepSeek para responder com/sem contexto.
- **Audio Provider** (v2): Gemini native audio para transcrição e normalização (fase posterior).
- **Claude Provider** (futuro): usado na camada Projetos (especialmente para código).

3. Modelo de dados (Postgres)

3.1 Agenda

tasks

- id (pk), user_id, title, notes, priority (1–5)

- kind: one_off | recurring
- due_date (one-off), rrule (recurring RFC5545), start_date
- created_at, updated_at

task_occurrences_done

- id (pk), task_id (fk), user_id
- occurrence_date (date), done_at (timestamp)

Motivo: não duplicar ocorrências futuras; calcula-se on-the-fly e registra-se apenas o que foi concluído.

3.2 Radar

radar_items

- id (pk), user_id, title, notes, priority (1–5)
- status: open | archived
- created_at, updated_at

3.3 Base de conhecimento + vetores

knowledge_items (um arquivo)

- id (pk), user_id, source (localfs | gdrive depois)
- file_path, folder_date, content_text, content_hash
- is_sensitive (bool), last_synced_at, created_at/updated_at

knowledge_chunks (trechos + embedding)

- id (pk), user_id, item_id (fk), chunk_index
- text, text_hash
- embedding (vector/pgvector), embedding_model
- created_at, updated_at

Índices: índice vetorial (ivfflat ou hnsw) + índices por user_id/item_id.

4. Fluxos principais

4.1 Ingestão (Sync)

- Ler diretório montado /app/base_conhecimento (no dev: bind do Mac).
- Para cada arquivo: calcular hash; se mudou, atualizar knowledge_items e preparar (re)chunking.
- Registrar contagens/erros do sync.

Importante: sync não precisa embeddar automaticamente; embeddings podem ser acionados manualmente ou por job.

4.2 Chunking

- Estratégia inicial: 800–1200 caracteres por chunk, com overlap de 150–200.
- Metadados: item_id, file_path, folder_date, chunk_index.
- Evolução futura: chunking semântico por seções do JSON (opcional).

4.3 Embeddings via API (PT/EN)

- Selecionar chunks sem embedding (ou com hash alterado).
- Gerar embeddings em lote e salvar em knowledge_chunks.embedding.
- Respeitar is_sensitive: itens sensíveis não são enviados para a API.

4.4 Busca semântica

- Gerar embedding da query.
- Buscar top-k por similaridade vetorial (pgvector).
- Retornar chunks + metadados para montagem de contexto.

4.5 Chat com contexto (DeepSeek)

- Receber mensagem do usuário.
- Se use_context=true: recuperar top-k chunks e montar bloco CONTEXT com origem (file_path/folder_date).
- Chamar DeepSeek com prompt final e devolver resposta + referências.

Modos: IA pura (sem contexto), IA com contexto, e (futuro) modo de aprovação manual de chunks.

5. Endpoints (v2)

Saúde: GET /health

Agenda: POST /tasks, GET /tasks/today, GET /tasks/next?days=14, POST /tasks/{id}/complete, POST /tasks/{id}/done, GET /agenda/overview?days=14

Radar: POST /radar, GET /radar

Knowledge: POST /knowledge/sync_local, GET /knowledge/items, GET /knowledge/{item_id}, POST /knowledge/chunk, POST /knowledge/embed, GET /knowledge/semantic_search?q=...&k;=...

Chat: POST /chat/preview, POST /chat/respond

6. Multiusuário e autenticação (planejado)

- Tudo carrega user_id (hoje: default).
- Futuro: token/JWT e integração com Telegram/WhatsApp/App para mapear user_id.
- Separar dados por usuário desde já evita retrabalho.

7. Privacidade e mitigação

- Flag is_sensitive em knowledge_items para bloquear envio a embeddings API.
- Redação/mascara (fase futura): remover e-mails, tokens, IPs e segredos antes de embeddar.
- Chunking consciente: enviar trechos mínimos necessários, não dumps completos.

8. Camada Projetos (futuro, compatível com v2)

Projeto = container de trabalho com histórico, decisões, artefatos e um handoff JSON sempre atualizado para alternar entre IAs (ChatGPT/Claude/DeepSeek) sem reexplicar contexto.

Tabelas futuras sugeridas

- projects (id, user_id, nome, objetivo, status, timestamps)
- project_events (timestamp, model, tipo, texto, artefatos)
- project_handoff (json estruturado: estado atual, decisões, próximos passos)

9. Roadmap técnico (ordem prática)

- Migrar SQLite → Postgres + pgvector (compose + models + db session).
- Criar knowledge_items + knowledge_chunks com embedding vector.
- Adaptar sync_local para preencher knowledge_items e (re)criar chunks.
- Gerar embeddings via API e persistir no pgvector.
- Implementar semantic_search e integrar ao chat/respond.
- Depois: voz (Gemini) e camada Projetos.