

MissionAI アーキテクチャ解説

	ステータス: アクティブ (アーキテクチャ概要)
	作成日: 2025-12-11
	用途: アプリケーション全体のアーキテクチャ概要と主要技術スタックの説明

概要

MissionAIは、AI機能を備えた事業計画策定・管理デスクトップアプリケーションです。Next.js + Reactで構築されたフロントエンドと、Rust + Tauriで構築されたバックエンドを組み合わせたハイブリッドアーキテクチャを採用しています。

1. アプリケーション構成

ハイブリッドデスクトップアプリケーション

- フロントエンド: Next.js 14 (App Router) + React 18 + TypeScript
- デスクトップフレームワーク: Tauri 2.0
- バックエンド: Rust (Axum HTTPサーバー)
- 実行環境: TauriのWebView内でNext.jsアプリを実行

通信方式

- Tauri IPC: フロントエンド \leftrightarrow Rust (Tauriコマンド経由)
- HTTP API: フロントエンド \leftrightarrow Rust (Axumサーバー経由、ポート3010/3011)

2. 主要ライブラリ・フレームワーク

フロントエンド (Next.js/React)

コアフレームワーク

- Next.js 14: App Router、SSR/SSG対応
- React 18: UIライブラリ
- TypeScript 5: 型安全性の確保

状態管理・データフェッチング

- @tanstack/react-query 5: サーバー状態管理・キャッシング
- React Context API: グローバル状態管理

UIコンポーネント・エディタ

- react-markdown + remark-gfm: Markdownレンダリング
- @monaco-editor/react: コードエディタ (Monaco Editor)

- **@dnd-kit**: ドラッグ&ドロップ機能
- **react-icons**: アイコンライブラリ

データ可視化ライブラリ

グラフ・チャート

- **D3.js系**
 - **d3-force**: 力指向グラフ
 - **d3-drag**: ドラッグ操作
 - **d3-zoom**: ズーム機能
 - **d3-hierarchy**: 階層構造
 - **d3-selection**: DOM操作
- **react-force-graph-3d**: 3Dグラフ可視化
- **Three.js**: 3Dレンダリングエンジン
- **troika-three-text**: Three.js用テキストレンダリング
- **Vega/Vega-Lite**: 統計グラフ・データ可視化
- **vega-embed**: Vegaグラフの埋め込み

エクスポート機能

- **html2canvas**: HTMLをCanvasに変換
- **jspdf**: PDF生成

バックエンド (Rust)

コアフレームワーク

- **Tauri 2.0**: デスクトップアプリケーションフレームワーク
- **Rust Edition**: 2021

HTTPサーバー

- **Axum 0.7**: 非同期Webフレームワーク
- **Tower 0.4**: ミドルウェアスタック
- **Tower HTTP 0.5**: HTTPミドルウェア (CORS対応)

データベース

- **rusqlite 0.31**: SQLiteバインディング (bundled機能)
- **r2d2 0.8**: コネクションプール
- **r2d2_sqlite 0.24**: SQLite用r2d2アダプター

非同期処理

- **Tokio 1**: 非同期ランタイム (full機能)
- **async-channel 2.0**: 非同期チャネル (書き込みキュー用)

ベクトル検索

- **ChromaDB 2.3.0:** ChromaDBクライアント
- **hnsw_rs 0.3.3:** RustネイティブのHNSW実装（検討中）

その他

- **serde + serde_json:** シリアライゼーション
 - **uuid:** UUID生成
 - **chrono:** 日時処理
 - **bcrypt:** パスワードハッシュ
 - **dotenv:** 環境変数読み込み
 - **reqwest:** HTTPクライアント
 - **csv:** CSVパーサー
 - **dirs:** ホームディレクトリ取得
 - **sha2:** SHAハッシュ
 - **tracing + tracing-subscriber:** 構造化ログ
 - **anyhow:** エラーハンドリング
-

3. データベース構成

SQLite（構造化データ）

役割: メタデータとリレーションナルデータの永続化

管理データ:

- 組織情報 (**organizations**)
- 組織メンバー情報 (**organizationMembers**)
- 事業会社情報 (**companies**)
- 組織・会社表示関係 (**organizationCompanyDisplay**)
- 議事録 (**meetingNotes**)
- 注力施策 (**focusInitiatives**)
- エンティティ (**entities**) - メタデータのみ
- 関係 (**relations**) - メタデータのみ
- トピック (**topics**) - メタデータのみ
- システム設計ドキュメント (**designDocSections, designDocSectionRelations**)

特徴:

- ACIDトランザクション保証
- リレーションナルデータの管理
- 構造化クエリ（JOIN、集計など）
- ChromaDB同期状態の管理 (**chromaSynced, chromaSyncError, lastChromaSyncAttempt**)
- コネクションプールによる効率的な接続管理
- 書き込みキューによるデータロック回避

ChromaDB (ベクトル検索)

役割: ベクトル検索とセマンティック検索

コレクション構造:

- `entities_{organizationId}`: エンティティの埋め込みベクトル
- `relations_{organizationId}`: 関係の埋め込みベクトル
- `topics_{organizationId}`: トピックの埋め込みベクトル
- `design_docs`: システム設計ドキュメントの埋め込みベクトル

特徴:

- 高次元ベクトルの保存と検索
- セマンティック類似度検索
- メタデータとベクトルの組み合わせ検索
- SQLiteとは独立したデータストア
- HNSWアルゴリズムによる高速な近似最近傍検索
- 組織ごとにコレクションを分離（マルチテナント対応）

パフォーマンス:

- 10,000件: 約0.05秒
- 100,000件: 約0.2秒
- 1,000,000件: 約1-2秒

4. フロントエンド構成

ディレクトリ構造

```
app/                                # Next.js App Router
  └── page.tsx                      # ホームページ
  └── layout.tsx                    # ルートレイアウト
  └── globals.css                   # グローバルスタイル
  └── organization/
    └── page.tsx                  # 組織管理ページ
    └── detail/
      └── initiative/          # 組織一覧
      └── companies/           # 組織詳細
      └── knowledge-graph/     # 注力施策
    └── companies/                # 事業会社管理ページ
    └── knowledge-graph/         # ナレッジグラフ可視化
  └── rag-search/                  # RAG検索
  └── analytics/                  # 分析ページ
  └── design/                     # システム設計ドキュメント
  └── settings/                  # 設定ページ

components/                            # Reactコンポーネント
  └── Layout.tsx                   # メインレイアウト
  └── KnowledgeGraph3D.tsx        # 3Dグラフ可視化
  └── RelationshipDiagram2D.tsx  # 2Dグラフ可視化
```

```

    └── RelationshipBubbleChart.tsx # バブルチャート
    └── VegaChart.tsx           # Vegaグラフ
    └── AIAssistantPanel.tsx   # AIアシスタントパネル
    └── QueryProvider.tsx     # React Queryプロバイダー
    └── ErrorBoundary.tsx      # エラーバウンダリー
    └── ...
    ...
lib/
    ├── apiClient.ts          # ビジネスロジック・APIクライアント
    ├── localFirebase.ts       # Rust APIサーバー用クライアント
    ├── orgApi.ts             # Tauriコマンドラッパー (Firebase互換API)
    ├── entityApi.ts          # 組織管理API
    ├── relationApi.ts        # エンティティAPI
    ├── companiesApi.ts       # 関係API
    ├── embeddings.ts          # 事業会社API
    ├── knowledgeGraphRAG.ts  # 埋め込み生成
    ├── designDocRAG.ts       # ナレッジグラフRAG
    └── ...
    ...

```

状態管理

サーバー状態

- **React Query (@tanstack/react-query):**
 - サーバー状態のキャッシング
 - 自動リフェッチ
 - 楽観的更新
 - エラーハンドリング

グローバル状態

- **React Context API:**
 - [EmbeddingRegenerationContext](#): 埋め込み再生成の状態管理
 - [QueryProvider](#): React Queryの設定

ローカル状態

- [useState](#): コンポーネント内の状態
- [useReducer](#): 複雑な状態ロジック

5. バックエンド構成 (Rust/Tauri)

ディレクトリ構造

```

src-tauri/src/
    └── main.rs
                    # エントリーポイント
                    # - データベース初期化

```

```

# - ChromaDB Server起動
# - HTTP APIサーバー起動
# - 書き込みワーカー起動

commands/          # Tauriコマンド (IPC)
└── mod.rs        # コマンド登録
└── db.rs         # データベース操作
└── chromadb.rs   # ChromaDB操作
└── organization.rs # 組織管理
└── companies.rs   # 事業会社管理
└── design_doc.rs  # 設計ドキュメント管理
└── app.rs         # アプリ情報
└── fs.rs          # ファイル操作

database/          # データベース層
└── mod.rs        # 初期化・設定
└── pool.rs       # コネクションプール管理
└── store.rs      # データアクセス層
└── chromadb.rs   # ChromaDB統合
└── organization.rs # 組織データアクセス
└── companies.rs   # 事業会社データアクセス
└── ...

```

```

api/               # HTTP APIサーバー
└── server.rs     # Axumサーバー起動
└── routes.rs     # ルーティング定義
└── handlers.rs   # リクエストハンドラー
└── mod.rs         # APIモジュール

db/                # 書き込みキューシステム
└── write_worker.rs # 書き込みワーカー
└── write_job.rs   # ジョブ定義

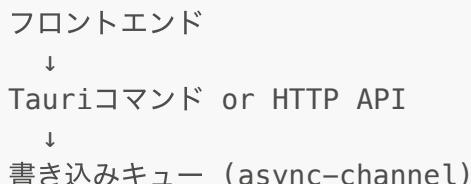
```

初期化フロー

- データベース初期化:** SQLiteデータベースの作成・マイグレーション
- 書き込みワーカー起動:** 単一の書き込みワーカーを起動
- ChromaDB Server起動:** PythonプロセスでChromaDB Serverを起動（ポート8000）
- HTTP APIサーバー起動:** Axumサーバーを起動（ポート3010/3011）

6. データフロー

書き込み処理





特徴:

- すべての書き込み操作はキューを経由
- 単一の書き込みワーカーが順次処理
- データロックを回避
- トランザクション整合性を保証

読み取り処理

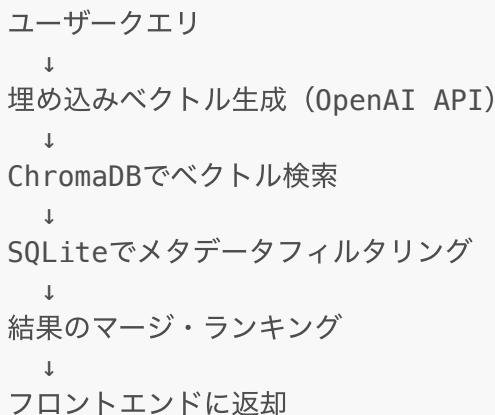
SQLite

- コネクションプールから直接読み取り
- 複数の読み取り操作を並列実行可能
- 書き込みロックの影響を受けない

ChromaDB

- Rust側のTauriコマンド経由で検索
- ChromaDB Server (Python) にHTTPリクエスト
- ベクトル類似度検索を実行

ハイブリッド検索フロー



7. セキュリティ・認証

認証システム

- **認証方式:** ローカルSQLiteデータベースにユーザー情報を保存
- **パスワードハッシュ:** bcryptアルゴリズム

- **セッション管理:** Tauriアプリ内で管理 (Firebase不使用)

データセキュリティ

- **ローカルストレージ:** すべてのデータはローカルに保存
- **暗号化:** パスワードはbcryptでハッシュ化
- **アクセス制御:** Tauriアプリ内でのみアクセス可能

8. 開発環境

ポート設定

- **Next.js開発サーバー:** ポート3010
- **Rust APIサーバー:**
 - 開発環境: ポート3010 (環境変数で変更可能)
 - 本番環境: ポート3011 (環境変数で変更可能)
- **ChromaDB Server:** ポート8000

ビルド・実行

開発モード

```
# Next.js開発サーバー起動
npm run dev

# Tauri開発モード (Next.js + Tauri)
npm run tauri:dev
```

本番ビルド

```
# Next.jsビルド
npm run build

# Tauriアプリビルド
npm run tauri:build
```

環境変数

- **.env** または **local.env**: 環境変数ファイル
- **API_SERVER_PORT**: APIサーバーのポート番号
- **NEXT_PUBLIC_API_SERVER_PORT**: フロントエンドから参照するAPIサーバーポート

9. 特徴的な設計パターン

1. データロック回避設計

問題: SQLiteの書き込みロックによるパフォーマンス問題

解決策:

- 書き込みキューシステムを実装
- すべての書き込み操作を単一のワーカーに集約
- 読み取り操作は並列実行可能

2. ハイブリッド検索アーキテクチャ

構成:

- SQLite: メタデータの構造化検索
- ChromaDB: セマンティック検索

メリット:

- 高速な構造化クエリ (SQLite)
- 意味的な類似度検索 (ChromaDB)
- 両方の結果を組み合わせた高精度な検索

3. マルチテナント対応

実装:

- ChromaDBコレクションを組織ごとに分離
- `entities_{organizationId}`形式でコレクション名を命名
- データの完全な分離を実現

4. フォールバック機構

HTTP API → Tauriコマンド:

- Rust APIサーバーが応答しない場合
- 自動的にTauriコマンドにフォールバック
- タイムアウト: 1秒

5. 埋め込みベクトル管理

二重保存:

- SQLite: メタデータのみ (軽量)
- ChromaDB: 埋め込みベクトル (検索用)

同期状態管理:

- `chromaSynced`: 同期済みフラグ
- `chromaSyncError`: 同期エラー情報
- `lastChromaSyncAttempt`: 最終同期試行時刻

10. パフォーマンス最適化

フロントエンド

- **動的インポート**: 重いコンポーネントを動的インポート (`next/dynamic`)
- **React Queryキャッシング**: サーバー状態の自動キャッシング
- **コード分割**: Next.jsの自動コード分割

バックエンド

- **コネクションプール**: SQLite接続の再利用
- **書き込みキュー**: 書き込み操作の最適化
- **非同期処理**: Tokioによる非同期I/O

データベース

- **インデックス**: SQLiteテーブルに適切なインデックスを設定
 - **HNSWアルゴリズム**: ChromaDBの高速ベクトル検索
 - **コレクション分離**: 組織ごとのコレクションで検索範囲を限定
-

11. 関連ドキュメント

アーキテクチャ関連

- [ポート・サーバー設計](#)
- [ポート設計の懸念点](#)
- [ポート設計レビュー](#)

データベース関連

- [データベース設計](#)
- [埋め込みベクトルの保存場所](#)
- [データ同期のリスク分析](#)

Rust/Tauri関連

- [Rust/Tauri設定](#)
- [API仕様](#)

ChromaDB関連

- [ChromaDB統合計画](#)
- [ChromaDB使用ガイド](#)

RAG検索関連

- [RAG検索のデータベースフロー](#)
 - [RAG検索の改善](#)
-

12.まとめ

MissionAIは、以下の特徴を持つモダンなデスクトップアプリケーションです：

1. **ハイブリッドアーキテクチャ:** Next.js + React (フロントエンド) + Rust + Tauri (バックエンド)
2. **二重データベース:** SQLite (構造化データ) + ChromaDB (ベクトル検索)
3. **高性能:** 書き込みキュー、コネクションプール、非同期処理
4. **AI機能:** RAG検索、セマンティック検索、埋め込みベクトル管理
5. **マルチテナント:** 組織ごとのデータ分離
6. **開発者フレンドリー:** TypeScript、構造化ログ、エラーハンドリング

このアーキテクチャにより、デスクトップアプリとして動作しながら、Web技術の利点を活かし、AI機能を統合した事業計画策定・管理システムを実現しています。